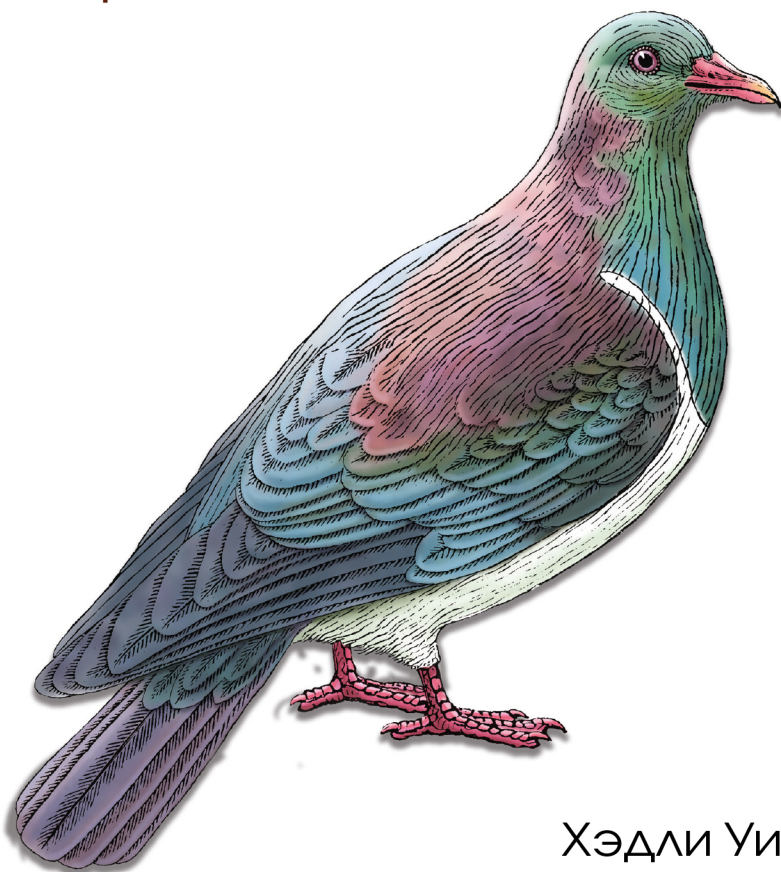


O'REILLY®

Изучаем Shiny

Строим интерактивные
приложения, отчеты и дашборды
с помощью языка R



Хэдли Уикем

Хэдли Уикем

Изучаем Shiny

Mastering Shiny

**Build Interactive Apps, Reports,
and Dashboards Powered by R**

Hadley Wickham

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Изучаем Shiny

**Создание интерактивных приложений,
отчетов и дашбордов при помощи R**

Хэдли Уикем



Москва, 2022

УДК 004.42
ББК 32.973
У35

Уикем Х.

У35 Изучаем Shiny / пер. с англ. А. Ю. Гинько. – М.: ДМК Пресс, 2022. – 374 с.: ил.

ISBN 978-5-97060-964-4

Эта книга знакомит читателей с фреймворком Shiny, который существенно облегчает работу программистам при создании интерактивных веб-приложений на языке R. В начале руководства описываются структура приложения и важные компоненты пользовательского интерфейса. Далее представлены способы решения распространенных задач, включая взаимодействие с пользователем, загрузку и скачивание данных, создание пользовательского интерфейса при помощи кода. Также рассматриваются углубленная теория и практика реактивного программирования.

Издание будет полезно разработчикам R, планирующим перейти от базового анализа к полноценным интерактивным веб-приложениям, а также разработчикам Shiny, желающим улучшить свои навыки владения этим инструментом для написания более быстрых и эффективных приложений.

УДК 004.42
ББК 32.973

Authorized Russian translation of the English edition of Mastering Shiny ISBN 9781492047384. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same. Russian language edition copyright © 2022 by ДМК Пресс. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-492-04738-4 (англ.)
ISBN 978-5-97060-964-4 (рус.)

© Hadley Wickham, 2021
© Перевод, оформление, издание,
ДМК Пресс, 2022

Содержание

Телеграм канал: https://t.me/it_boooks

От издательства	14
Введение	15
Благодарности	20
Как была написана эта книга	21
Об изображении на обложке	22
 Часть I. ПРИСТУПАЕМ К РАБОТЕ	 23
Глава 1. Ваше первое приложение Shiny	24
Введение	24
Создание директории и файла приложения	24
Запуск и остановка	25
Добавление элементов пользовательского интерфейса	27
Добавление поведения	28
Снижение дублирования кода при помощи реактивных выражений	29
Заключение	30
Упражнения	31
 Глава 2. Основы интерфейса пользователя	 35
Введение	35
Элементы ввода	35
Базовая структура	36
Текст	36
Числовой ввод	37
Даты	38
Ограниченный выбор	39
Загрузка файлов	41
Кнопки	42
Упражнения	43
Элементы вывода	43
Текст	44
Таблицы	45
Графики	47
Загрузка файлов	48
Упражнения	48
Заключение	49

Глава 3. Основы реактивного программирования	50
Введение	50
Функция <code>server</code>	50
Input	51
Output	52
Реактивное программирование	53
Императивное программирование против декларативного	55
Ленивые вычисления	55
Реактивный график	56
Реактивные выражения	57
Порядок выполнения	57
Упражнения	58
Реактивные выражения	59
Предпосылки	60
Приложение	62
Реактивный график	63
Упрощение реактивного графика	65
Зачем нужны реактивные выражения?	67
Контроль времени запуска реактивных выражений	68
Обновление по расписанию	69
Щелчки мыши	70
Наблюдатели	73
Заключение	74
 Глава 4. Практический пример: несчастные случаи	75
Введение	75
Данные	75
Описание	77
Прототип	81
Доработка таблиц	83
Процент против количества	85
Истории получения травмы	87
Упражнения	89
Заключение	89
 Часть II. SHINY В ДЕЙСТВИИ	90
 Глава 5. Рабочий процесс	91
Рабочий процесс разработки приложения	91
Создание приложения	92
Отслеживание изменений	92
Управление запуском приложения	94
Отладка	94
Чтение трассировки	95
Трассировка в Shiny	96
Интерактивный отладчик	98

Практический пример	100
Отладка реактивных выражений	104
Получение помощи	106
Основы воспроизводимых примеров	106
Создание воспроизводимого примера	107
Минимальный воспроизводимый пример	108
Практический пример	109
Заключение	113
Глава 6. Макеты, темы, HTML	114
Введение	114
Одностраничные макеты	114
Функции страницы	115
Страница с боковой панелью	116
Многострочный вывод	118
Упражнения	119
Многостраничные макеты	119
Наборы вкладок	119
Навигационный список и навигационная панель	121
Bootstrap	122
Темы	123
Подготовка	124
Темы Shiny	124
Темы графиков	125
Упражнения	126
За кулисами	126
Заключение	128
Глава 7. Графики	129
Интерактивность	129
Основы	129
Щелчки мыши	131
Другие события мыши	133
Выделение прямоугольной области	133
Изменение графика	135
Ограничения интерактивности	138
Динамическая ширина и высота	139
Изображения	141
Заключение	143
Глава 8. Обратная связь с пользователем	144
Проверка значений	144
Проверка ввода	145
Отмена выполнения при помощи функции req()	146
Функция req() и проверка данных	149
Проверка элементов вывода	151

Оповещения.....	152
Временные оповещения.....	152
Оповещения о выполнении процесса	153
Обновляемые оповещения.....	154
Индикатор хода выполнения задачи	155
Shiny.....	155
Waiter	157
Вращающийся индикатор прогресса (спиннер)	158
Подтверждение и отмена действий	161
Явное подтверждение	161
Отмена действия.....	163
Корзина	165
Заключение	165
Глава 9. Загрузка и скачивание файлов.....	166
Загрузка	166
Интерфейс пользователя.....	166
Серверная часть	167
Загрузка данных.....	168
Скачивание	169
Основы.....	169
Скачивание данных.....	170
Скачивание отчетов	171
Практический пример.....	174
Упражнения	176
Заключение	178
Глава 10. Динамический интерфейс пользователя	179
Обновление элементов ввода	179
Простое использование	181
Иерархические выпадающие списки	182
Заморозка реактивного ввода	185
Циклические ссылки	186
Взаимосвязанные элементы ввода	187
Упражнения	188
Динамическая видимость.....	189
Условный интерфейс пользователя	190
Интерфейс мастера	192
Упражнения	193
Создание интерфейса пользователя при помощи кода.....	194
Введение.....	194
Множественные элементы управления	196
Динамическая фильтрация	198
Диалоговые окна.....	203
Упражнения	203
Заключение	204

Глава 11. Закладки	206
Основная идея	206
Обновление ссылки	209
Сохранение состояния в файл	210
Сложности при сохранении закладок.....	210
Упражнения	211
Заключение	212
Глава 12. Tidy eval	213
Предпосылки	213
Маскирование данных	215
Введение.....	215
Пример: ggplot2	216
Пример: dplyr	219
Пользовательские данные	221
Почему бы не использовать базовый синтаксис R?	223
Tidy-Selection	223
Косвенная адресация	224
Tidy-Selection и маскирование данных.....	224
Функции parse() и eval()	225
Заключение	226
Часть III. ОСВАИВАЕМ РЕАКТИВНОСТЬ	227
Глава 13. Зачем нужна реактивность?	228
Введение.....	228
Зачем нужно реактивное программирование?.....	229
Почему нельзя использовать переменные?	229
А как насчет функций?.....	230
Событийно-ориентированное программирование	230
Реактивное программирование	232
Краткая история реактивного программирования	233
Заключение	234
Глава 14. Реактивный график	235
Введение	235
Пошаговое реактивное выполнение	235
Начало сессии.....	236
Начало выполнения первого элемента вывода	237
Чтение реактивного выражения	238
Чтение элемента ввода	238
Окончание выполнения реактивного выражения.....	239
Окончание выполнения элемента вывода	239
Начало выполнения второго элемента вывода.....	240
Завершение процесса выполнения, вывод расчетов	240
Изменение элемента ввода	241

Инвалидация ввода	241
Оповещение зависимостей	242
Удаление текущих связей	242
Повторное выполнение	243
Упражнения	244
Динамизм	245
Пакет reactlog	246
Заключение	248

Глава 15. Строительные блоки реактивного

программирования	249
Реактивные значения	249
Упражнения	251
Реактивные выражения	251
Ошибки	251
on.exit()	252
Упражнения	252
Наблюдатели и элементы вывода	253
Изолирование кода	255
isolate()	255
observeEvent() и eventReactive()	256
Упражнения	256
Инвалидация по времени	257
Опрос	257
Долгоиграющие реактивы	258
Точность таймера	259
Упражнения	260
Заключение	260

Глава 16. Отхождение от графика

Введение	261
Что не охватывает реактивный график?	261
Практические примеры	263
Один элемент вывода изменяется посредством нескольких элементов ввода	263
Накапливание ввода	264
Приостановка анимации	265
Упражнения	266
Антишаблоны	267
Заключение	268

Часть IV. ЭФФЕКТИВНЫЕ ПРИЕМЫ

Глава 17. Общие принципы	271
Введение	271
Организация кода	272

Тестирование	273
Управление зависимостями	273
Контроль версий исходного кода.....	274
Непрерывная интеграция/развертывание.....	275
Анализ кода.....	276
Заключение	277
Глава 18. Функции	278
Организация файлов	279
Функции интерфейса пользователя.....	279
Другое применение	280
Функциональное программирование	281
Интерфейс пользователя в виде структуры данных	281
Серверные функции	282
Чтение загруженных данных	282
Внутренние функции	284
Заключение	284
Глава 19. Модули Shiny	286
Предпосылки	286
Основы модульной системы	288
Интерфейс модуля.....	289
Серверная логика модуля	289
Обновленное приложение.....	290
Пространства имен.....	291
Соглашение об именовании.....	292
Упражнения	292
Ввод и вывод.....	293
Приступим: интерфейсный ввод и серверный вывод	294
Практический пример: выбор числовой переменной.....	295
Серверный ввод	297
Вложенные модули.....	298
Практический пример: гистограмма.....	298
Множественный вывод.....	300
Упражнения	302
Практические примеры	303
Ограниченный выбор вариантов и пункт Другие	303
Мастер	306
Динамический интерфейс пользователя.....	310
Модули в виде единого объекта.....	312
Заключение	314
Глава 20. Пакеты	315
Преобразование существующего приложения	316
Один файл	316
Модульная структура.....	318

Пакет	319
Преимущества	320
Рабочий процесс	320
Совместное использование	321
Дополнительные шаги	322
Развертывание приложения-пакета	322
R CMD check	322
Заключение	324
Глава 21. Тестирование	325
Тестирование функций	326
Базовая структура	327
Основной рабочий процесс	327
Основные функции ожидания	328
Функции интерфейса пользователя	330
Рабочий процесс	332
Покрытие кода	332
Сочетания клавиш	333
Резюме по рабочему процессу	333
Тестирование реактивов	334
Модули	335
Ограничения	337
Тестирование JavaScript	337
Основные операции	338
Практический пример	340
Тестирование визуальных элементов	342
Философия	343
Когда стоит писать тесты?	343
Заключение	344
Глава 22. Безопасность	345
Данные	346
Вычислительные ресурсы	347
Глава 23. Производительность	350
Ужин в ресторане Shiny	351
Оценка производительности	352
Запись	352
Запуск	353
Анализ	354
Профилирование	356
Огненный график	356
Профилирование кода R	358
Профилирование приложения Shiny	359
Ограничения	360
Повышение производительности	361

Кеширование	361
Основы.....	362
Кеширование реактивов.....	362
Кеширование графиков	364
Ключи кеширования	365
Область видимости кеша	366
Другие способы оптимизации	366
Запланированные преобразования данных.....	366
Управление ожиданиями пользователя.....	367
Заключение	368
Предметный указатель.....	369

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и O'Reilly очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Введение

Если вы никогда прежде не использовали Shiny, добро пожаловать! *Shiny* представляет собой фреймворк языка программирования R, позволяющий с легкостью создавать функциональные интерактивные веб-приложения. С помощью Shiny вы можете перенести свою работу в R и представить ее результаты в браузере, чтобы все могли свободно ими пользоваться. Shiny способствует разработке сложных и эффективных веб-приложений с минимумом усилий.

В прошлом разработка веб-приложений на R давалась программистам весьма непросто, и на то было две основные причины:

- им нужно было в полной мере владеть современными веб-технологиями, включая языки HTML, CSS и JavaScript;
- в сложных интерактивных приложениях им приходилось внимательно отслеживать все связи между элементами, чтобы изменения входных значений влияли только на связанные с ними выходные.

Фреймворк Shiny значительно облегчает работу программистам при создании веб-приложений за счет:

- предоставления тщательно проработанного набора функций пользовательского интерфейса для автоматического генерирования кода HTML, CSS и JavaScript, необходимого для решения конкретных задач. Это означает, что вам не понадобится доскональное знание этих языков программирования и разметки, пока вам не станет тесно в рамках предоставляемых Shiny возможностей;
- применения нового стиля программирования, получившего название реактивное. С помощью этой концепции можно легко отслеживать и поддерживать зависимости между фрагментами кода. На практике это означает, что при изменении значения входного элемента Shiny автоматически определит, как с наименьшими усилиями обновить все связанные выходные элементы.

Разработчики используют Shiny для:

- создания дашбордов, помогающих в отслеживании высокоуровневых показателей с возможностью проводить детализированный анализ при необходимости;
- замены сотен страниц в формате PDF на одно интерактивное приложение, позволяющее пользователю переключаться между нужными ему результатами;
- донесения информации о сложных моделях до аудитории, не обладающей техническими знаниями, в виде информативных визуализаций и средств интерактивного анализа;
- автоматизации анализа данных в общих рабочих процессах с заменой процесса обмена электронными сообщениями на приложение Shiny, позволяющее пользователям загружать свои данные и выполнять

стандартный анализ. Таким образом можно сделать доступным продвинутый анализ в R пользователям, не обладающим навыками программирования;

- создания интерактивных демонстраций при обучении статистике и концепциям науки о данных, позволяющих студентам менять входные значения и наблюдать за изменениями в итоговом анализе.

Иными словами, с Shiny вы с легкостью можете делегировать некоторые свои суперспособности в R всякому, у кого есть доступ в интернет.

Для кого предназначена эта книга?

Прочитать данную книгу стоит двум основным группам аудитории:

- разработчикам R, заинтересованным в освоении фреймворка Shiny с целью перехода от базового анализа к полноценным интерактивным веб-приложениям. Чтобы взять от книги все, вам необходимо обладать определенным опытом анализа данных при помощи языка R и написания функций;
- разработчикам Shiny, желающим улучшить свои навыки владения этим инструментом для написания более быстрых и эффективных приложений. Вам будет особенно полезна эта книга, если ваши приложения постепенно начинают разрастаться и вам становится все сложнее контролировать происходящие в них процессы.

Что вы узнаете из этой книги?

Книга поделена на четыре части.

1. В первой части мы познакомимся с основами фреймворка Shiny, чтобы вы могли как можно быстрее написать свое первое приложение. Мы поговорим о структуре приложения, полезных компонентах пользовательского интерфейса и основах реактивного программирования.
2. Во второй части книги мы сделаем один шаг вперед и познакомимся со способами решения распространенных задач, включая взаимодействие с пользователем, загрузку и скачивание данных, создание пользовательского интерфейса при помощи кода, сокращение дублирующихся фрагментов кода и использование Shiny совместно с tidyverse.
3. Третья часть будет посвящена углубленной теории и практике реактивного программирования – базовой парадигмы, лежащей в основе Shiny. Если вы уже работаете с этим фреймворком, вы сможете извлечь максимум выгоды из данной главы, поскольку она закладывает теоретический фундамент, который поможет вам в разработке сложных интерактивных приложений, предназначенных для решения широкого спектра задач.
4. В заключительной части книги мы завершим исследование полезных техник, призванных повысить эффективность ваших приложений Shiny.

Вы узнаете, как выполнять процесс декомпозиции сложного приложения на функции и модули, использовать пакеты для лучшей организации вашего кода, тестировать свои работы на предмет наличия ошибок, а также измерять и улучшать производительность приложений.

ЧЕГО ВЫ НЕ УЗНАЕТЕ ИЗ ЭТОЙ КНИГИ?

Данная книга целиком и полностью посвящена созданию эффективных приложений с помощью Shiny и принципам лежащего в ее основе реактивного программирования. Я сделаю все возможное, чтобы показать вам лучшие приемы в работе с данными, программировании на языке R и инженерии программного обеспечения, но вам придется почитать и другие источники, чтобы в полной мере овладеть всеми этими навыками. Если вам нравится, как я пишу, вы можете посмотреть и другие мои книги на эту тему: *R for Data Science* (<https://r4ds.had.co.nz>), *Advanced R* (<https://adv-r.hadley.nz>) и *R Packages* (<https://r-pkgs.org>).

Также есть несколько тем из области Shiny, которых я не буду касаться в данной книге:

- при написании приложений мы будем рассматривать только встроенный набор инструментов пользовательского интерфейса (user interface toolkit). Может, он и не самый привлекательный, но для обучения очень даже подойдет. Если вам нужно больше или просто надоел базовый интерфейс, на просторах интернета есть множество пакетов, позволяющих до неузнаваемости изменить внешний вид ваших приложений. Подробнее читайте в разделе книги, посвященном фреймворку *Bootstrap*;
- тема развертывания приложений Shiny выходит за рамки данной книги, поскольку здесь многое зависит от сторонних факторов, не имеющих отношения к R, – больше культурного и организационного свойства, но никак не технического. Если вы делаете первые шаги в вопросах развертывания приложений Shiny, я бы посоветовал посмотреть выступление Джо Ченга по адресу <https://www.rstudio.com/resources/rstudioconf-2019/shiny-in-production-principles-practices-and-tools/>. Из него вы почерпнете все базовые вещи, связанные с развертыванием приложений, включая распространенные проблемы и способы их решения. После этого я бы рекомендовал посетить страницу <https://www.rstudio.com/products/connect>, посвященную продукту RStudio Connect, предназначенному для развертывания приложений в рамках организации, а также соответствующий раздел на сайте Shiny по адресу <https://shiny.rstudio.com/articles/#deployment>.

ТРЕБОВАНИЯ

Перед тем как продолжить, убедитесь в том, что у вас на компьютере стоит все необходимое программное обеспечение для работы:

R

Если у вас еще не установлен R на компьютере, возможно, вы читаете не ту книгу. Мы на протяжении книги будем предполагать, что у вас есть базовые знания о языке R. Если вы хотите узнать, как использовать R, я бы рекомендовал для чтения книгу *R for Data Science* (<https://r4ds.had.co.nz>), которая позволит вам сделать первые шаги в этой новой для вас среде.

RStudio

RStudio представляет собой бесплатную *интегрированную среду разработки* (integrated development environment – IDE) для R. И хотя вы можете создавать и использовать приложения на Shiny в любом окружении R, включая R GUI и ESS, именно RStudio обладает некоторыми полезными особенностями, облегчающими написание, отладку и развертывание приложений Shiny. Мы рекомендуем вам загрузить RStudio Desktop по адресу <https://www.rstudio.com/products/rstudio/download/> и дать ему шанс. В то же время это совершенно не обязательно для разработки приложений и чтения книги.

Пакеты R

В этой книге мы будем использовать множество пакетов R. Вы можете установить их все сразу, запустив следующий код:

```
install.packages(c(
  "gapminder", "ggforce", "gh", "globals", "openintro", "profvis",
  "RSQLite", "shiny", "shinycssloaders", "shinyFeedback",
  "shinythemes", "testthat", "thematic", "tidyverse", "vroom",
  "waiter", "xml2", "zeallot"
))
```

Если вы уже загружали пакет Shiny ранее, убедитесь, что у вас установлена его версия не ниже 1.6.0.

ПРИНЯТЫЕ В КНИГЕ ОБОЗНАЧЕНИЯ

При написании книги мы использовали следующие обозначения:

- *курсив* – обозначает новые термины, ссылки, электронные адреса, имена и расширения файлов;
- моноширинный шрифт – используется в листингах и при обозначении программных элементов, таких как имена переменных или функций, баз данных, типов данных, переменных окружения, выражений и ключевых слов.

ИСПОЛЬЗОВАНИЕ ПРИМЕРОВ КОДА

Сопроводительные материалы (фрагменты кода, упражнений и т. д.) доступны для загрузки по адресу <https://mastering-shiny.org/>. На все листинги в книге распространяется лицензия MIT (MIT License): <https://www.mit.edu/~amini/LICENSE.md>.

Если у вас есть технические вопросы относительно фрагментов кода, можете написать нам по адресу bookquestions@oreilly.com.

Вы вправе использовать материалы из книги для решения своих задач. В основном код, представленный в данной книге, можно использовать в собственных программах и документации. Вы не обязаны обращаться к нам за разрешением, если речь идет не о копировании большого фрагмента текста. Например, использование нескольких фрагментов из книги при написании своей программы не потребует специального разрешения. В то же время продажа или распространение материалов, принадлежащих O'Reilly, предполагает наличие разрешения. Ответы на вопросы с использованием цитирования из этой книги не требуют разрешения. А включение большей части текста из этой книги в свою документацию – требует.

Мы будем признательны, если вы будете ссылаться на книгу при цитировании, хотя и не требуем от вас этого. Обычно такие ссылки включают в себя название книги, имя автора и название издательства, а также ISBN. Например, «Mastering Shiny by Hadley Wickham (O'Reilly). Copyright 2021 Hadley Wickham, 978-1-492-04738-4».

Для получения разрешений можно обратиться по адресу permissions@oreilly.com.

Благодарности

Данная книга была написана в открытую, а по ее окончании главы свободно выкладывались в Twitter. Таким образом, книгу можно назвать полноценным совместным опытом, а в процессе ее написания очень многие люди вычитывали черновики, исправляли опечатки, предлагали улучшения и вносили свой вклад в содержимое. Без этой неоценимой помощи книга никогда не стала бы такой, какой вы ее видите, и я очень признателен всем, кто внес свою лепту в ее написание.

Мы хотели бы поблагодарить поименно всех 83 человек, предложивших свои улучшения на GitHub (в алфавитном порядке): Adam Pearce (@1wheel), Adi Sarid (@adisarid), Alexandros Melemenidis (@alex-m-ffm), Anton Klåvus (@antonvsdata), Betsy Rosalen (@betsyrosalen), Michael Beigelmacher (@brooklynbagel), Bryan Smith (@BSCowboy), clau6io_hh (@clau6io), @canovasjm, Chris Beeley (@ChrisBeeley), @chsafouane, Chuliang Xiao (@ChuliangXiao), Conor Neilson (@condwanaland), @d-edison, Dean Attali (@daattali), Daniel-David521 (@Danieldavid521), David Granjon (@DivadNojnarg), Eduardo Vásquez (@edovtp), Emil Hvitfeldt (@EmilHvitfeldt), Emilio (@emilopezcano), Emily Riederer (@emilyriederer), Eric Simms (@esimms999), Federico Marini (@federicomarini), Frederik Kok Hansen (@fkoh111), Frans van Dunné (@FvD), Giorgio Comai (@giocomai), Hedley (@heds1), Henning (@henningsway), Hlynur (@hlynurhallgrims), @hsm207, @jacobxk, James Pooley (@jamespooley), Joe Cheng (@jcheng5), Julien Colomb (@jcolomb), Juan C. Rodriguez (@jcrodriguez1989), Jennifer (Jenny) Bryan (@jennybc), Jim Hester (@jimhester), Joachim Gassen (@joachim-gassen), Jon Calder (@jonmcalder), Jonathan Carroll (@jonocarroll), Julian Stanley (@julianstanley), @jyuu, @kaanpekkel, Karandeep Singh (@kdpsingh), Robert Kirk DeLisle (@KirkDCO), Elaine (@loomalaine), Malcolm Barrett (@malcolmbarrett), Marly Gotti (@marlycormar), Matthew Wilson (@MattW-Geospatial), Matthew T. Warkentin (@mattwarkentin), Mauro Lepore (@maurolepore), Maximilian Rohde (@maxdrohde), Matthew Berginski (@mbergins), Michael Dewar (@michael-dewar), Mine Cetinkaya-Rundel (@mine-cetinkaya-rundel), Maria Paula Caldas (@mpaulacaldas), nthobservation (@nthobservation), Pietro Monticone (@pitmonticone), psychometrician (@psychometrician), Ram Thapa (@raamthapa), Janko Thyson (@rappster), Rebecca Janis (@rbjanis), Tom Palmer (@remlapmot), Russ Hyde (@russHyde), Barret Schloerke (@schloerke), Scott (@scottyd22), Matthew Sedaghatfar (@sedaghatfar), Shixiang Wang (@ShixiangWang), Praer (Suthira Owlarn) (@sowla), Sébastien Rochette (@statnmap), @stevensbr, André Calero Valdez (@Sumidu), Tanner Stauss (@tmstauss), Tony Fujs (@tonyfujs), Stefan Moog (@trekonom), Jeff Allen (@trestletech), Trey Gilliland (@treygilliland), Albrecht (@Tungurahua), Valeri Voev (@ValeriVoev), Vickus (@Vickusr), William Doane (@WilDoane), 黄湘云 (@XiangyunHuang) и gXcloud (@xwydq).

Как была написана эта книга

Книга была написана в RStudio с использованием пакета bookdown (<http://bookdown.org>).

При написании книги использовался R версии 4.0.3 (2020-10-10) и следующие версии пакетов:

Пакет	Версия	Источник
gapminder	0.3.0	standard (@0.3.0)
Ggforce	0.3.2	standard (@0.3.2)
Gh	1.2.0	standard (@1.2.0)
Globals	0.14.0	standard (@0.14.0)
Openintro	2.0.0	standard (@2.0.0)
Profvis	0.3.7.9000	GitHub (rstudio/profvis@ca1b272)
RSQLite	2.2.3	standard (@2.2.3)
Shiny	1.6.0	standard (@1.6.0)
shinycssloaders	1.0.0	standard (@1.0.0)
shinyFeedback	0.3.0	standard (@0.3.0)
shinythemes	1.2.0	standard (@1.2.0)
Testthat	3.0.2.9000	Hub (r-lib/testthat@4793514)
Thematic	0.1.1	GitHub (rstudio/thematic@d78d24a)
Tidyverse	1.3.0	standard (@1.3.0)
Vroom	1.3.2	standard (@1.3.2)
Waiter	0.2.0	standard (@0.2.0)
xml2	1.3.2	standard (@1.3.2)
Zeallot	0.1.0	standard (@0.1.0)

Об изображении на обложке

На обложке книги изображен кереру, или новозеландский плодоядный голубь (*kererū*, лат. *Hemiphaga novaeseelandiae*) – единственная птица семейства голубиных, встречающаяся на территории Новой Зеландии.

Оперение новозеландского голубя имеет бронзовый окрас, а его тело и голова окрашены в глянцевые пурпурно-зеленые оттенки. Брюшко у птицы белое, а клюв – под стать глазам – красный. Новозеландский голубь издает мягкие воркующие звуки, а при взлете и приземлении отчетливо и звучно бьет крыльями. Обычно стройный, изящный и активный голубь набирает массу в сезон фруктов, а также в период размножения.

Поскольку кереру являются одними из немногих представителей птиц в Новой Зеландии, способных глотать фрукты целиком, они играют заметную роль в распространении семян растений по территории страны. Любопытно, но за ними прочно закрепилось прозвище пьяный голубь – объевшись забродивших фруктов, они часто падают с веток деревьев.

В культурной жизни племени маори новозеландский плодоядный голубь всегда играл очень заметную роль. Так, по преданию, один из основных персонажей полинезийской мифологии и трикстер Мауи в поисках предков в загробном мире принял форму именно этой птицы. И хотя народ маори традиционно использовал в быту мясо, кости и перья кереру, в настоящее время охота на этих птиц строго запрещена.

По причине засилья браконьеров, а также ввиду загрязнения окружающей среды Международный союз по охране природы внес новозеландского голубя в список исчезающих животных. Многие представители фауны, помещенные на обложки книг издательства O'Reilly, находятся в опасности, им грозит исчезновение.

Цветная иллюстрация Карена Монтгомери (Karen Montgomery) основана на черно-белом оттиске из британского журнала *British Birds*. При оформлении обложки были использованы шрифты *Gilroy Semibold* и *Guardian Sans*. Шрифт текста в оригинале – Adobe Minion Pro, шрифт заголовков – Adobe Myriad Condensed, а шрифт листингов – Ubuntu Mono от компании Dalton Maag.

Часть I

ПРИСТУПАЕМ К РАБОТЕ

Прочитав первые четыре главы книги, вы сможете написать свое первое приложение с использованием фреймворка Shiny. Первая глава будет посвящена базовым принципам приложения Shiny и его структуре. Во второй и третьей главах вы погрузитесь в детальный разбор двух основных составляющих любого приложения Shiny: клиентского интерфейса, или *фронтенда* (того, что пользователь видит в браузере), и серверной части, или *бэкенда* (кода, заставляющего приложение работать). А завершим мы первую часть книги примером из практики, который поможет закрепить все полученные ранее знания.

Глава 1

Telegram канал: https://t.me/it_boooks

Ваше первое приложение Shiny

ВВЕДЕНИЕ

В этой главе вы создадите свое первое приложение Shiny. Начнем мы с минимальной шаблонной заготовки для полноценного приложения, после чего научимся запускать и останавливать его. Далее вы узнаете, что из себя представляют два ключевых компонента любого приложения Shiny, а именно *UI* (сокращенно от *user interface* – интерфейс пользователя), определяющий внешний вид приложения, и *функция server*, в которой реализовано поведение приложения. Фреймворк Shiny использует принципы *реактивного программирования* (reactive programming) для автоматического обновления элементов вывода при изменении значений элементов ввода, так что завершив эту главу мы знакомством с третьим важнейшим компонентом приложений Shiny – *реактивными выражениями* (reactive expressions).

Если вы еще не установили пакет Shiny, сейчас самое время сделать это, запустив следующую строку кода:

```
install.packages("shiny")
```

Если пакет Shiny у вас уже установлен, проверьте, что его версия – 1.5.0 или выше, при помощи функции `packageVersion("shiny")`.

Затем загрузите пакет Shiny в вашу текущую сессию:

```
library(shiny)
```

СОЗДАНИЕ ДИРЕКТОРИИ И ФАЙЛА ПРИЛОЖЕНИЯ

Создать *приложение Shiny* (Shiny app) можно несколькими способами. Простейший из них – создать новую директорию для приложения и сохранить в ней файл с именем *app.R*. В этом файле будет содержаться вся необходимая информация как по внешнему виду приложения, так и по его поведению.

Итак, создайте папку и сохраните в ней файл *app.R* со следующим содержанием:

```
library(shiny)
ui <- fluidPage(
  "Hello, world!"
)
server <- function(input, output, session) {
}
shinyApp(ui, server)
```

Не поверите, но это уже полноценное, хоть и весьма тривиальное, приложение Shiny! Этот простой скрипт делает ровно четыре вещи.

1. Загружает пакет Shiny при помощи инструкции `library(shiny)`.
2. Определяет пользовательский интерфейс – страницу HTML, с которой будет взаимодействовать пользователь. В данном случае это страница со словами «Hello, world!».
3. Формирует поведение приложения путем определения функции *server*. У нас эта функция пустая, так что наше приложение не будет делать ровным счетом ничего, но очень скоро мы это исправим.
4. Вызывает функцию `shinyApp(ui, server)` для сборки и запуска приложения Shiny с пользовательским интерфейсом и серверным поведением.

Примечание. В RStudio предусмотрено два удобных способа создания приложения Shiny:

- вы можете создать новую папку и файл *app.R* с шаблонным приложением, выбрав в меню **File** пункт **New Project**, затем в открывшемся диалоговом окне щелкнув на пункт **New Directory** и выбрав вариант **Shiny Web Application**;
- если у вас уже есть файл *app.R*, вы можете воспользоваться возможностью быстрой вставки заранее заготовленного фрагмента кода, называемого *снippetом* (snippet), – для этого введите текст `shinyapp` и нажмите сочетание клавиш **Shift+Tab**.

ЗАПУСК И ОСТАНОВКА

Запустить приложение также можно несколькими способами:

- нажать на кнопку **Run App** на панели инструментов, как показано на рис. 1.1;
- использовать комбинацию клавиш **Cmd/Ctrl+Shift+Enter**;
- если вы не используете RStudio, вы можете загрузить `(source())`¹ весь документ или вызвать функцию `shiny::runApp()` с указанием пути к папке, в которой располагается файл *app.R*.

¹ Дополнительные обрамляющие скобки здесь очень важны. Функция `shinyApp()` создает приложение только при печати, а скобки позволяют подгрузить последний результат из файла, который в противном случае вернулся бы невидимым.

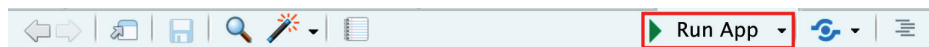


Рис. 1.1 ❖ Кнопка **Run App** располагается в правой части панели инструментов

Выберите один из перечисленных способов запуска приложения, и вы увидите его на экране в виде, показанном на рис. 1.2. Наши поздравления! Вот вы и написали свое первое приложение Shiny!

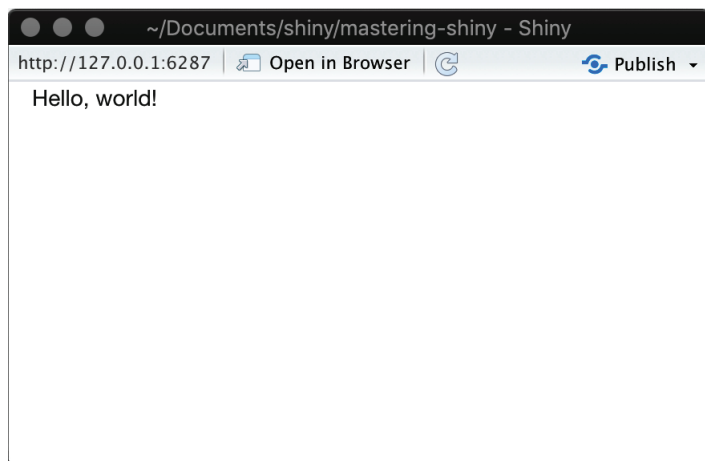


Рис. 1.2 ❖ Самое простое приложение Shiny

Перед тем как закрыть приложение, вернитесь в RStudio и взгляните на консоль. Вы обнаружите что-то типа этого:

```
#> Listening on http://127.0.0.1:3827
```

Здесь вы видите адрес, где находится ваше приложение, – 127.0.0.1, что означает «этот компьютер» и номер порта, присвоенный случайным образом, – в данном случае это 3827. Вы можете ввести этот адрес в любой совместимый¹ браузер, чтобы открыть еще одну копию приложения.

Заметьте, что при запущенном приложении оболочка R переходит в состояние занятости: командная строка не видна, а на панели инструментов в консоли показывается иконка с символом остановки. При запуске приложения Shiny происходит блокировка консоли R, так что до остановки приложения вы не сможете вводить в нее новые команды.

Остановить выполнение приложения и получить доступ к консоли можно одним из следующих способов:

¹ Shiny стремится поддерживать все современные браузеры (<https://www.rstudio.com/about/platform-support>). Обратите внимание, что Internet Explorer версии ниже IE11 не поддерживается для запуска Shiny напрямую из сессии R. При этом приложения Shiny, развернутые на сервере Shiny или на сайте *ShinyApps.io*, могут работать в IE10 (более ранние версии IE больше не поддерживаются).

- нажать на иконку с символом остановки на панели инструментов в консоли;
- перейти в консоль и нажать на клавишу **Esc** (или сочетание клавиш **Ctrl+C**, если вы не используете RStudio);
- закрыть окно приложения Shiny.

Традиционный для приложений Shiny процесс разработки состоит из написания кода, запуска приложения, его проверки, написания нового кода, повторного запуска и так по кругу. Если вы используете RStudio, вам не нужно будет каждый раз останавливать и запускать приложение, чтобы увидеть изменения. Вместо этого вы можете нажать на кнопку **Reload app** на панели инструментов или сочетание клавиш **Cmd/Ctrl+Shift+Enter**. Другие этапы разработки приложения мы обсудим в главе 5.

ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА

Теперь давайте добавим пару элементов управления в наше приложение, чтобы оно не выглядело таким пустым. К примеру, выведем информацию о всех доступных наборах данных из пакета *datasets*.

Замените код в переменной `ui` на приведенный ниже:

```
ui <- fluidPage(
  selectInput("dataset", label = "Dataset", choices = ls("package:datasets")),
  verbatimTextOutput("summary"),
  tableOutput("table")
)
```

Здесь мы видим следующие функции:

- *fluidPage()* – функция разметки (layout function), отвечающая за визуальную структуру приложения. Далее в этой книге мы еще поговорим о ней более подробно;
- *selectInput()* – элемент ввода, представляющий собой список, с помощью которого пользователь может сделать свой выбор. В данном случае мы присвоили списку метку *Dataset* и заполнили его наименованиями встроенных в R наборов данных. Подробнее об элементах ввода вы узнаете далее в этой главе;
- *verbatimTextOutput()* и *tableOutput()* – элементы вывода, предназначенные для отображения обработанной Shiny информации (как именно это происходит, мы покажем очень скоро). При этом элемент *verbatimTextOutput()* служит для вывода текста, а *tableOutput()* – для отображения табличных данных. Больше об элементах вывода вы узнаете в процессе чтения этой главы.

Функции разметки и элементы ввода и вывода применяются для совершенно разных целей, но все они, по сути, представляют собой лишь причудливые способы генерирования HTML-кода, и если вы вызовете любую

из этих функций за пределами приложения Shiny, то увидите код HTML на консоли – ничего более. Вскоре вы узнаете, как именно работают эти функции и элементы «под капотом».

Давайте запустим приложение снова. Теперь оно выглядит так, как показано на рис. 1.3, – в виде страницы с выпадающим списком. При этом мы видим только элемент ввода, но не видим определенные нами элементы вывода, поскольку еще не сообщили Shiny, как между собой должны быть связаны ввод и вывод.



Рис. 1.3 ❖ Приложение Shiny с интерфейсом пользователя

ДОБАВЛЕНИЕ ПОВЕДЕНИЯ

Теперь давайте оживим наши элементы вывода, определив их поведение в функции `server`.

Интерактивность приложений Shiny обеспечивается при помощи реактивного программирования. В главе 3 мы более подробно поговорим о реактивном программировании, а сейчас вам достаточно будет знать, что эта концепция позволяет Shiny определиться со способом выполнения расчетов, не запуская сами действия. Это как разница между тем, чтобы дать человеку рецепт и потребовать от него приготовить вам бутерброд.

Мы расскажем Shiny, как заполнять наши элементы вывода, предложив для этого свои рецепты или инструкции. Замените пустую функцию `server` на код, представленный ниже:

```
server <- function(input, output, session) {
  output$summary <- renderPrint({
    dataset <- get(input$dataset, "package:datasets")
    summary(dataset)
  })

  output$table <- renderTable({
    dataset <- get(input$dataset, "package:datasets")
    dataset
  })
}
```

В левой части от оператора присваивания (`<-`) располагаются идентификаторы наших выходных элементов вида `output$ID` – именно с их помощью мы сообщаем Shiny инструкцию для заполнения элементов с указанными идентификаторами. Справа от оператора находятся специальные *функции отображения* (render function) с заключенным в них кодом. Каждая функция с шаблонным именем `render{Тип}` призвана генерировать вывод определенного типа (например, текст, таблицу или график) и часто соотносится с шаблонными

функциями вида {тип}Output. В нашем случае функция `genderPrint()` работает совместно с `verbatimTextOutput()` для отображения статистической сводки в виде *текста с фиксированной шириной* (`verbatim`), а функция `genderTable()` действует в паре с функцией `tableOutput()` для вывода табличных данных.

Снова запустите приложение и посмотрите, что произойдет при изменении выбора в выпадающем списке. На рис. 1.4 показано, как будет выглядеть приложение после запуска.

Dataset

ability.cov ▼

	Length	Class	Mode
cov	36	-none-	numeric
center	6	-none-	numeric
n.obs	1	-none-	numeric

cov.general	cov.picture	cov.blocks	cov.maze	cov.reading	cov.vocab	center	n.obs
24.64	5.99	33.52	6.02	20.75	29.70	0.00	112.00
5.99	6.70	18.14	1.78	4.94	7.20	0.00	112.00
33.52	18.14	149.83	19.42	31.43	50.75	0.00	112.00
6.02	1.78	19.42	12.71	4.76	9.07	0.00	112.00
20.75	4.94	31.43	4.76	52.60	66.76	0.00	112.00
29.70	7.20	50.75	9.07	66.76	135.29	0.00	112.00

Рис. 1.4 ❖ Соединив воедино элементы ввода и вывода, мы получили полноценное интерактивное приложение

Обратите внимание, что выходной текст и таблица обновляются сразу после выбора набора данных из списка. Такая зависимость создается неявно, поскольку мы ссылаемся на элемент ввода `input$dataset` непосредственно в функциях вывода. В переменной `input$dataset` находится текущее значение элемента ввода с идентификатором `dataset`, и элементы вывода будут обновляться сразу после изменения этого значения. В этом состоит суть *реактивности* (*reactivity*): элементы вывода будут автоматически *реагировать* (*react*), то есть пересчитываться, при изменении значений соответствующих элементов ввода.

СНИЖЕНИЕ ДУБЛИРОВАНИЯ КОДА ПРИ ПОМОЩИ РЕАКТИВНЫХ ВЫРАЖЕНИЙ

Вы наверняка заметили, что даже в таком простом примере, как этот, мы вынуждены были продублировать часть кода – в частности, представленную ниже строку мы написали дважды:

```
dataset <- get(input$dataset, "package:datasets")
```

Но вы, будучи программистом, прекрасно знаете все минусы дублирования кода: от дополнительных временных затрат на его выполнение до сложности с поддержкой и отладкой. В нашем случае это не критично, но мне хотелось продемонстрировать эту проблему на предельно простом примере.

В традиционном программировании на языке R мы обычно используем две техники для снижения количества повторений в коде: значения сохраняем в переменные, а вычисления – в функции. К сожалению, применительно к Shiny ни одна из этих техник не позволит добиться желаемого результата, и причины этого будут подробно описаны при обсуждении концепции реактивного программирования. Таким образом, здесь нам просто не обойтись без *реактивных выражений* (reactive expression).

Реактивные выражения создаются путем заключения фрагмента кода в блок `reactive({...})` и присвоения его переменной. Запуск реактивного выражения осуществляется путем обращения к этой переменной как к функции. И хотя внешне это действительно выглядит как обычный вызов функции, в случае с реактивным выражением есть один важный нюанс – фактически оно вычисляется только при первом обращении, после чего результат вычисления кешируется до тех пор, пока не потребуются его обновить.

Ниже показано, как можно переписать нашу функцию `server()`, чтобы в ней использовались реактивные выражения. Внешне приложение будет вести себя так же, как и прежде, но его эффективность повысится за счет того, что извлекать набор данных мы будем один раз, а не два:

```
server <- function(input, output, session) {
  # Создаем реактивное выражение
  dataset <- reactive({
    get(input$dataset, "package:datasets")
  })

  output$summary <- renderPrint({
    # Используем реактивное выражение, обращаясь к нему как к функции
    summary(dataset())
  })

  output$table <- renderTable({
    dataset()
  })
}
```

Позже мы подробнее поговорим о реактивных выражениях, но даже полученного вами на данном этапе поверхностного знания об элементах ввода и вывода, а также о реактивных выражениях будет достаточно, чтобы писать полезные приложения Shiny!

ЗАКЛЮЧЕНИЕ

В данной главе вы создали простое приложение Shiny. Конечно, оно не потрясает воображение и не несет большой пользы, но вы увидели, как легко

можно сконструировать веб-приложение с использованием базовых знаний в R. В следующих двух главах вы еще больше узнаете об интерфейсе пользователя и реактивном программировании – двух основных строительных блоках Shiny. Теперь пришло время взглянуть на удобные шпаргалки, которые помогут вам лучше запомнить основы структуры приложений Shiny.

Shiny:: ШПАРГАЛКА

Основы

Приложение Shiny – это веб-страница (интерфейс), подключенная к компьютеру с запущенной сессией R



Пользователи могут взаимодействовать с интерфейсом, что будет приводить к изменению элементов посредством кода R

ШАБЛОН ПРИЛОЖЕНИЯ

Начинать новое приложение с этого шаблона. Предварительный просмотр приложения доступен при запуске кода в командной строке R

```
library(shiny)
ui <- fluidPage()
server <- function(input, output) {}
shinyApp(ui = ui, server = server)
```

- `ui` – вложенные функции R, образующие HTML-интерфейс приложения
- `server` – функция, управляющая объектами R, отображаемыми в интерфейсе
- `shinyApp` – объединяет `ui` и `server` в единое приложение. Оберните функцию в `runApp()`, если вызываете ее из загруженного скрипта или изнутри функции

ДЕЛИТЕСЬ СВОИМ ПРИЛОЖЕНИЕМ



Простейший способ поделиться приложением – выполнить его на shinyapps.io, облачный сервис от RStudio

1. Создайте бесплатный профессиональный аккаунт на сайте <http://shinyapps.io>
2. Щелкните по иконке Publish в RStudio или введите инструкцию: `rscconnect::deployApp()` (путь к папке)

Создайте или купите собственный сервер Shiny по адресу www.rstudio.com/products/shiny-server/

Сборка приложения

Завершите шаблон, добавив аргументы в функцию `fluidPage()` и тело – в функцию `server()`

Добавляем элементы ввода в интерфейс с помощью `Input`-функций
Добавляем элементы вывода с помощью `Output`-функций
Говорим серверу, как отображать элементы вывода. Для этого:

1. Обращаемся к элементам вывода как к `output$<id>`
2. Обращаемся к элементам ввода как к `input$<id>`
3. Оборачиваем код в `render`-функцию перед сохранением в элемент вывода

Сохраните шаблон в файле `app.R`. Также можно разбить приложение на файлы `ui.R` и `server.R`

```
library(shiny)
ui <- fluidPage(
  numericInput("age", "Sample size", value = 25),
  plotOutput("hist")
)

server <- function(input, output) {
  output$hist <- renderPlot({
    hist(norm(rnorm(input$age)))
  })
}

shinyApp(ui = ui, server = server)
```

Сохраняйте приложения в виде папок с файлом `app.R` (или файлами `server.R` и `ui.R`) и дополнительными файлами

- `app-name` – имя папки – это имя приложения
- `app.R` (необязательно) определяет объекты, доступные в `ui.R` и `server.R`
- `DESCRIPTION` (необязательно) используются в режиме презентации
- `<other files>` (необязательно) данные, скрипты и т.д.
- (необязательно) папка с файлами для веб-браузеров (изображения, CSS, js и т.д.). Должна иметь имя `www`

Запуск приложений с помощью `runApp()` (путь к папке)

Элементы Вывода – `render`- и `output`-функции работают совместно для вывода информации в интерфейс

DT `renderDataTable(expr, options, callback, escape, env, quoted)` → `dataTableOutput(outputId, icon, ...)`

renderImage `renderImage(expr, env, quoted, deleteFile)` → `imageOutput(outputId, width, height, click, dblclick, hover, hoverDelay, inline, hoverDelayType, brush, clickId, hoverId)`

renderPlot `renderPlot(expr, width, height, res, ..., env, quoted, func)` → `plotOutput(outputId, width, height, click, dblclick, hover, hoverDelay, inline, hoverDelayType, brush, clickId, hoverId)`

renderPrint `renderPrint(expr, env, quoted, func, width)` → `verbatimTextOutput(outputId)`

renderTable `renderTable(expr, ..., env, quoted, func)` → `tableOutput(outputId)`

renderText `renderText(expr, env, quoted, func)` → `textOutput(outputId, container, inline)`

renderUI `renderUI(expr, env, quoted, func)` → `uiOutput(outputId, inline, container, ...)` & `htmlOutput(outputId, inline, container, ...)`

Элементы ввода

Запрашивают ввод пользователя
Получите доступ к текущему значению элемента ввода при помощи `input$<inputId>`. Значения элементов ввода являются **реактивными**.

Action
`actionButton(inputId, label, icon, ...)`

Link
`actionLink(inputId, label, icon, ...)`

Choice 1
`checkboxGroupInput(inputId, label, choices, selected, inline)`

Choice 2
`checkboxInput(inputId, label, value)`

Choice 3
`dateInput(inputId, label, value, min, max, format, startview, weekstart, language)`

Choice 4
`dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator)`

Choice 5
`fileInput(inputId, label, multiple, accept)`

Choice 6
`numericInput(inputId, label, value, min, max, step)`

Choice 7
`passwordInput(inputId, label, value)`

Choice 8
`radioButton(inputId, label, choices, selected, inline)`

Choice 9
`selectInput(inputId, label, choices, selected, width, size) (также selectizeInput())`

Choice 10
`sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)`

Choice 11
`submitButton(text, icon) (предотвращает действия в приложении)`

Choice 12
`textInput(inputId, label, value)`



RStudio® is a trademark of RStudio, Inc. • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at shiny.rstudio.com • shiny 0.12.0 • Updated 2016-01

УПРАЖНЕНИЯ

Упражнение 1

Создайте приложение, приветствующее пользователя по имени. Вы пока еще не знаете всех функций, которые понадобятся для выполнения этого задания, так что мы включили несколько строк кода в случайном порядке. Подумайте, какие из них вам понадобятся для приложения, и вставьте их в свой код в правильном порядке:

```
tableOutput("mortgage")
output$greeting <- renderText({
```

```

  paste0("Hello ", input$name)
})
numericInput("age", "How old are you?", value = NA)
textInput("name", "What's your name?")
textOutput("greeting")
output$histogram <- renderPlot({
  hist(rnorm(1000))
}, res = 96)

```

Упражнение 2

Представьте, что ваш коллега пишет приложение, в котором пользователь может ввести значение (x) между 1 и 50 и посмотреть, чему будет равно произведение введенного значения и цифры 5. Вот его первая попытка:

```

library(shiny)

ui <- fluidPage(
  sliderInput("x", label = "Если x равно", min = 1, max = 50, value = 30),
  "то x умножить на 5 будет",
  textOutput("product")
)

server <- function(input, output, session) {
  output$product <- renderText({
    x * 5
  })
}

shinyApp(ui, server)

```

К сожалению, при запуске приложение выдало ошибку, показанную на рис. 1.5.

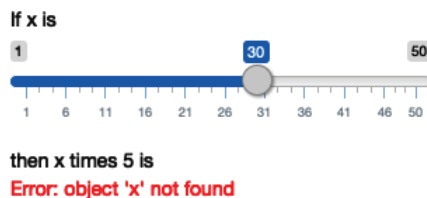


Рис. 1.5 ❖ Объект x не найден

Помогите коллеге решить проблему!

Упражнение 3

Расширьте приложение из предыдущего упражнения таким образом, чтобы пользователь мог вводить оба множителя: x и y . Итоговый результат должен выглядеть так, как показано на рис. 1.6.

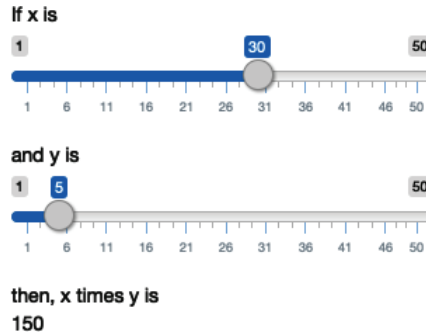


Рис. 1.6 ❖ Приложение, рассчитывающее произведение введенных чисел

Упражнение 4

Взгляните на код приложения, ставшего развитием предыдущего примера. Что здесь добавилось? Как можно снизить количество дублирующегося кода за счет применения реактивных выражений?

```
library(shiny)

ui <- fluidPage(
  sliderInput("x", "Если x равно", min = 1, max = 50, value = 30),
  sliderInput("y", "и y равно", min = 1, max = 50, value = 5),
  "то (x * y) будет", textOutput("product"),
  "(x * y) + 5 будет", textOutput("product_plus5"),
  "а (x * y) + 10 будет", textOutput("product_plus10")
)

server <- function(input, output, session) {
  output$product <- renderText({
    product <- input$x * input$y
    product
  })
  output$product_plus5 <- renderText({
    product <- input$x * input$y
    product + 5
  })
  output$product_plus10 <- renderText({
    product <- input$x * input$y
    product + 10
  })
}

shinyApp(ui, server)
```

Упражнение 5

Приведенное ниже приложение очень похоже на то, что вы видели ранее в этой главе: вы выбираете набор данных из пакета (на этот раз мы использовали пакет `ggplot2`) и на выходе получаете сводку по нему и график. Заметьте, что здесь мы использовали реактивные выражения для уменьшения дублирования кода. Но в коде есть три ошибки. Вы сможете найти и исправить их?

```
library(shiny)
library(ggplot2)

datasets <- c("economics", "faithful", "seals")

ui <- fluidPage(
  selectInput("dataset", "Dataset", choices = datasets),
  verbatimTextOutput("summary"),
  tableOutput("plot")
)

server <- function(input, output, session) {
  dataset <- reactive({
    get(input$dataset, "package:ggplot2")
  })

  output$summary <- renderPrint({
    summary(dataset())
  })

  output$plot <- renderPlot({
    plot(dataset)
  }, res = 96)
}

shinyApp(ui, server)
```

Глава 2

Основы интерфейса пользователя

ВВЕДЕНИЕ

Теперь, когда вы знаете базовую структуру приложений Shiny, пришло время начать знакомиться с элементами, составляющими ее основу. Вы, наверное, обратили внимание, что в Shiny есть строгое разграничение между кодом, который генерирует интерфейс пользователя (фронтенд), и кодом, определяющим поведение приложения (бэкенд).

В данной главе мы сосредоточимся на *пользовательском интерфейсе* (user interface – UI) и пройдемся по элементам ввода и вывода HTML, представленным в Shiny. С их помощью вы сможете получать от пользователя входные данные разного типа и выводить на экран информацию в разных форматах. Пока вы еще не освоили все многообразие способов объединения элементов ввода и вывода в приложении – об этом мы будем подробно говорить в главе 6.

Здесь мы обсудим элементы, встроенные во фреймворк Shiny. В то же время в сообществе постоянно появляются все новые расширения элементов интерфейса Shiny, такие как *shinyWidgets* (<https://github.com/dreamRs/shinyWidgets>), *colourpicker* (<https://github.com/daattali/colourpicker>) и *sortable* (<https://rstudio.github.io/sortable>). Нан Сяо (Nan Xiao) также поддерживает актуальный и всеобъемлющий список пакетов Shiny (<https://github.com/nanxstats/awesome-shiny-extensions>).

Как обычно, начнем с загрузки Shiny:

```
library(shiny)
```

ЭЛЕМЕНТЫ ВВОДА

В предыдущей главе вы уже видели, как можно использовать функции `sliderInput()`, `selectInput()`, `textInput()` и `numericInput()` для внедрения элементов управления в ваше приложение. Сейчас мы обсудим базовую структуру, ле-

жащую в основе всех функций *элементов ввода* (Input), а также пройдемся по элементам, встроенным в Shiny.

Базовая структура

Все функции ввода принимают одинаковый первый аргумент – *inputId*. Этот идентификатор используется для соединения фронтенда с бэкендом, то есть интерфейсной части приложения с серверной. Если в вашем приложении есть элемент ввода с идентификатором *name*, на стороне сервера вы сможете обращаться к нему по имени *input\$name*.

У аргумента *inputId* есть два следующих ограничения:

- он должен быть текстовым и состоять из букв, цифр и символов подчеркивания. Пробелы, слеш, точки и другие специальные символы недопустимы. Именуите его так, как именовали бы соответствующую переменную в R;
- он должен быть уникальным. В противном случае вы не сможете обратиться к создаваемому элементу управления в серверной функции.

Большинство функций ввода имеют также второй аргумент с именем *label*. Он используется для создания текстовой метки рядом с элементом управления. Shiny не устанавливает никаких ограничений на содержимое этого аргумента, но вы должны тщательно подходить к выбору его значения, поскольку его будет видеть пользователь вашего приложения. Третьим аргументом, как правило, идет *value*, позволяющий – где это возможно – установить для элемента значение по умолчанию. Остальные аргументы уникальны для каждого элемента ввода.

При создании элементов управления я советую передавать аргументы *inputId* и *label* по позиции, а все остальные – по имени:

```
sliderInput("min", "Limit (minimum)", value = 50, min = 0, max = 100)
```

В следующих разделах мы пройдем по элементам ввода, встроенным в Shiny, с группировкой по типу входных значений. Цель – быстро познакомить вас с имеющимся в вашем арсенале оружием, а не приводить бесконечный список всех возможных аргументов. Мы ограничимся лишь наиболее значимыми параметрами для каждого элемента, а с их полным перечнем вы сможете ознакомиться при чтении документации.

Текст

Небольшие фрагменты текста удобно обрабатывать при помощи функции *textInput()*, для ввода паролей предусмотрен элемент *passwordInput()*¹, а если

¹ Все, что делает функция *passwordInput()*, – так это скрывает содержимое ввода, чтобы никто не мог подсмотреть пароль через плечо. В вашу ответственность входит гарантия того, чтобы пароли не попали в чужие руки, так что без наличия опыта программирования систем безопасности братья за приложения, в которых

вы хотите, чтобы пользователь ввел один или несколько абзацев, используйте функцию `textAreaInput()`:

```
ui <- fluidPage(
  textInput("name", "What's your name?"),
  passwordInput("password", "What's your password?"),
  textAreaInput("story", "Tell me about yourself", rows = 3)
)
```

Внешний вид этих элементов ввода показан на рис. 2.1.

The image shows a web interface with three input elements. The first is a text input field with the label "What's your name?". The second is a password input field with the label "What's your password?". The third is a text area with the label "Tell me about yourself" and a height of 3 rows.

Рис. 2.1 ❖ Текстовые поля для ввода

Если вам необходимо отследить ввод пользователя, вы можете использовать функцию `validate()`, о которой мы подробнее расскажем в главе 8.

Числовой ввод

Для ввода числовых значений в Shiny можно использовать текстовое поле с ограничениями, представленное функцией `numericInput()`, или ползунок (slider), за отображение которого отвечает функция `sliderInput()`. При этом если в качестве аргумента `value` функции `sliderInput()` передать вектор, состоящий из двух числовых значений, вы получите ползунок с двумя маркерами для указания диапазона, как показано на рис. 2.2:

```
ui <- fluidPage(
  numericInput("num", "Number one", value = 0, min = 0, max = 100),
  sliderInput("num2", "Number two", value = 50, min = 0, max = 100),
  sliderInput("rng", "Range", value = c(10, 20), min = 0, max = 100)
)
```

Я бы рекомендовал использовать ползунок только для небольших диапазонов, в которых точные границы не столь важны. Дело в том, что добиться

пользователи вводят пароли, не рекомендуется.

абсолютной точности выбора значений при помощи этого элемента ввода бывает очень непросто.

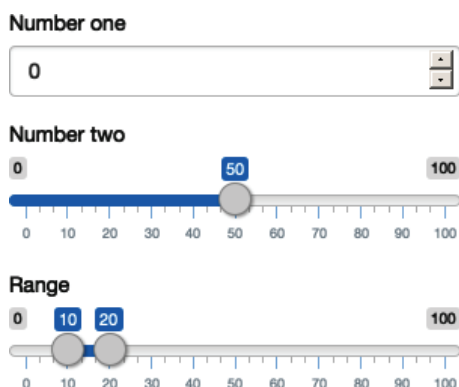


Рис. 2.2 ❖ Элементы ввода числовых значений

При этом ползунок – довольно хорошо настраиваемый элемент, и его внешний вид можно легко подогнать под свои требования. Подробно читайте в разделе «Using sliders» по адресу <https://shiny.rstudio.com/articles/sliders.html>.

Даты

Ввод единственной даты можно обрабатывать при помощи функции *dateInput()*, а диапазона из двух дат – посредством *dateRangeInput()*. Эти функции позволяют пользователю выбрать нужные даты с использованием удобного визуального календаря, а дополнительные аргументы, такие как *datesdisabled* и *daysofweekdisabled*, помогут ограничить ввод диапазоном допустимых значений:

```
ui <- fluidPage(
  dateInput("dob", "When were you born?"),
  dateRangeInput("holiday", "When do you want to go on vacation next?")
)
```

Внешний вид этих элементов ввода показан на рис. 2.3.



Рис. 2.3 ❖ Элементы ввода дат

Формат даты, язык и первый день недели по умолчанию настроены по американскому стандарту. Если вы пишете приложение для международной аудитории, настройте поля при помощи аргументов `format`, `language` и `weekstart`, чтобы пользователям было привычнее работать с датами.

Ограниченный выбор

Если вы хотите, чтобы пользователь выбирал значение из ограниченного числа вариантов, то можете воспользоваться функциями `selectInput()` и `radioButtons()`, как показано ниже:

```
animals <- c("dog", "cat", "mouse", "bird", "other", "I hate animals")

ui <- fluidPage(
  selectInput("state", "What's your favourite state?", state.name),
  radioButtons("animal", "What's your favourite animal?", animals)
)
```

Внешний вид элементов выбора значений показан на рис. 2.4.



What's your favourite state?

Alabama ▼

What's your favourite animal?

☒ dog

☐ cat

☐ mouse

☐ bird

☐ other

☐ I hate animals

Рис. 2.4 ❖ Элементы выбора значений

При использовании *радиокнопок* (radio button) на экран выводятся все возможные варианты выбора, что делает этот элемент ввода пригодным только в случае ограниченного выбора с небольшим количеством вариантов. Кроме того, настройка аргументов `choiceNames` и `choiceValues` позволит выводить в качестве вариантов не только текст. Аргумент `choiceNames` отвечает за внешнее отображение имен переключателей, а `choiceValues` – за возвращаемые значения в серверной функции:

```
ui <- fluidPage(
  radioButtons("rb", "Choose one:",
    choiceNames = list(
      icon("angry"),
      icon("smile"),
      icon("sad-tear")
    )
  )
)
```

```

    ),
    choiceValues = list("angry", "happy", "sad")
  )
)

```

Внешний вид переключателя с иконками показан на рис. 2.5.

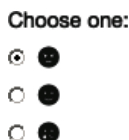


Рис. 2.5 ❖ Переключатель с иконками

Выпадающие списки, созданные при помощи функции `selectInput()`, занимают одинаковое место вне зависимости от количества элементов выбора, что делает их подходящими в случае большого числа вариантов. Кроме того, вы можете передать в качестве аргумента `multiple` значение `TRUE`, что позволит пользователю сделать множественный выбор, как показано на рис. 2.6:

```

ui <- fluidPage(
  selectInput(
    "state", "What's your favourite state?", state.name, multiple = TRUE
  )
)

```

What's your favourite state?

Рис. 2.6 ❖ Множественный выбор из списка

Если у вас слишком уж много вариантов для выбора, вы можете использовать серверную версию элемента ввода `selectInput()`. В этом случае все возможные выборы не встраиваются в клиентский интерфейс, что могло бы привести к задержкам загрузки, а посылаются сервером при необходимости. Подробнее об этом можно почитать в разделе «Server-side selectize» по адресу <https://shiny.rstudio.com/articles/selectize.html#server-side-selectize>.

При использовании радиокнопок пользователь не сможет выбрать несколько значений одновременно – для этого придется воспользоваться флажками и соответствующей им функцией `checkboxGroupInput()`, как показано ниже:

```

ui <- fluidPage(
  checkboxGroupInput("animal", "What animals do you like?", animals)
)

```

Внешний вид флажков показан на рис. 2.7.

What animals do you like?

☐ dog

☐ cat

☐ mouse

☐ bird

☐ other

☐ I hate animals

Рис. 2.7 ❖ Флажки с выбором

Если вам нужен отдельный флажок для ответа на вопрос типа да/нет, используйте функцию *checkboxInput()*, как показано ниже:

```
ui <- fluidPage(
  checkboxInput("cleanup", "Clean up?", value = TRUE),
  checkboxInput("shutdown", "Shutdown?")
)
```

Одиночные флажки представлены на рис. 2.8.

☒ Clean up?

☐ Shutdown?

Рис. 2.8 ❖ Одиночные флажки с выбором

Загрузка файлов

Позволить пользователю загружать файлы со своего компьютера можно при помощи функции *fileInput()*, как показано ниже:

```
ui <- fluidPage(
  fileInput("upload", NULL)
)
```

Элемент загрузки файла отображен на рис. 2.9.

Browse...

No file selected

Рис. 2.9 ❖ Элемент загрузки файла

Функция *fileInput()* требует особой обработки на стороне сервера, о чем мы подробно поговорим в главе 9.

Кнопки

Для подтверждения действия пользователю можно дать в распоряжение кнопку или ссылку с помощью функций `actionButton()` и `actionLink()` соответственно, как показано ниже:

```
ui <- fluidPage(
  actionButton("click", "Click me!"),
  actionButton("drink", "Drink me!", icon = icon("cocktail"))
)
```

Внешний вид созданных кнопок показан на рис. 2.10.



Рис. 2.10 ❖ Кнопки

Обычно кнопки и ссылки работают в паре с функциями `observeEvent()` или `tReactive()`, располагающимися в серверной части приложения. Мы пока не встречались с этими функциями в книге, но очень скоро закроем данный пробел.

Вы можете настроить внешний вид кнопок по своему желанию, передав в качестве аргумента `class` одно из следующих значений: `"btn-primary"`, `"btn-success"`, `"btn-info"`, `"btn-warning"` или `"btn-danger"`. Вы также можете изменить размер кнопки при помощи значений `"btn-lg"`, `"btn-sm"` или `"btn-xs"`. Наконец, вы можете заставить кнопку занять всю свободную ширину внутри элемента, в который она встроена, используя значение `"btn-block"`:

```
ui <- fluidPage(
  fluidRow(
    actionButton("click", "Click me!", class = "btn-danger"),
    actionButton("drink", "Drink me!", class = "btn-lg btn-success")
  ),
  fluidRow(
    actionButton("eat", "Eat me!", class = "btn-block")
  )
)
```

Внешний вид измененных кнопок показан на рис. 2.11.



Рис. 2.11 ❖ Модифицированные кнопки

Аргумент `class` устанавливает соответствующий атрибут в коде HTML, влияющий на стиль отображения кнопок. За информацией о дополнитель-

ных опциях можно обратиться к документации по *Bootstrap* – фреймворку CSS, используемому в Shiny, по адресу <http://bootstrapdocs.com/v3.3.6/docs/css/#buttons>.

Упражнения

Упражнение 1

Когда свободного пространства в приложении не так много, бывает полезно размещать метку текстового поля непосредственно внутри него. Каким должен быть вызов функции `textInput()` для получения результата, показанного на рис. 2.12?



Рис. 2.12 ❖ Текстовое поле со встроенной меткой

Упражнение 2

Внимательно прочитайте документацию к функции `sliderInput()` и попробуйте написать такой ее вызов, чтобы пользователь увидел показанный на рис. 2.13 ползунок с датами.



Рис. 2.13 ❖ Ползунок выбора даты

Упражнение 3

Создайте ползунок для выбора значения в интервале от 0 до 100 с шагом 5. После этого добавьте такую анимацию виджету, чтобы при нажатии на кнопку запуска ползунок перемещался по диапазону автоматически.

Упражнение 4

Если в вашем списке выборов для функции `selectInput()` чересчур много вариантов, бывает полезно создать внутри списка подзаголовки для его лучшего восприятия. Ознакомьтесь с документацией к функции и постарайтесь реализовать этот функционал на практике (подсказка: соответствующий тег в HTML называется `<optgroup>`).

ЭЛЕМЕНТЫ ВЫВОДА

Элементы вывода (output) представляют собой своеобразные заглушки в интерфейсе пользователя, которые при необходимости заполняются с по-

мощью функции `server`, как и элементы ввода, элементы вывода принимают идентификатор в качестве обязательного первого аргумента¹. Если в вашем пользовательском интерфейсе есть элемент вывода с идентификатором "plot", в серверной части приложения обратиться к нему можно будет по имени `output$plot`.

Каждая функция вывода в клиентской части сопоставляется с функцией отображения в серверной. Существует три основных типа вывода, соответствующих трем базовым видам представления данных в отчетах, а именно текст, таблицы и графики. В следующих разделах мы подробно остановимся на функциях вывода и отображения.

Текст

Для вывода обычного текста в приложении Shiny служит функция `textOutput()`, а для моноширинного (например, как в консоли) – функция `verbatimTextOutput()`:

```
ui <- fluidPage(
  textOutput("text"),
  verbatimTextOutput("code")
)

server <- function(input, output, session) {
  output$text <- renderText({
    "Hello friend!"
  })
  output$code <- renderPrint({
    summary(1:10)
  })
}
```

Вывод приложения показан на рис. 2.14.

Hello friend!

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1.00	3.25	5.50	5.50	7.75	10.00

Рис. 2.14 ❖ Вывод текстового содержания в приложении

Заметьте, что фигурные скобки `{ }` в функциях отображения являются обязательными только в случае, если блок внутри них состоит из нескольких строк. Как вы увидите позже, зачастую вам не придется писать много кода внутри таких функций, а значит, и фигурные скобки нужны будут не так

¹ Обратите внимание, что имя первого аргумента для элементов ввода и вывода отличается – `inputId` и `outputId` соответственно. Я не использую имя этого аргумента – он настолько важен, что забыть его предназначение не представляется возможным.

часто. Вот как могла бы выглядеть серверная часть приложения без этих дополнительных скобок. Согласитесь, получилось довольно компактно:

```
server <- function(input, output, session) {
  output$text <- renderText("Hello friend!")
  output$code <- renderPrint(summary(1:10))
}
```

Обратите внимание, что мы использовали две разные функции отображения, которые и ведут себя чуть по-разному:

- *renderText()* – собирает результат в строку и обычно применяется в паре с функцией *textOutput()*;
- *renderPrint()* – выводит результат на печать, как если бы мы осуществляли вывод в консоль R, и обычно употребляется совместно с функцией *verbatimTextOutput()*.

Различия между выводами хорошо видны в следующем приложении:

```
ui <- fluidPage(
  textOutput("text"),
  verbatimTextOutput("print")
)

server <- function(input, output, session) {
  output$text <- renderText("hello!")
  output$print <- renderPrint("hello!")
}
```

Вывод приложения показан на рис. 2.15.

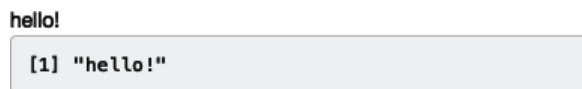


Рис. 2.15 ❖ Разница в выводе
между функциями *renderText()* и *renderPrint()*

Различия между показанными функциями отображения такие же, как между функциями *cat()* и *print()* в R.

Таблицы

В Shiny есть два способа отображения *датафреймов* (data frame) в виде таблиц:

- *tableOutput()* и *renderTable()* – используются для вывода статических таблиц с показом на экране всех данных;
- *dataTableOutput()* и *renderDataTable()* – применяется для отображения динамических таблиц с выводом фиксированного количества строк и возможностью переключаться между страницами.

Функция `tableOutput()` обычно используется для вывода на экран небольших таблиц со сводными данными (например, с коэффициентами модели), тогда как `dataTableOutput()` применяется для вывода полноценного датафрейма, с которым пользователь сможет работать, используя богатый встроенный функционал. Следующий простой пример наглядно демонстрирует разницу между функциями `tableOutput()` и `dataTableOutput()`:

```
ui <- fluidPage(
  tableOutput("static"),
  dataTableOutput("dynamic")
)

server <- function(input, output, session) {
  output$static <- renderTable(head(mtcars))
  output$dynamic <- renderDataTable(mtcars, options = list(pageLength = 5))
}
```

Вывод этого приложения показан на рис. 2.16.

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
21.00	6.00	160.00	110.00	3.90	2.62	16.46	0.00	1.00	4.00	4.00
21.00	6.00	160.00	110.00	3.90	2.88	17.02	0.00	1.00	4.00	4.00
22.80	4.00	108.00	93.00	3.85	2.32	18.61	1.00	1.00	4.00	1.00
21.40	6.00	258.00	110.00	3.08	3.21	19.44	1.00	0.00	3.00	1.00
18.70	8.00	360.00	175.00	3.15	3.44	17.02	0.00	0.00	3.00	2.00
18.10	6.00	225.00	105.00	2.76	3.46	20.22	1.00	0.00	3.00	1.00

Show entries

Search:

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
21	6	160	110	3.9	2.62	16.46	0	1	4	4
21	6	160	110	3.9	2.875	17.02	0	1	4	4
22.8	4	108	93	3.85	2.32	18.61	1	1	4	1
21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
18.7	8	360	175	3.15	3.44	17.02	0	0	3	2

Showing 1 to 5 of 32 entries

Previous
1
2
3
4
5
6
7
Next

Рис. 2.16 ❖ Разница в выводе между функциями `renderTable()` и `renderDataTable()`

Если вам нужен еще больший контроль над выводом функции `dataTableOutput()`, я очень рекомендую вам использовать пакет *reactable* от Грегга Лина (Greg Lin): <https://glin.github.io/reactable>.

Графики

Вы можете выводить на экран различные диаграммы (например, из базового пакета `ggplot2`) при помощи функций `plotOutput()` и `renderPlot()`, как показано ниже:

```
ui <- fluidPage(
  plotOutput("plot", width = "400px")
)

server <- function(input, output, session) {
  output$plot <- renderPlot(plot(1:5), res = 96)
}
```

Вывод приложения показан на рис. 2.17.

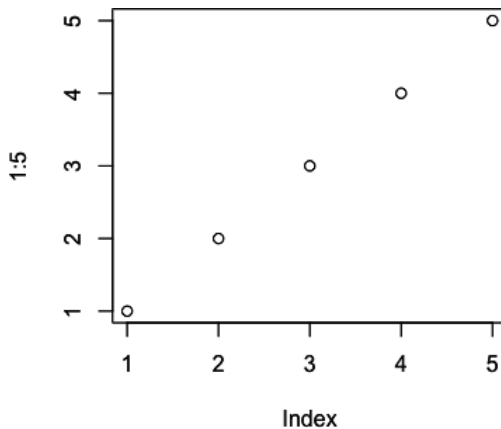


Рис. 2.17 ❖ Вывод диаграммы при помощи функции `renderPlot()`

По умолчанию вывод функции `plotOutput()` будет занимать все доступное пространство по ширине в рамках своего контейнера (подробнее об этом мы поговорим далее), а его высота составит 400 пикселей. Переопределить эти параметры по умолчанию можно, передав в функцию аргументы `height` и `width`. Аргументу `res` мы советуем всегда присваивать значение 96 – так ваши диаграммы в Shiny будут максимально похожи на то, что вы видите в RStudio.

Графики представляют собой особый вид вывода по причине того, что они одновременно могут выступать и как элементы ввода. У функции `plotOutput()` есть множество аргументов, таких как `click`, `dblclick` и `hover`. Если передать одному из этих аргументов строку – например, `click = "plot_click"`, – будет создан реактивный элемент ввода (`input$plot_click`), который вы сможете использовать для отслеживания взаимодействия пользователя с графиком (к примеру, щелчки мышью). Мы будем подробно говорить об интерактивных графиках Shiny в главе 7.

Загрузка файлов

Вы можете позволить пользователю скачивать файлы, используя функции `downloadButton()` и `downloadLink()`. Это требует определенной обработки на серверной стороне приложения, так что мы вернемся к данной теме и обсудим ее более подробно в главе 9.

Упражнения

Упражнение 1

В паре с какой функцией (`textOutput()` или `verbatimTextOutput()`) могут использоваться перечисленные ниже функции отображения?

- a) `renderPrint(summary(mtcars));`
- b) `renderText("Good morning!");`
- c) `renderPrint(t.test(1:5, 2:6));`
- d) `renderText(str(lm(mpg ~ wt, data = mtcars))).`

Упражнение 2

Измените приложение из раздела «Графики» таким образом, чтобы высота диаграммы составляла 300 пикселей, а ширина – 700 пикселей. Также снабдите график альтернативным текстом, чтобы слабовидящие пользователи могли понять, что перед ними точечная диаграмма по пяти случайным числам.

Упражнение 3

Обновите вызов функции `renderDataTable()` таким образом, чтобы данные отображались, но все элементы управления таблицей были убраны (в частности, не должны отображаться поле поиска и инструменты сортировки и фильтрации). Для выполнения этого задания вам придется почитать вывод команды `?renderDataTable` и пройти по опциям соответствующей библиотеки JavaScript по адресу <https://datatables.net/reference/option>:

```
ui <- fluidPage(  
  dataTableOutput("table")  
)  
  
server <- function(input, output, session) {  
  output$table <- renderDataTable(mtcars, options = list(pageLength = 5))  
}
```

Упражнение 4

Прочитайте о пакете *reactable* по адресу <https://glin.github.io/reactable> и используйте его в предыдущем приложении.

ЗАКЛЮЧЕНИЕ

В данной главе вы познакомились с функциями ввода и вывода, составляющими основу интерфейса любого приложения Shiny. Информации в этой главе было довольно много, так что не пытайтесь запомнить все, что прочитали. Лучше вернитесь к ней при выборе того или иного элемента – вы очень быстро все вспомните и найдете нужный вам фрагмент кода.

В следующей главе мы перейдем к обсуждению серверной части – кода, приводящего в чувство наши приложения Shiny.

Глава 3

Основы реактивного программирования

ВВЕДЕНИЕ

В Shiny серверная логика зачастую реализуется при помощи реактивного программирования. *Реактивное программирование* (reactive programming) представляет собой очень элегантную и мощную парадигму, которая поначалу способна вводить в ступор по причине своих кардинальных отличий от традиционных методов написания программ. Ключевая идея реактивного программирования состоит в определении схемы зависимостей между элементами приложения – чтобы при изменении состояния элементов ввода автоматически менялись и соответствующие им элементы вывода. Это значительно упрощает ход выполнения программы, но вам может потребоваться некоторое время, чтобы осознать, как именно функционирует эта новая для вас концепция.

В данной главе мы поговорим об основах реактивного программирования и рассмотрим основные реактивные конструкции, которые вы будете применять в приложениях Shiny. Начнем мы с описания важной функции `server` и того, как именно используются ее аргументы `input` и `output`. Далее рассмотрим простейший пример реактивной концепции (когда элементы ввода и вывода связаны между собой напрямую), а позже подробно поговорим о реактивных выражениях, способных сократить количество дублирующегося кода в приложениях. В заключение обсудим некоторые распространенные затруднения, с которыми традиционно сталкиваются новички в Shiny.

ФУНКЦИЯ SERVER

Как вы уже видели, структура любого приложения Shiny выглядит примерно так:

```
library(shiny)
```

```
ui <- fluidPage(
```

```
# клиентский интерфейс
)

server <- function(input, output, session) {
  # серверная логика
}

shinyApp(ui, server)
```

В предыдущей главе мы затронули вопросы основы клиентской части приложений, представленной переменной `ui`, в которой содержится весь код HTML, демонстрируемый в браузере пользователю. Переменная `ui` до крайности проста – в конце концов все пользователи видят на экране один и тот же HTML. Что касается переменной `server`, она куда более сложная, поскольку каждый пользователь должен получить свою версию приложения – когда один пользователь перемещает ползунок, второй не должен видеть результат этого действия.

Для достижения такой независимости запусков Shiny вызывает функцию `server()` каждый раз, когда стартует сессия¹. Как и в случае с любыми другими функциями в R, при вызове функции `server()` создается новое локальное окружение, независимое от других вызовов функции. Это позволяет каждой сессии хранить свое уникальное состояние и изолировать переменные, созданные внутри функции. Именно поэтому почти все реактивное программирование в Shiny будет происходить в функции `server()`².

Функция `server` принимает на вход три аргумента: `input`, `output` и `session`. Поскольку вы никогда не будете вызывать эту функцию вручную, создавать эти объекты вы также не будете. Вместо этого они создаются автоматически при запуске сессии с привязкой к ней. Сейчас мы подробнее поговорим об аргументах `input` и `output`, а объект `session` оставим на следующие главы книги.

Input

Объект `input` представляет собой список, включающий все данные, посылаемые из браузера, именованные в соответствии с идентификаторами элементов. Например, если в вашем интерфейсе пользователя присутствует числовое поле для ввода с идентификатором `count`, как показано в примере ниже:

```
ui <- fluidPage(
  numericInput("count", label = "Number of values", value = 100)
)
```

¹ Каждое подключение к приложению Shiny открывает новую сессию вне зависимости от того, выполняют ли его разные пользователи или один с разных вкладок браузера.

² Исключение составляют ситуации, когда выполняется некая разделяемая между пользователями работа. Например, несколько пользователей могут просматривать один и тот же большой файл CSV, так что для экономии ресурсов можно загрузить его один раз и обеспечить к нему общий доступ. Мы будем обсуждать эту концепцию подробно далее в данной книге.

то обратиться к значению этого элемента можно будет при помощи выражения `input$count`. Изначально в этой переменной будет содержаться число *100*, но это значение будет меняться динамически при изменении пользователем содержимого поля в браузере.

В отличие от обычных объектов списочного типа, объект `input` доступен только для чтения. Если попытаться переопределить значение переменной `input` внутри функции `server`, возникнет ошибка, как показано ниже:

```
server <- function(input, output, session) {
  input$count <- 10
}

shinyApp(ui, server)
#> Error: Can't modify read-only reactive value 'count'
```

Причина ошибки состоит в том, что объект `input` отражает происходящее в браузере, а браузер для Shiny – это «единственный источник истины». Если бы вы могли менять значение этого объекта в R, возникла бы неразбериха – физическое значение в поле браузера было бы одним, а значение `input$count` – другим. Позже, в главе 8, вы увидите, как можно использовать функции вроде `updateNumericInput()` для модификации значения в браузере, что будет приводить к изменению значения переменной `input$count`.

Важно помнить еще одно свойство объекта `input`, а именно его избирательность в отношении того, кто может читать его значение. Чтобы иметь доступ к содержимому объекта `input`, необходимо находиться в *реактивном контексте* (reactive context), созданном функциями вроде `renderText()` или `reactive()`. Очень скоро мы вернемся к этой идее, но важно понимать, что данное ограничение позволяет элементам вывода обновляться автоматически при изменении значений элементов ввода. Во фрагменте кода ниже продемонстрирована эта ошибка:

```
server <- function(input, output, session) {
  message("Значение input$count - ", input$count)
}

shinyApp(ui, server)
#> Error: Can't access reactive value 'count' outside of reactive consumer.
#> i Do you need to wrap inside reactive() or observer()?
```

Output

Объект `output` очень похож на `input`: он также характеризуется списочным типом и содержит переменные, названные в соответствии с идентификаторами элементов вывода. Главным отличием между этими объектами является то, что `input` предназначен для отправки значений элементов ввода, а `output` – вывода. Объект `output` всегда используется совместно с соответствующей ему функцией отображения, как показано в простом примере ниже:

```
ui <- fluidPage(
  textOutput("greeting")
)

server <- function(input, output, session) {
  output$greeting <- renderText("Hello human!")
}
```

Помните, что идентификатор элемента вывода указывается в кавычках при его объявлении в интерфейсе пользователя и без кавычек – при обращении к нему в функции `server`.

Функция отображения выполняет две задачи:

- устанавливает специальный реактивный контекст, автоматически отслеживающий, какие элементы ввода используются в выводе;
- преобразует поток вывода языка R в соответствующий код HTML, пригодный для отображения в браузере.

Как и объект `input`, `output` является весьма требовательным в обращении. К примеру, вы получите на выходе ошибки, если:

- забудете указать функцию отображения:

```
server <- function(input, output, session) {
  output$greeting <- "Hello human"
}
shinyApp(ui, server)
#> Error: Unexpected character object for output$greeting
#> i Did you forget to use a render function?
```

- попытаетесь выполнить чтение из объекта `output`:

```
server <- function(input, output, session) {
  message("The greeting is ", output$greeting)
}
shinyApp(ui, server)
#> Error: Reading from shinyoutput object is not allowed.
```

РЕАКТИВНОЕ ПРОГРАММИРОВАНИЕ

Приложение, в котором есть только элементы ввода или элементы вывода, – это скучное приложение. Настоящая магия Shiny проявляется при наличии обоих объектов. Давайте рассмотрим простой пример:

```
ui <- fluidPage(
  textInput("name", "What's your name?"),
  textOutput("greeting")
)

server <- function(input, output, session) {
  output$greeting <- renderText({
```

```

    paste0("Hello ", input$name, "!")
  })
}

```

В книге очень трудно показать работу приложения в динамике, но я сделал это как смог на рис. 3.1. Если вы запустите приложение и начнете писать свое имя в поле ввода, вы увидите, что приветствие будет обновляться автоматически по мере ввода символов¹.



Рис. 3.1 ❖ Реактивность приложения

обуславливает автоматическое обновление содержимого элементов.

Опробовать это приложение можно по адресу <https://hadley.shinyapps.io/ms-connection>

В этом состоит основная идея Shiny – вы не должны говорить элементу вывода, когда обновляться, Shiny сделает это за вас. Как это работает? Что именно происходит в теле функции? Давайте внимательнее присмотримся к коду функции `server`:

```

output$greeting <- renderText({
  paste0("Hello ", input$name, "!")
})

```

Это можно прочитать как «Соедини вместе слово *Hello*, введенное имя и восклицательный знак и отправь обработанный текст в `output$greeting`». В такой модели поведения есть небольшая, но важная неточность. Дело в том, что в этой модели вы предполагаете, что инструкция отправляется лишь раз. Но Shiny действует не так, он посылает эту инструкцию каждый раз, когда обновляется содержимое поля `input$name`.

По сути, представленный выше код не говорит Shiny собрать строку и отправить ее в браузер – вместо этого он информирует Shiny о том, как необходимо собрать строку при необходимости. Дальше Shiny сам решает, когда запускать код, и запускать ли его вообще. Он может быть выполнен сразу после запуска приложения, а может и гораздо позже, может быть запущен несколько раз, а может и не выполняться вовсе. Дело не в капризах Shiny, просто это его ответственность – решать, когда запускать код, – а не ваша. Думайте о своем приложении как о сборнике инструкций для Shiny, а не о перечне команд.

¹ При запуске приложения вы можете обнаружить, что для обновления строки приветствия после каждой буквы вам придется вводить символы в поле достаточно медленно. Это происходит из-за того, что Shiny использует технику, получившую название *дебаунсинг* (debouncing), заключающуюся в ожидании в течение нескольких секунд перед обновлением. Это существенно снижает нагрузку на приложение без ощутимого влияния на его время отклика.

Императивное программирование против декларативного

Разница между командами и инструкциями составляет основу противоречия между двумя базовыми стилями программирования:

- *императивное программирование* (imperative programming): содержащиеся в коде программы команды выполняются сразу, как только до них доходит очередь. Такой стиль программирования вы обычно применяете в программах для анализа данных: вы командуете R загрузить данные, преобразовать их, визуализировать и сохранить результаты на диске;
- *декларативное программирование* (declarative programming): вы выражаете цели высшего уровня или описываете важные ограничения, после чего полагаетесь на кого-то, кто решит, когда и надо ли приводить в исполнение ваши инструкции. Именно такой стиль программирования применяется в Shiny.

В случае с императивным программированием вы говорите «Сделай мне бутерброд»¹. В декларативном коде фраза звучит так: «Убедись, что бутерброд будет лежать в холодильнике, когда я захочу его съесть». Императивный код настойчив, а декларативный – пассивно-агрессивен.

В большинстве случаев декларативный стиль программирования воспринимается как весьма вольный: вы описываете свои планы, а программа сама решает, как и когда их реализовать. Недостатком этого стиля является то, что иногда вы точно знаете, чего хотите добиться, но не понимаете, как сформулировать это в виде, понятном для движка декларативного программирования². Цель этой книги – помочь вам понять принципы, лежащие в основе декларативного программирования, чтобы такие проблемы возникали как можно реже.

Ленивые вычисления

Одним из преимуществ декларативного программирования в Shiny является то, что оно позволяет приложению быть очень *ленивым* (lazy). Приложение Shiny будет выполнять минимум работы, требуемой для обновления элементов вывода, которые вы видите в текущий момент³. В то же время принципы ленивого программирования таят в себе и некоторую опасность, о которой нужно знать. Посмотрите на серверную функцию ниже. Что в ней не так?

```
server <- function(input, output, session) {
  output$greting <- renderText({
    paste0("Hello ", input$name, "!")
  })
}
```

¹ Вы можете посмотреть комиксы на эту тему по адресу <https://xkcd.com/149>.

² Если вы когда-нибудь сталкивались со сложностями в отображении легенды на диаграмме ggplot2, вы понимаете, о чем речь.

³ Shiny действительно не будет обновлять элементы вывода, которые вы не видите в браузере! Он настолько ленив, что не станет выполнять работу, результаты которой вы все равно не заметите.

```
  })  
}
```

Если вы присмотритесь внимательнее, то заметите, что я написал `greeting` вместо `greeting`. Выполнение такого кода не приведет к ошибке в Shiny, но и своих функций программа выполнять не будет. Элемента вывода с идентификатором `greeting` просто не существует, так что блок кода внутри функции `renderText()` никогда не будет выполнен.

Таким образом, если, работая с приложением Shiny, вы обнаружите, что некий код не выполняется, дважды проверьте, что в функциях интерфейса пользователя и сервера обращение к элементам производится по одному идентификатору.

Реактивный график

У ленивых вычислений, характерных для Shiny, есть еще одно важное свойство. В большинстве программ, написанных на языке R, вы можете отследить логику их выполнения, читая код последовательно сверху вниз. В Shiny такой подход не сработает, поскольку код здесь выполняется по мере необходимости. Таким образом, для понимания логики выполнения программы вам придется обратиться к *реактивному графику* (reactive graph), на котором показано, как элементы ввода связаны с элементами вывода. Реактивный график для предыдущего простого приложения будет выглядеть так, как показано на рис. 3.2.



Рис. 3.2 ❖ Реактивный график
со связями между элементами ввода и вывода

На графике содержится по одной фигуре для каждого элемента ввода и вывода, и мы соединяем их, когда элемент вывода получает доступ к элементу ввода. По этому графику видно, что элемент `greeting` должен пересчитываться всякий раз, когда меняется значение элемента `name`. Для описания этой связи мы будем говорить, что у `greeting` есть реактивная зависимость от `name`.

Обратите внимание на графические обозначения, использованные для элементов ввода и вывода: элемент `name` естественным образом подходит для связки с элементом `greeting`. Мы могли бы изобразить их в виде жесткой сцепки, как показано на рис. 3.3, но обычно не будем этого делать, поскольку это подходит только для самых простых приложений.



Рис. 3.3 ❖ Фигуры для элементов выбраны так,
чтобы была отчетлива видна направленность связи между ними

Реактивный график является превосходным инструментом для понимания работы приложений Shiny. При росте сложности приложения бывает полезно сделать набросок такого графика, чтобы уяснить для себя, как должны взаимодействовать элементы между собой. На протяжении книги мы будем показывать вам реактивные графики приложений для лучшего понимания их работы, а в главе 14 вы познакомитесь с пакетом *reactlog*, который умеет рисовать такие графики.

Реактивные выражения

Еще один важный компонент, входящий в состав реактивных графиков, – это *реактивные выражения* (reactive expression). Позже мы поговорим об этом понятии более подробно, а сейчас вы можете думать о реактивных выражениях как об инструменте, позволяющем сократить количество дублирующегося кода путем введения дополнительных узлов на реактивном графике.

В нашем простом приложении нет никакого смысла вводить дополнительное реактивное выражение, но мы сделаем это, чтобы показать, как это отразится на реактивном графике, показанном на рис. 3.4.

```
server <- function(input, output, session) {
  string <- reactive(paste0("Hello ", input$name, "!"))
  output$greeting <- renderText(string())
}
```

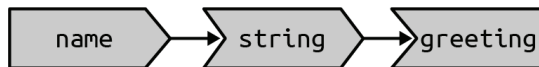


Рис. 3.4 ❖ Реактивное выражение на графике показано с входящей и исходящей стрелками, поскольку оно связано и с элементом ввода, и с элементом вывода

Реактивные выражения принимают элементы ввода и способствуют образованию элементов вывода, поэтому их геометрическая форма на графике напоминает одновременно элементы ввода и вывода. Внешний вид фигур на реактивном графике позволит вам легче запомнить назначение каждого из соответствующих элементов.

Порядок выполнения

Очень важно понимать, что *порядок выполнения кода* (execution order) вашего приложения определяется исключительно реактивным графиком. При этом он будет значительно отличаться от порядка выполнения традиционного кода на языке R, который определяется порядком следования строк. К примеру, мы могли бы легко изменить порядок следования двух строк в нашем предыдущем примере следующим образом:

```
server <- function(input, output, session) {
  output$greeting <- renderText(string())
  string <- reactive(paste0("Hello ", input$name, "!"))
}
```

Вы могли бы подумать, что такой код вернет ошибку, поскольку строка присваивания выражения `output$greeting` ссылается на реактивное выражение `string`, которое на этот момент еще не определено. Но не забывайте о том, что Shiny очень ленив, так что этот код будет запущен только после старта сессии, после того как переменная `string` уже будет создана.

Представленному выше коду будет соответствовать такой же точно реактивный график, как и раньше, так что порядок выполнения кода от перестановки строк в нем не изменится. При этом стоит отметить, что подобные перестановки могут вводить разработчиков в заблуждение, поэтому лучше так не делать. Вместо этого старайтесь, чтобы ваши реактивные выражения и элементы вывода ссылались на переменные, определенные выше, а не ниже¹. Это сделает ваш код более простым для понимания.

Представленная здесь концепция существенно отличается от принципов, применимых к традиционному программированию на R, поэтому я повторю еще раз: порядок, в котором выполняется реактивный код, определяется только реактивным графиком, а не последовательностью строк в функции `server`.

Упражнения

Упражнение 1

Есть код пользовательского интерфейса:

```
ui <- fluidPage(
  textInput("name", "What's your name?"),
  textOutput("greeting")
)
```

Исправьте ошибки в трех следующих вариантах функции `server`. Постарайтесь найти ошибки, просто читая код. После этого проверьте правильность внесенных вами исправлений:

```
server1 <- function(input, output, server) {
  input$greeting <- renderText(paste0("Hello ", name))
}
```

```
server2 <- function(input, output, server) {
  greeting <- paste0("Hello ", input$name)
  output$greeting <- renderText(greeting)
}
```

```
server3 <- function(input, output, server) {
```

¹ Для такого порядка следования инструкций есть технический термин *топологическая сортировка* (topological sort).

```
    output$greting <- paste0("Hello", input$name)
  }
```

Упражнение 2

Нарисуйте реактивные графики для следующих функций server:

```
server1 <- function(input, output, session) {
  c <- reactive(input$a + input$b)
  e <- reactive(c() + input$d)
  output$f <- renderText(e())
}

server2 <- function(input, output, session) {
  x <- reactive(input$x1 + input$x2 + input$x3)
  y <- reactive(input$y1 + input$y2)
  output$z <- renderText(x() / y())
}

server3 <- function(input, output, session) {
  d <- reactive(c() ^ input$d)
  a <- reactive(input$a * 10)
  c <- reactive(b() / input$c)
  b <- reactive(a() + input$b)
}
```

Упражнение 3

Почему представленный ниже код выдает ошибку?

```
var <- reactive(df[[input$var]])
range <- reactive(range(var(), na.rm = TRUE))
```

Почему имена `range()` и `var()` являются нелучшим выбором для определения реактивных выражений?

РЕАКТИВНЫЕ ВЫРАЖЕНИЯ

Ранее мы уже несколько раз упоминали в книге реактивные выражения, так что вы должны примерно представлять, что они из себя представляют. В данном разделе мы поговорим о них более подробно и покажем, чем обусловлена их важность при написании настоящих приложений.

Реактивные выражения невероятно важны, поскольку они дают Shiny дополнительную информацию, позволяющую производить меньше вычислений при изменении элементов ввода, делают приложения более эффективными и облегчают понимание работы приложений за счет упрощения реактивных графиков. Реактивные выражения одинаково полезны и как элементы ввода, и как элементы вывода:

- в качестве элементов ввода реактивные выражения могут передавать свои результаты соответствующим элементам вывода;
- в качестве элементов вывода реактивные выражения зависят от элементов ввода и автоматически обновляются.

С учетом этой двойственности поведения реактивных выражений нам понадобится ввести новые термины: давайте будем называть элементы ввода в связке с реактивными выражениями *поставщиками* (producer), а реактивные выражения, являющиеся источником для элементов вывода, – *потребителями* (consumer). На рис. 3.5 показаны эти связки при помощи диаграммы Венна.

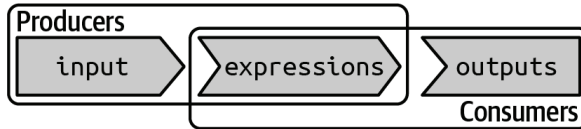


Рис. 3.5 ❖ Элементы ввода и выражения – реактивные поставщики, а выражения и элементы вывода – реактивные потребители

Нам понадобится более сложное приложение, чтобы понять все преимущества использования реактивных выражений. Но сначала напомним пару функций на языке R, которые используем в нашем приложении.

Предпосылки

Представьте, что есть два вымышленных набора данных, которые нам необходимо сравнить при помощи графика и критерия для проверки гипотез. Я немного поэкспериментировал и пришел к приведенным ниже функциям: функция `freqpoly()` визуализирует два распределения при помощи *многоугольника частот* (frequency polygon)¹, а функция `t_test()` использует *t-критерий* (t-test) для сравнения средних значений и создания текстовой сводки:

```
library(ggplot2)

freqpoly <- function(x1, x2, binwidth = 0.1, xlim = c(-3, 3)) {
  df <- data.frame(
    x = c(x1, x2),
    g = c(rep("x1", length(x1)), rep("x2", length(x2)))
  )

  ggplot(df, aes(x, colour = g)) +
    geom_freqpoly(binwidth = binwidth, size = 1) +
    coord_cartesian(xlim = xlim)
}

t_test <- function(x1, x2) {
  test <- t.test(x1, x2)
}
```

¹ Если вы незнакомы с термином *многоугольник частот*, считайте, что это обычная гистограмма, в которой используются линии вместо столбиков, – так проще сравнивать несколько наборов данных на одном графике.

```
# используем функцию sprintf() для компактного форматирования результатов
sprintf(
  "p value: %0.3f\n[%0.2f, %0.2f]",
  test$p.value, test$conf.int[1], test$conf.int[2]
)
}
```

Если взять два набора данных, можно использовать эти функции для сравнения двух переменных:

```
x1 <- rnorm(100, mean = 0, sd = 0.5)
x2 <- rnorm(200, mean = 0.15, sd = 0.9)

freqpoly(x1, x2)
cat(t_test(x1, x2))
#> p value: 0.003
#> [-0.38, -0.08]
```

Результат показан на рис. 3.6.

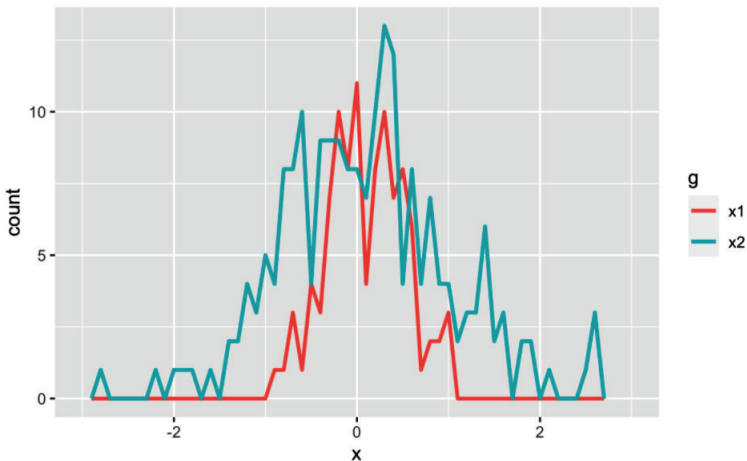


Рис. 3.6 ❖ Сравнение наборов данных

В реальном анализе вам, вероятно, потребовалось бы дать массу объяснений перед написанием этих функций. Но мы решили опустить все эти нюансы, чтобы поскорее перейти к самому приложению. Вынесение блоков императивного кода в отдельные функции – это очень важная техника при проектировании и разработке приложений Shiny: чем больше кода вы сможете вынести за пределы приложения, тем легче будет понять работу приложения. Это хорошая практика в программировании, нацеленная на разделение кода по назначению: внешние функции сосредоточены на вычислениях, тогда как программный код приложения занимается обработкой взаимодействий с пользователем. Мы вернемся к этой идее в главе 18.

Приложение

Итак, мне не терпится использовать написанные нами инструменты для проведения моделирования. Приложение Shiny – прекрасный выбор для этого, поскольку позволяет избежать бесконечных изменений в коде и повторных запусков. Давайте просто соберем все элементы ввода в приложении Shiny и позволим пользователю самому настраивать входные данные.

Начнем с интерфейса пользователя. Далее в книге мы подробно остановимся на том, как работает сочетание функций `fluidRow()` и `column()`, но вы и сами можете догадаться об этом по их именам. В первой строке у нас будет три колонки для элементов ввода (два распределения и управление графиком). Во второй строке мы разместим одну широкую колонку для графика и узкую для результатов проверки гипотез:

```
ui <- fluidPage(
  fluidRow(
    column(4,
      "Distribution 1",
      numericInput("n1", label = "n", value = 1000, min = 1),
      numericInput("mean1", label = "μ", value = 0, step = 0.1),
      numericInput("sd1", label = "σ", value = 0.5, min = 0.1, step = 0.1)
    ),
    column(4,
      "Distribution 2",
      numericInput("n2", label = "n", value = 1000, min = 1),
      numericInput("mean2", label = "μ", value = 0, step = 0.1),
      numericInput("sd2", label = "σ", value = 0.5, min = 0.1, step = 0.1)
    ),
    column(4,
      "Frequency polygon",
      numericInput("binwidth", label = "Bin width", value = 0.1, step = 0.1),
      sliderInput("range", label = "range", value = c(-3, 3), min = -5, max = 5)
    )
  ),
  fluidRow(
    column(9, plotOutput("hist")),
    column(3, verbatimTextOutput("ttest"))
  )
)
```

В функции `server` мы выведем на экран соответствующие графики и вызовем подготовленные заранее функции `freqpoly()` и `t_test()`:

```
server <- function(input, output, session) {
  output$hist <- renderPlot({
    x1 <- rnorm(input$n1, input$mean1, input$sd1)
    x2 <- rnorm(input$n2, input$mean2, input$sd2)

    freqpoly(x1, x2, binwidth = input$binwidth, xlim = input$range)
  }, res = 96)
```

```

output$ttest <- renderText({
  x1 <- rnorm(input$n1, input$mean1, input$sd1)
  x2 <- rnorm(input$n2, input$mean2, input$sd2)

  t_test(x1, x2)
})

```

Результат запуска этого приложения показан на рис. 3.7. Я рекомендую вам открыть это приложение в браузере по адресу <https://hadley.shinyapps.io/ms-case-study-1> и поиграть с параметрами, чтобы точно убедиться, что вы понимаете, как работает это приложение.

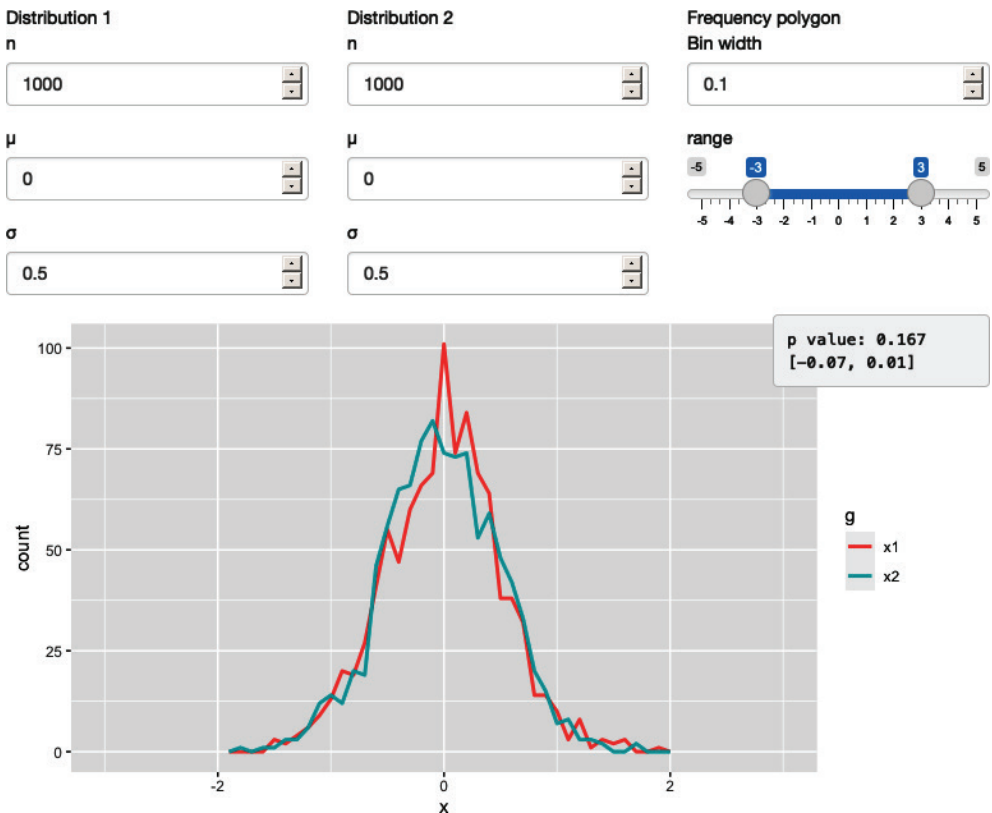


Рис. 3.7 ❖ Это приложение Shiny позволяет сравнить два распределения при помощи t-критерия и многоугольника частот

Реактивный график

Давайте попробуем нарисовать реактивный график нашего приложения. Shiny достаточно умен, чтобы обновлять элементы вывода только при изменении значений соответствующих элементов ввода, но недостаточно для

того, чтобы выполнять строки кода в функции вывода выборочно. Иными словами, функции вывода атомарны – они либо выполняются целиком, либо не выполняются вовсе.

Возьмем для примера такой фрагмент кода из функции `server`:

```
x1 <- rnorm(input$n1, input$mean1, input$sd1)
x2 <- rnorm(input$n2, input$mean2, input$sd2)
t_test(x1, x2)
```

Будучи человеком разумным, вы понимаете, что переменную `x1` необходимо пересчитывать только при изменении значений в переменных `n1`, `mean1` или `sd1`, а переменную `x2` – при изменении `n2`, `mean2` или `sd2`. Shiny же видит картину иначе: он будет обновлять `x1` и `x2` всякий раз, когда будет изменяться один из следующих параметров: `n1`, `mean1`, `sd1`, `n2`, `mean2` или `sd2`. Таким образом, реактивный график этой связки будет выглядеть так, как показано на рис. 3.8.

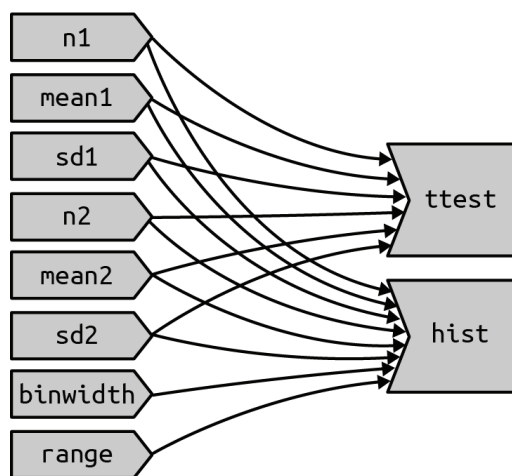


Рис. 3.8 ❖ Реактивный график:
все выходы связаны со всеми вводами

Как видите, график получился довольно насыщенным линиями – почти все элементы ввода связаны со всеми элементами вывода. Это может приводить к следующим проблемам:

- приложение будет гораздо более сложным для понимания из-за большого количества связей между элементами. К тому же вам просто не удастся извлечь определенный фрагмент кода и проанализировать его в отрыве от остальной программы;
- снизится производительность приложения по причине выполнения большого количества лишней работы. К примеру, если поменять интервалы на графике, данные пересчитаются, и если изменится значение `n1`, переменная `x2` обновится (в двух местах!).

В нашем приложении также есть еще одна серьезная проблема, связанная с тем, что многоугольник частот и t -критерий используют разные случайные

данные. Это приводит в замешательство, поскольку мы ожидаем, что будем анализировать одни и те же исходные данные.

К счастью, все эти недостатки мы можем устранить при помощи реактивных выражений и повторного использования кода.

Упрощение реактивного графика

В следующем примере мы провели реорганизацию нашего кода из функции `server`, введя реактивные выражения `x1` и `x2`, в которых формируются исходные наборы данных для анализа. Для создания реактивного выражения необходимо вызвать функцию `reactive()` и присвоить ее результат переменной. В дальнейшем мы будем обращаться к созданным переменным как к функциям:

```
server <- function(input, output, session) {
  x1 <- reactive(rnorm(input$n1, input$mean1, input$sd1))
  x2 <- reactive(rnorm(input$n2, input$mean2, input$sd2))

  output$hist <- renderPlot({
    freqpoly(x1(), x2(), binwidth = input$binwidth, xlim = input$range)
  }, res = 96)

  output$ttest <- renderText({
    t_test(x1(), x2())
  })
}
```

В результате этого изменения мы получим гораздо более компактный реактивный график приложения, показанный на рис. 3.9.

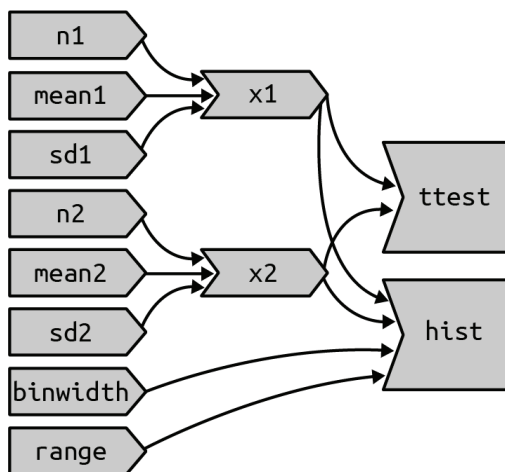


Рис. 3.9 ❖ Использование реактивных выражений позволило существенно упростить график приложения, сделав его более легким для восприятия

Приложение с таким простым реактивным графиком понять будет намного легче, поскольку мы при желании можем рассматривать связанные компоненты отдельно от остальных. Взгляните на реактивные выражения `x1` и `x2` – теперь они зависят только от параметров своих распределений, а изменение этих параметров затрагивает лишь одно выражение. Как вы понимаете, произведенные изменения существенно повысят производительность нашего приложения, поскольку вычислений в нем будет производиться гораздо меньше. К тому же теперь при изменении параметров `binwidth` и `range` будет обновляться лишь график, а исходные данные останутся прежними.

Чтобы подчеркнуть выраженную *модульность* (modularity) обновленного приложения, мы на рис. 3.10 независимые компоненты объединили единым контуром. В главе 19, когда мы будем подробно обсуждать модули, мы вернемся к этой концепции. Если говорить кратко, *модули* (module) позволяют выделить повторяющийся код для повторного использования, тем самым гарантируя его изолированность от остальных фрагментов кода в приложении. Модули представляют собой исключительно полезную и мощную технику применительно к сложным приложениям.

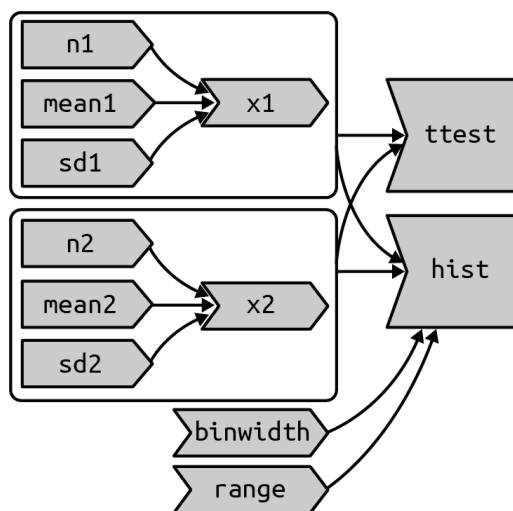


Рис. 3.10 ❖ Модули позволяют изолировать часть кода от приложения

Возможно, вы знакомы с так называемым правилом трех копирований в программировании, говорящим о том, что если вы трижды скопировали и вставили фрагмент кода, пришло время задуматься об использовании этого фрагмента в качестве повторяющегося блока, – обычно путем написания функции. Следовать этому правилу очень важно, ведь так вы сможете существенно снизить количество повторений в вашей программе, а приложение в целом станет легче для понимания и поддержки.

Полагаю, в Shiny будет более уместно применять правило одного копирования – если вы хоть раз скопировали и вставили фрагмент кода, пора подумать о выделении его в реактивное выражение. Мы пошли на такое

ужесточение правила в Shiny, поскольку реактивные выражения не только облегчают понимание программного кода приложения, но и значительно повышают его эффективность в целом.

Зачем нужны реактивные выражения?

Впервые встретив реактивный код в приложении Shiny, вы могли подумать: а зачем мне это все? Почему нельзя воспользоваться имеющимися инструментами в виде переменных и функций для уменьшения количества дублирующегося кода? К сожалению, ни одна из этих техник не работает в *реактивном окружении* (reactive environment).

Если бы вы захотели использовать переменные для сокращения дублирования кода, вы бы могли попытаться написать нечто похожее:

```
server <- function(input, output, session) {
  x1 <- rnorm(input$n1, input$mean1, input$sd1)
  x2 <- rnorm(input$n2, input$mean2, input$sd2)

  output$hist <- renderPlot({
    freqpoly(x1, x2, binwidth = input$binwidth, xlim = input$range)
  }, res = 96)

  output$ttest <- renderText({
    t_test(x1, x2)
  })
}
```

Если вы запустите этот код, то получите ошибку, говорящую о том, что вы пытаетесь получить доступ к значению элемента ввода вне реактивного контекста. Но даже если бы вы не получили ошибку, у вас были бы проблемы – переменные `x1` и `x2` вычислялись бы лишь раз при старте сессии, а не каждый раз при изменении значений элементов ввода.

Если вы попытаетесь обойти это ограничение при помощи функций, получится код, показанный ниже:

```
server <- function(input, output, session) {
  x1 <- function() rnorm(input$n1, input$mean1, input$sd1)
  x2 <- function() rnorm(input$n2, input$mean2, input$sd2)

  output$hist <- renderPlot({
    freqpoly(x1(), x2(), binwidth = input$binwidth, xlim = input$range)
  }, res = 96)

  output$ttest <- renderText({
    t_test(x1(), x2())
  })
}
```

Это приложение будет работать, однако мы вернемся к нашей изначальной проблеме, от которой пытаемся избавиться, – изменение значения любого элемента ввода приведет к пересчету всех элементов вывода, а *t*-критерий

и многоугольник частот будут построены на основании разных выборок данных. При расчете реактивных выражений результаты автоматически кешируются и пересчитываются только при изменении значений соответствующих элементов ввода¹.

Тогда как переменные вычисляются лишь раз (недолет), а функции пересчитывают значения при каждом вызове (перелет), реактивные выражения производят вычисления только тогда, когда значения могут измениться (в точку).

КОНТРОЛЬ ВРЕМЕНИ ЗАПУСКА РЕАКТИВНЫХ ВЫРАЖЕНИЙ

Теперь, когда вы познакомились с основной идеей реактивного программирования, мы обсудим две продвинутые техники, позволяющие контролировать время запуска реактивных выражений. Подробнее о них мы будем говорить в главе 15, а сейчас пройдемся по их основам.

Для представления вам этих техник я немного упрощу наше предыдущее приложение. Давайте будем использовать распределение с единственным параметром, и оба набора данных будут построены на основании одного и того же числа n . Также уберем элементы управления графиком. В результате получим приложение с простым пользовательским интерфейсом и функцией `server`, как показано ниже:

```
ui <- fluidPage(
  fluidRow(
    column(3,
      numericInput("lambda1", label = "lambda1", value = 3),
      numericInput("lambda2", label = "lambda2", value = 5),
      numericInput("n", label = "n", value = 1e4, min = 0)
    ),
    column(9, plotOutput("hist"))
  )
)
server <- function(input, output, session) {
  x1 <- reactive(rpois(input$n, input$lambda1))
  x2 <- reactive(rpois(input$n, input$lambda2))
  output$hist <- renderPlot({
    freqpoly(x1(), x2(), binwidth = 1, xlim = c(0, 40))
  }, res = 96)
}
```

Наше приложение будет выглядеть так, как показано на рис. 3.11, а реактивный график будет как на рис. 3.12.

¹ Если вы знакомы с таким понятием, как *мемоизация* (memoization), здесь смысл схожий.

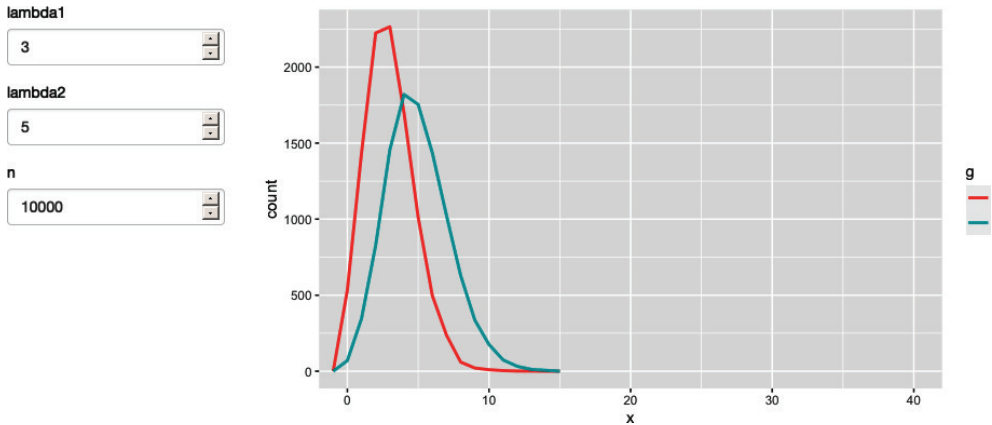


Рис. 3.11 ❖ Простое приложение с многоугольниками частот на основании случайных чисел, выбранных из двух распределений Пуассона. Посмотреть это приложение в браузере можно по адресу <https://hadley.shinyapps.io/ms-simulation-2>

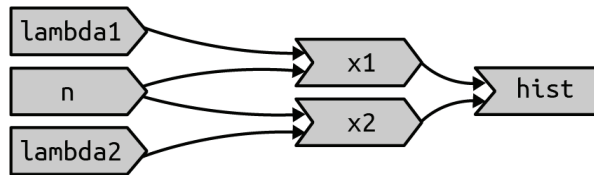


Рис. 3.12 ❖ Реактивный график приложения

Обновление по расписанию

Представьте, что вам захотелось усилить впечатление от того, что исходные данные получены случайным образом, путем выполнения циклической повторной симуляции. Это приведет к созданию своеобразной анимации вместо статического графика¹. Частоту обновлений можно регулировать при помощи функции `reactiveTimer()`.

`reactiveTimer()` представляет собой реактивное выражение, зависящее от скрытого элемента ввода в виде текущего времени. Функцию `reactiveTimer()` можно использовать, если вы хотите, чтобы реактивное выражение принимало статус *недействительного* (*invalidate*) чаще, чем в обычной ситуации. К примеру, в приведенном ниже коде мы использовали временной интервал 500 мс, так что график будет обновляться дважды в секунду. Это достаточно часто, чтобы продемонстрировать выполнение симуляций в реальном времени, но не так часто, чтобы свести зрителя с ума. Показанные ниже изменения в приложении приведут к обновлению реактивного графика, новый вид которого показан на рис. 3.13.

¹ На сайте New York Times эту технику, в частности, использовали в статье о том, как можно интерпретировать данные в отчете о безработице.

```
server <- function(input, output, session) {
  timer <- reactiveTimer(500)

  x1 <- reactive({
    timer()
    rpois(input$n, input$lambda1)
  })
  x2 <- reactive({
    timer()
    rpois(input$n, input$lambda2)
  })

  output$hist <- renderPlot({
    freqpoly(x1(), x2(), binwidth = 1, xlim = c(0, 40))
  }, res = 96)
}
```

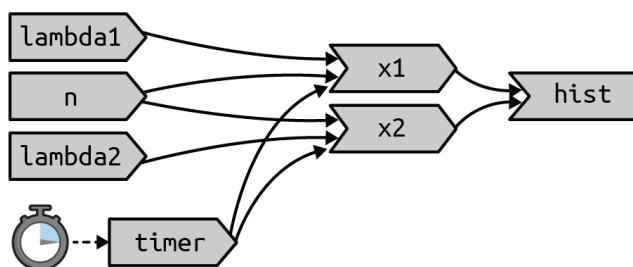


Рис. 3.13 ❖ Вызов функции `reactiveTimer(500)` привел к образованию нового элемента ввода, каждые полсекунды утрачивающего актуальность

Обратите внимание, как мы применяем `timer()` в реактивном выражении, вычисляющем `x1()` и `x2()`: мы вызываем функцию, но не используем значение. Это позволяет реактивным выражениям `x1` и `x2` обрести зависимость от `timer` вне зависимости от возвращаемого значения.

Щелчки мыши

Представьте, что было бы, если бы в предыдущем сценарии код симуляции выполнялся целую секунду. С учетом того, что симуляции запускаются каждые полсекунды, у Shiny постепенно будут накапливаться все новые и новые задачи, выполнить которые просто не хватит времени. То же самое будет происходить, если кто-то в вашем приложении будет часто нажимать на кнопки, запускающие дорогостоящие операции. В результате Shiny будет накапливать невыполненные задания, и пока они будут находиться в очереди, приложение не сможет реагировать на новые события, что вряд ли понравится пользователю.

Если подобные ситуации регулярно возникают в вашем приложении, можно подумать о том, чтобы длительные операции пользователь запускал по нажатию на кнопку. Для этого отлично подойдет элемент `actionButton()`:

```

ui <- fluidPage(
  fluidRow(
    column(3,
      numericInput("lambda1", label = "lambda1", value = 3),
      numericInput("lambda2", label = "lambda2", value = 5),
      numericInput("n", label = "n", value = 1e4, min = 0),
      actionButton("simulate", "Simulate!")
    ),
    column(9, plotOutput("hist"))
  )
)

```

Но, чтобы использовать кнопку, придется изучить новый инструмент. Зачем? А давайте сначала попробуем изменить код приложения с использованием предыдущего подхода. Как и раньше, попробуем обратиться к элементу `simulate` без использования его значения, чтобы установить зависимость от него:

```

server <- function(input, output, session) {
  x1 <- reactive({
    input$simulate
    rpois(input$n, input$lambda1)
  })
  x2 <- reactive({
    input$simulate
    rpois(input$n, input$lambda2)
  })
  output$hist <- renderPlot({
    freqpoly(x1(), x2(), binwidth = 1, xlim = c(0, 40))
  }, res = 96)
}

```

В результате приложение будет выглядеть так, как показано на рис. 3.14, а реактивный график – как на рис. 3.15. Примененный нами подход не помог решить поставленную задачу, а лишь привел к образованию новых зависимостей: `x1()` и `x2()` будут обновляться при нажатии на кнопку `simulate`, но они также продолжают обновляться и при изменении параметров `lambda1`, `lambda2` и `n`. Нам же нужно не добавить зависимости, а заменить существующие.

Выходит, что для правильной реализации предложенного сценария вам все-таки придется изучить новый инструмент, позволяющий использовать значения элементов ввода без создания зависимости от них. В этом нам поможет функция `eventReactive()`, принимающая два аргумента: первый указывает, от какого элемента следует установить зависимость, а второй определяет само вычисление. Это позволит изменять состояние `x1()` и `x2()` только при нажатии на кнопку `simulate`:

```

server <- function(input, output, session) {
  x1 <- eventReactive(input$simulate, {
    rpois(input$n, input$lambda1)
  })

```

```
x2 <- eventReactive(input$simulate, {
  rpois(input$n, input$lambda2)
})
output$hist <- renderPlot({
  freqpoly(x1(), x2(), binwidth = 1, xlim = c(0, 40))
}, res = 96)
}
```

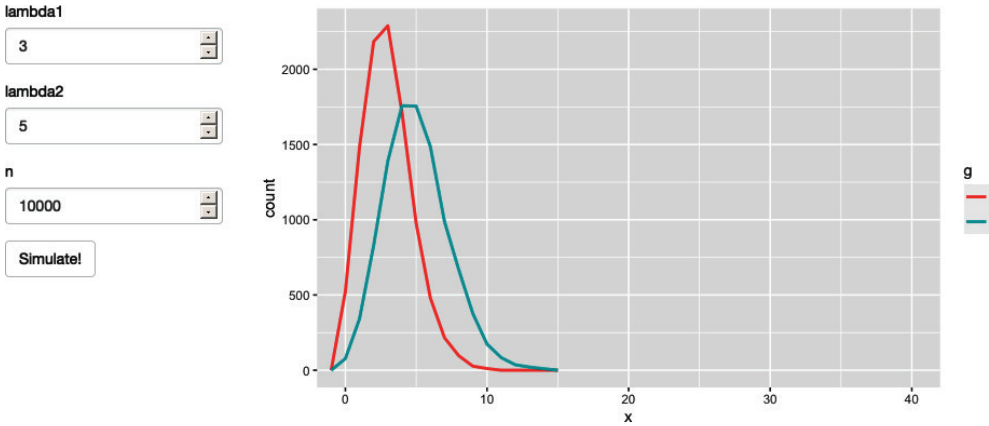


Рис. 3.14. Приложение с кнопкой.

Попробовать приложение в интернете можно по адресу
<https://hadley.shinyapps.io/ms-simulation-2>

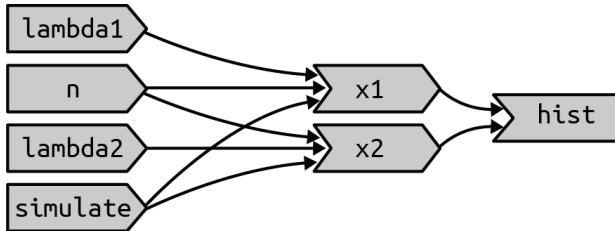


Рис. 3.15 ❖ Реактивный график, не решающий нашей проблемы, – мы просто добавили еще одну зависимость вместо замены существующих зависимостей

На рис. 3.16 показан реактивный график, который получится в результате. Заметьте, что $x1$ и $x2$, как мы и хотели, больше не зависят от элементов $\lambda1$, $\lambda2$ и n , а значит, изменение их значений не приведет к запуску вычисления. На графике я оставил прежние зависимости серым цветом в качестве напоминания о том, что $x1$ и $x2$ продолжают использовать значения элементов ввода, хоть больше и не имеют от них реактивной зависимости.

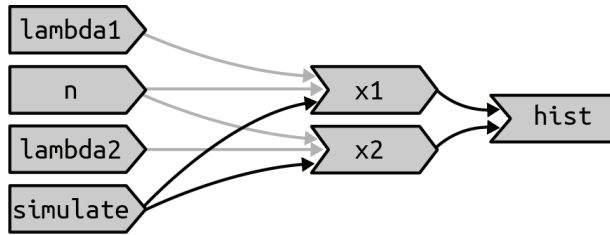


Рис. 3.16 ❖ Функция `eventReactive()` позволяет отделить зависимости (черные стрелки) от значений, используемых для расчетов (серые стрелки)

НАБЛЮДАТЕЛИ

До сих пор мы концентрировались на том, что происходит внутри приложения. Но иногда вам необходимо выполнить действия за пределами приложения. Это может быть сохранение файла на общий сетевой ресурс, отсылка данных веб-API, обновление базы данных или (самый распространенный вариант) печать отладочного сообщения в консоли. Все эти действия никак не влияют на внешний вид вашего приложения, так что нет никакой необходимости применять здесь функции вывода или отображения. Вместо этого мы будем использовать *наблюдателей* (*observer*).

Создавать наблюдателей можно разными способами, и в главе 15 мы обратимся к этой теме более подробно. Сейчас же я хочу показать вам, как использовать функцию `observeEvent()`, поскольку с ее помощью вы сможете отлаживать свои первые приложения Shiny.

Функция `observeEvent()` очень похожа на `eventReactive()`. Она принимает на вход два важных аргумента: `eventExpr` и `handlerExpr`. Первый аргумент представляет собой элемент ввода или выражение для установления зависимости, а второй заключает в себе код для исполнения. Например, следующая модификация функции `server()` предполагает, что при каждом изменении значения элемента ввода `name` будет выводиться сообщение в консоль:

```
ui <- fluidPage(
  textInput("name", "What's your name?"),
  textOutput("greeting")
)

server <- function(input, output, session) {
  string <- reactive(paste0("Hello ", input$name, "!"))

  output$greeting <- renderText(string())
  observeEvent(input$name, {
    message("Greeting performed")
  })
}
```

Между функциями `observeEvent()` и `eventReactive()` есть два важных отличия:

- результаты функций `observeEvent()` не присваиваются переменным;
- как итог вы не можете обратиться к ним из других реактивных потребителей.

Наблюдатели и элементы вывода связаны тесным образом. Вы можете думать об элементах вывода как об особом побочном эффекте в виде обновления HTML в браузере пользователя. Чтобы подчеркнуть эту связь, мы включили эти элементы в обновленный реактивный график, показанный на рис. 3.17.

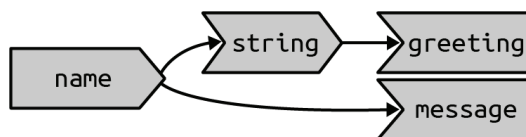


Рис. 3.17 ❖ На реактивном графике наблюдатель выглядит так же, как элемент вывода

ЗАКЛЮЧЕНИЕ

Данная глава должна была улучшить ваше понимание внутренней кухни приложений Shiny, реализованной посредством функции `server()`, отвечающей за взаимодействие с пользователем. Также вы сделали свои первые важные шаги в области парадигмы реактивного программирования, лежащей в основе Shiny. Полученные здесь базовые знания очень пригодятся вам в дальнейшем, и в главе 13 мы вернемся к этой теории и продолжим ее развивать. Реактивное программирование славится своей мощью, но в то же время оно существенно отличается концептуально от обычного императивного стиля программирования, принятого в языке R. Не удивляйтесь, если вам потребуется какое-то время, чтобы впитать и усвоить эти новые для вас подходы.

В этой главе мы заканчиваем знакомить вас с фундаментальными основами Shiny. Следующая глава будет посвящена практике – вы постараетесь применить все полученные ранее знания при создании более сложного приложения Shiny для анализа данных.

Глава 4

Практический пример: несчастные случаи

ВВЕДЕНИЕ

В последних трех главах вы познакомились со множеством полезных концепций, и для того чтобы глубже понять внутренние механизмы Shiny, мы решили предложить вам написать приложение посложнее, объединяющее в себе все идеи, с которыми вы уже встретились в этой книге. Начнем мы с небольшого предварительного анализа данных за пределами Shiny, а затем построим приложение, постепенно двигаясь от простого к сложному.

В данном проекте фреймворку Shiny будут помогать пакеты *vroom* (для быстрого и удобного чтения файлов) и *tidyverse* (для общего анализа данных):

```
library(shiny)
library(vroom)
library(tidyverse)
```

ДАННЫЕ

Мы будем исследовать данные из *Национальной электронной системы по контролю за несчастными случаями* (National Electronic Injury Surveillance System – NEISS), собранные Комиссией по потребительской безопасности (Consumer Product Safety Commission). В этой базе данных собраны все несчастные случаи, зарегистрированные в больницах США. Исследовать ее очень и очень интересно: во-первых, всем нам хорошо знакома эта предметная область, а во-вторых, каждый инцидент содержит текстовое описание произошедшего. Вы можете больше узнать об этом наборе данных на GitHub по адресу <https://github.com/hadley/neiss>.

В данной главе мы сконцентрируемся только на несчастных случаях, произошедших в 2017 году. Это позволит удержать объем набора данных в пределах 10 Гб, чтобы можно было комфортно хранить его на Git вместе с осталь-

ными сопроводительными материалами для книги и не думать о каких-то сложных методах быстрой загрузки данных, о чем мы будем говорить в более поздних главах книги. Код, который я использовал для создания выжимки данных, можно посмотреть на GitHub по адресу <https://github.com/hadley/mastering-shiny/blob/master/neiss/data.R>.

Если вам нужно скачать данные на свой компьютер, запустите следующий код:

```
dir.create("neiss")
#> Warning in dir.create("neiss"): 'neiss' already exists
download <- function(name) {
  url <- "https://github.com/hadley/mastering-shiny/raw/master/neiss/"
  download.file(paste0(url, name), paste0("neiss/", name), quiet = TRUE)
}
download("injuries.tsv.gz")
download("population.tsv")
download("products.tsv")
```

Основной набор данных, который мы будем использовать, – это `injuries`, насчитывающий порядка 250 тыс. наблюдений:

```
injuries <- vroom::vroom("neiss/injuries.tsv.gz")
injuries
#> # A tibble: 255,064 x 10
#>   trmt_date   age sex  race body_part diag      location  prod_code weight
#>   <date>     <dbl> <chr> <chr> <chr> <chr> <chr>          <dbl> <dbl>
#> 1 2017-01-01    71 male  white Upper Tru... Contusion... Other Publ...    1807  77.7
#> 2 2017-01-01    16 male  white Lower Arm  Burns, Th... Home      676  77.7
#> 3 2017-01-01    58 male  white Upper Tru... Contusion... Home      649  77.7
#> 4 2017-01-01    21 male  white Lower Tru... Strain, S... Home     4076  77.7
#> 5 2017-01-01    54 male  white Head      Inter Org... Other Publ...    1807  77.7
#> 6 2017-01-01    21 male  white Hand      Fracture Home      1884  77.7
#> # ... with 255,058 more rows, and 1 more variable: narrative <chr>
```

Каждая строка характеризует отдельный несчастный случай и обладает десятью переменными:

- `trmt_date` – дата приема пациента в клинике (не дата, когда произошел несчастный случай);
- `age`, `sex` и `race` – демографическая информация о пациенте, получившем травму;
- `body_part` – травмированная часть тела (например, *лодыжка* (ankle) или *ухо* (ear));
- `location` – место, где произошла травма (например, дома или в школе);
- `diag` – основной диагноз травмы (например, *перелом* (fracture) или *рваная рана* (laceration));
- `prod_code` – предмет, ассоциированный с травмой;
- `weight` – статистический вес, дающий представление о количестве людей, которые могли бы получить эту травму, если бы набор данных был масштабирован на все население США;
- `narrative` – краткая история получения травмы.

Давайте объединим наш основной набор данных с еще двумя для получения расширенного контекста: набор данных *products* поможет нам найти наименование предмета, ассоциированного с травмой, по его коду, а *population* представит данные о населении США в 2017 году по всем комбинациям возрастов и полов:

```
products <- vroom::vroom("neiss/products.tsv")
products
#> # A tibble: 38 x 2
#>   prod_code title
#>   <dbl> <chr>
#> 1      464 knives, not elsewhere classified
#> 2      474 tableware and accessories
#> 3      604 desks, chests, bureaus or buffets
#> 4      611 bathtubs or showers
#> 5      649 toilets
#> 6      676 rugs or carpets, not specified
#> # ... with 32 more rows

population <- vroom::vroom("neiss/population.tsv")
population
#> # A tibble: 170 x 3
#>   age sex      population
#>   <dbl> <chr>      <dbl>
#> 1     0 female    1924145
#> 2     0 male     2015150
#> 3     1 female    1943534
#> 4     1 male     2031718
#> 5     2 female    1965150
#> 6     2 male     2056625
#> # ... with 164 more rows
```

ОПИСАНИЕ

Перед созданием приложения давайте произведем некоторое описание данных. Начнем с предмета, ассоциированного с травмой, с интересным кодом 649 и еще более интересным наименованием – *toilets* (туалетная комната). Давайте извлечем количество инцидентов, произошедших в таких обстоятельствах:

```
selected <- injuries %>% filter(prod_code == 649)
nrow(selected)
#> [1] 2993
```

Далее произведем некоторые базовые вычисления на основании мест получения травмы, поврежденных частей тела и диагнозов по несчастным случаям, произошедшим в туалетных комнатах. Обратите внимание, что мы будем взвешивать подсчет по переменной *weight*, чтобы полученные цифры можно было интерпретировать как примерное количество травм определенного характера по всем США:

```
selected %>% count(location, wt = weight, sort = TRUE)
#> # A tibble: 6 x 2
#>   location      n
#>   <chr>      <dbl>
#> 1 Home      99603.
#> 2 Other Public Property 18663.
#> 3 Unknown   16267.
#> 4 School     659.
#> 5 Street Or Highway    16.2
#> 6 Sports Or Recreation Place 14.8
```

```
selected %>% count(body_part, wt = weight, sort = TRUE)
#> # A tibble: 24 x 2
#>   body_part      n
#>   <chr>      <dbl>
#> 1 Head      31370.
#> 2 Lower Trunk 26855.
#> 3 Face      13016.
#> 4 Upper Trunk 12508.
#> 5 Knee       6968.
#> 6 N.S./Unk   6741.
#> # ... with 18 more rows
```

```
selected %>% count(diag, wt = weight, sort = TRUE)
#> # A tibble: 20 x 2
#>   diag      n
#>   <chr>      <dbl>
#> 1 Other Or Not Stated 32897.
#> 2 Contusion Or Abrasion 22493.
#> 3 Inter Organ Injury 21525.
#> 4 Fracture      21497.
#> 5 Laceration     18734.
#> 6 Strain, Sprain    7609.
#> # ... with 14 more rows
```

Как мы и могли предположить, инциденты в туалетных комнатах чаще всего происходят дома. Информация о частях тела, травмирующихся в этих случаях, позволяет сделать вывод, что чаще речь идет о падениях, поскольку голова и лицо обычно не задействуются в использовании туалетной комнаты по назначению. Что касается диагнозов, то мы наблюдаем их большое разнообразие.

Также мы можем исследовать определенные шаблоны по возрасту и полу пациентов. В таблице представленные данные будут не слишком показательными, так что я построил график, продемонстрированный на рис. 4.1, по которому можно лучше проследить тенденции:

```
summary <- selected %>%
  count(age, sex, wt = weight)
summary
#> # A tibble: 208 x 3
#>   age sex      n
#>   <dbl> <chr>  <dbl>
```

```
#> 1    0 female  4.76
#> 2    0 male   14.3
#> 3    1 female 253.
#> 4    1 male  231.
#> 5    2 female 438.
#> 6    2 male  632.
#> # ... with 202 more rows
```

```
summary %>%
  ggplot(aes(age, n, colour = sex)) +
  geom_line() +
  labs(y = "Estimated number of injuries")
```

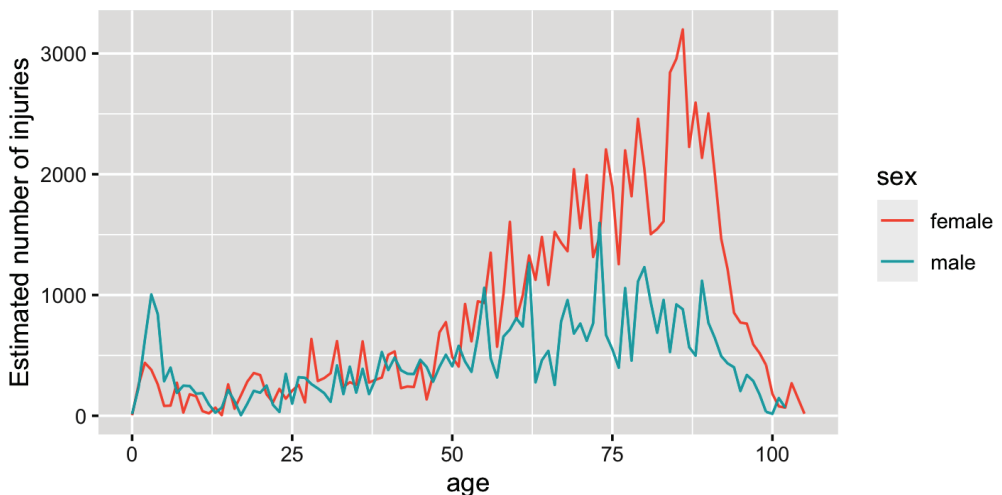


Рис. 4.1 ❖ Количество несчастных случаев в туалетных комнатах по возрасту и полу

Мы видим пик несчастных случаев у мальчиков в районе трех лет, а затем идет плавный рост количества травм в районе среднего возраста (особенно у женщин) и резкое падение показателей после 80 лет. Предполагаю, что пик травм, полученных мальчиками, связан с тем, что они в основном делают свои дела в туалете стоя, а рост количества несчастных случаев у женщин старше среднего возраста я связываю с развитием остеопороза – мужчины и женщины, возможно, получают травмы в предложенных обстоятельствах с одинаковой частотой, но женщины чаще обращаются в клиники по причине большего риска получить перелом.

Проблема с интерпретированием этого шаблона связана с нашим знанием о том, что молодых людей гораздо больше, чем пожилых, что вносит определенный дисбаланс. Избавиться от нее можно путем деления количества травм для определенного сочетания возраста и пола на численность этой группы населения. В результате мы получим коэффициент травматизма. В примере ниже я определил этот коэффициент в расчете на 10 тыс. человек:

```
summary <- selected %>%
  count(age, sex, wt = weight) %>%
  left_join(population, by = c("age", "sex")) %>%
  mutate(rate = n / population * 1e4)
```

```
summary
#> # A tibble: 208 x 5
#>   age sex      n population  rate
#>   <dbl> <chr> <dbl>      <dbl> <dbl>
#> 1     0 female  4.76   1924145 0.0247
#> 2     0 male   14.3   2015150 0.0708
#> 3     1 female 253.    1943534 1.30
#> 4     1 male  231.    2031718 1.14
#> 5     2 female 438.    1965150 2.23
#> 6     2 male  632.    2056625 3.07
#> # ... with 202 more rows
```

График по вычисленному коэффициенту, показанный на рис. 4.2, демонстрирует уже совсем другую тенденцию в возрасте после 50: разница между полами оказалась не столь разительной, и мы больше не наблюдаем резкого спада. Причина этого в том, что женщины, по статистике, живут дольше мужчин, так что в пожилом возрасте их просто больше:

```
summary %>%
  ggplot(aes(age, rate, colour = sex)) +
  geom_line(na.rm = TRUE) +
  labs(y = "Injuries per 10,000 people")
```

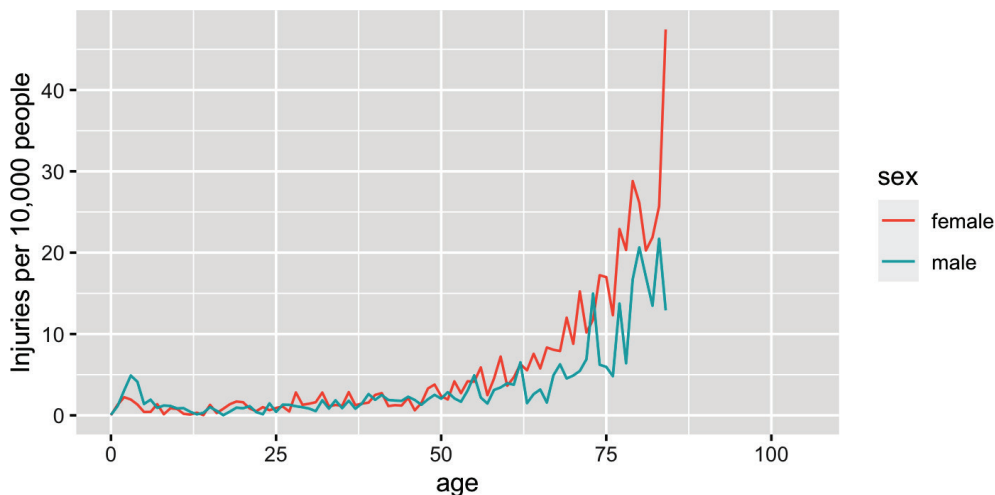


Рис. 4.2 ❖ Коэффициент травматизма
в расчете на 10 тыс. человек по возрасту и полу

Заметьте, что данные здесь представлены до 80 лет, поскольку я просто не смог найти данные о численности населения старше этого возраста.

Наконец, мы можем взглянуть на описание ситуаций, в которых были получены травмы. Читая эти небольшие рассказы, можно проверять наши гипотезы и генерировать идеи для дальнейших исследований. Ниже я приведу случайную выборку из десяти описаний:

```
selected %>%
  sample_n(10) %>%
  pull(narrative)

#> [1] "68YOF STRAINED KNEE MOVING FROM TOILET TO POWER CHAIR AT HOME. DX:..."
#> [2] "97YOM LWR BACK PAIN - MISSED TOILET SEAT, FELL FLOOR AT NH"
#> [3] "54 YOF DX ALCOHOL INTOXICATION - PT STATES SHE FELL OFF TOILET."
#> [4] "85YOF-STAFF AT NH STATES PT WAS TRANSITIONIN TO TOILET FROM WHEELCH..."
#> [5] "FOREHEAD LACERATION. 64 YOM FELL AND HIT HIS HEAD ON TOILET."
#> [6] "70YOM-STAFF STATES PT FELL OFF TOILET ONTO CONCRETE FLOOR AT *** AR..."
#> [7] "40YOF WAS INTOXICATED AND FELL OFF THE TOILET STRUCK HEAD ON THE WA..."
#> [8] "66 Y/O F FELL FROM COMMODE ONTO FLOOR AND FRACTURED CLAVICLE"
#> [9] "25YOF SYNCOPAL EPS W ON TOILET FELL HIT RS OF HEAD REPORTLY LOC UNK..."
#> [10] "4 YO M W/LAC TO FOREHEAD SLIPPED IN BATHROOM HIT ON TOILET FLUSH HA..."
```

Проведя такой анализ по одному предмету, ассоциированному с травмой, было бы здорово перенести процедуру на остальные предметы без необходимости заново писать весь код. Так давайте напомним приложение Shiny!

Прототип

При разработке сложного приложения я настоятельно рекомендую начинать с самых простых вещей, чтобы убедиться, что базовая механика работает как надо, а после этого уже добавлять более сложные элементы. В данном случае мы начнем с одного элемента ввода (код предмета), трех таблиц и одного графика.

Разрабатывая свой первый прототип приложения, вы столкнетесь с настоящим вызовом, пытаясь сделать его максимально простым, насколько это возможно. Бывает очень трудно достичь нужного компромисса между простотой и сложностью. Обе крайности плохи: если сделать приложение изначально слишком простым, в дальнейшем потребуется много времени потратить на его доработку, а если слишком сложным и замысловатым, большая часть написанного вами кода впоследствии может пойти под нож. Для достижения правильного баланса я обычно делаю несколько предварительных набросков карандашом для понимания структуры пользовательского интерфейса и реактивного графика будущего приложения.

В нашем приложении я решил выделить одну строку для элементов ввода с учетом того, что наверняка добавлю сюда еще несколько элементов, прежде чем приложение будет выпущено. Еще одну строку отведем для всех трех таблиц (каждой дадим по четыре колонки в ширину, то есть ровно по трети от нашей общей ширины в 12 колонок), а нижнюю строку в приложении оставим для графика:

```

prod_codes <- setNames(products$prod_code, products$title)

ui <- fluidPage(
  fluidRow(
    column(6,
      selectInput("code", "Product", choices = prod_codes)
    ),
    fluidRow(
      column(4, tableOutput("diag")),
      column(4, tableOutput("body_part")),
      column(4, tableOutput("location"))
    ),
    fluidRow(
      column(12, plotOutput("age_sex"))
    )
  )
)

```

Мы пока так и не поговорили о таких важных функциях, как `fluidRow()` и `column()`, но, думаю, по их названиям и контексту вы догадались, что они делают. Подробно мы разберем их в главе 6, когда будем говорить о структуре многострочных приложений. Также обратите внимание на использование функции `setNames()` в списочном элементе ввода `selectInput()`: такой подход позволяет показывать пользователю наименования предметов, а при выборе возвращать их коды.

Функция `server` в данном случае будет довольно простой. Посмотрите, как мы преобразовали статические переменные `selected` и `summary` в реактивные выражения. Это очень полезный общий шаблон: при анализе данных вы создаете переменные с целью разделения процесса на шаги и избежания повторных вычислений. И в приложениях Shiny эту роль играют реактивные выражения.

Зачастую бывает полезно потратить немного времени на очистку вашего кода для анализа перед созданием приложения Shiny, чтобы решить все эти задачи при помощи традиционного программирования в R, прежде чем приступить к сложным вещам в виде добавления приложению реактивности:

```

server <- function(input, output, session) {
  selected <- reactive(injuries %>% filter(prod_code == input$code))

  output$diag <- renderTable(
    selected() %>% count(diag, wt = weight, sort = TRUE)
  )
  output$body_part <- renderTable(
    selected() %>% count(body_part, wt = weight, sort = TRUE)
  )
  output$location <- renderTable(
    selected() %>% count(location, wt = weight, sort = TRUE)
  )

  summary <- reactive({

```

```

    selected() %>%
      count(age, sex, wt = weight) %>%
      left_join(population, by = c("age", "sex")) %>%
      mutate(rate = n / population * 1e4)
  })

output$age_sex <- renderPlot({
  summary() %>%
    ggplot(aes(age, n, colour = sex)) +
    geom_line() +
    labs(y = "Estimated number of injuries")
}, res = 96)
}

```

Обратите внимание, что реактивное выражение `summary` мы здесь могли и не создавать, поскольку это вычисление используется лишь одним потребителем. В любом случае хорошей практикой является разделение блоков с расчетами и с построением графиков – это делает приложение более легким для восприятия и дальнейшего внедрения.

Результат работы приложения показан на рис. 4.3. Исходный код можно скачать на GitHub по адресу <https://github.com/hadley/mastering-shiny/blob/master/neiss/prototype.R>.

ДОРАБОТКА ТАБЛИЦ

Теперь, когда все базовые компоненты приложения на месте, мы можем постепенно начать его улучшать. Первое, что бросается в глаза в нашем приложении, – это избыточная информация в таблицах, тогда как нам интересны лишь выжимки из них. Чтобы решить эту проблему, нужно сначала подумать, как сократить таблицы. Я решил использовать для этого комбинацию функций из пакета *forcats*: переведу переменную в фактор, отсортирую по частоте уровней и объединю вместе элементы за границами первых пяти элементов:

```

injuries %>%
  mutate(diag = fct_lump(fct_infreq(diag), n = 5)) %>%
  group_by(diag) %>%
  summarise(n = as.integer(sum(weight)))

#> # A tibble: 6 x 2
#>   diag          n
#>   <fct>      <int>
#> 1 Other Or Not Stated 1806436
#> 2 Fracture          1558961
#> 3 Laceration         1432407
#> 4 Strain, Sprain      1432556
#> 5 Contusion Or Abrasion 1451987
#> 6 Other              1929147

```


Product

knives, not elsewhere classified

diag	n
Laceration	287925.65
Avulsion	9970.11
Puncture	5870.19
Other Or Not Stated	4852.37
Contusion Or Abrasion	1687.99
Amputation	1360.71
Fracture	574.87
Foreign Body	325.72
Strain, Sprain	226.77
Hemorrhage	179.56
Poisoning	110.03
Hematoma	99.58
Dermat Or Conj	95.36
Dental Injury	79.17
Ingested Object	79.17
Nerve Damage	79.17
Burns, Thermal	38.74
Burns, Chemical	16.18
Inter Organ Injury	16.18

body_part	n
Finger	212292.96
Hand	59287.40
Lower Arm	10986.55
Wrist	5512.13
Lower Leg	5156.99
Upper Leg	4931.13
Foot	4792.77
Knee	1666.74
Lower Trunk	1563.82
Face	1462.47
Toe	1228.27
Upper Arm	828.96
Ankle	750.35
Upper Trunk	644.14
Eyeball	555.77
Mouth	379.93
Neck	336.24
Head	309.10
Elbow	263.10
Shoulder	183.52
All Of Body	110.03
N.S./Unk	94.66
Ear	88.69
Public Region	82.66
Internal	79.17

location	n
Home	214092.90
Unknown	93006.86
Sports Or Recreation Place	2841.22
Other Public Property	2307.34
School	1067.87
Street Or Highway	254.34
Farm	16.99

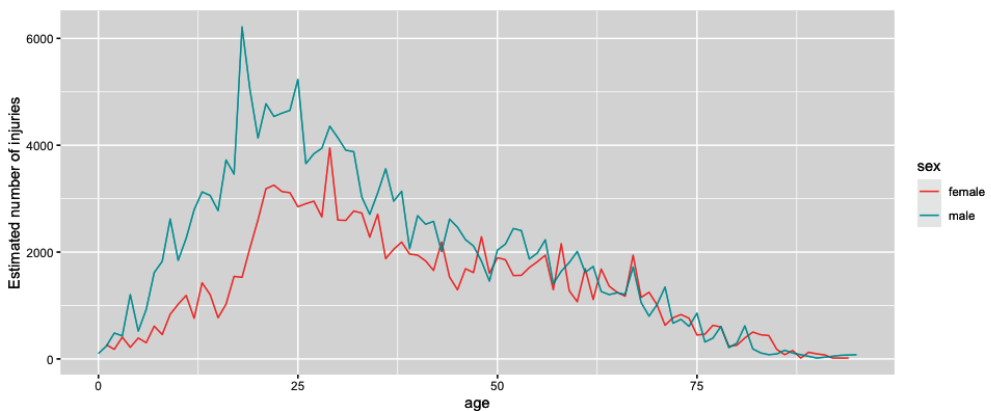


Рис. 4.3 ❖ Первый прототип нашего исследовательского приложения.
Поработать с приложением онлайн можно по адресу
<https://hadley.shinyapps.io/ms-prototype>

Написав этот фрагмент кода для конкретного случая, можно вынести его в функцию для автоматизации процесса применительно к любой переменной. Мы пока опустим все тонкости создания функций, а подробно об этом поговорим в главе 12. Если этот код вас сильно пугает, можете обойтись простым копированием и вставкой предыдущего фрагмента.

```
count_top <- function(df, var, n = 5) {
  df %>%
    mutate({{ var }} := fct_lump(fct_infreq({{ var }}), n = n)) %>%
    group_by({{ var }}) %>%
    summarise(n = as.integer(sum(weight)))
}
```

Теперь мы можем вызывать эту функцию при необходимости внутри функции `server`:

```
output$diag <- renderTable(count_top(selected(), diag), width = "100%")
output$body_part <- renderTable(count_top(selected(), body_part), width = "100%")
output$location <- renderTable(count_top(selected(), location), width = "100%")
```

Попутно я внес еще одно улучшение в плане эстетики, указав таблицам занимать всю доступную ширину, т. е. колонку, в которой они располагаются. Это положительно сказалось на внешнем виде приложения – уменьшились хаотичные разрывы между таблицами.

Итоговый вид приложения показан на рис. 4.4. Исходный код вы можете скачать на GitHub по адресу <https://github.com/hadley/mastering-shiny/blob/master/neiss/polish-tables.R>.

ПРОЦЕНТ ПРОТИВ КОЛИЧЕСТВА

До этого мы выводили в нашем приложении только один график с абсолютными числами. Но иногда бывает полезно дать пользователю самому выбрать, что выводить на графике: количество травм или проценты от общего населения. Для начала добавим в пользовательский интерфейс соответствующий элемент управления. Пусть это будет `selectInput()` – это логичный вариант, и в будущем будет легко добавлять новые выборы:

```
fluidRow(
  column(8,
    selectInput("code", "Product",
      choices = setNames(products$prod_code, products$title),
      width = "100%"
    )
  ),
  column(2, selectInput("y", "Y axis", c("rate", "count")))
),
```

Product

knives, not elsewhere classified

diag	n	body_part	n	location	n
Laceration	287925	Finger	212292	Home	214092
Avulsion	9970	Hand	59287	Unknown	93006
Puncture	5870	Lower Arm	10986	Other Public Property	2307
Other Or Not Stated	4852	Wrist	5512	Sports Or Recreation Place	2841
Contusion Or Abrasion	1687	Foot	4792	School	1067
Other	3281	Other	20715	Other	271

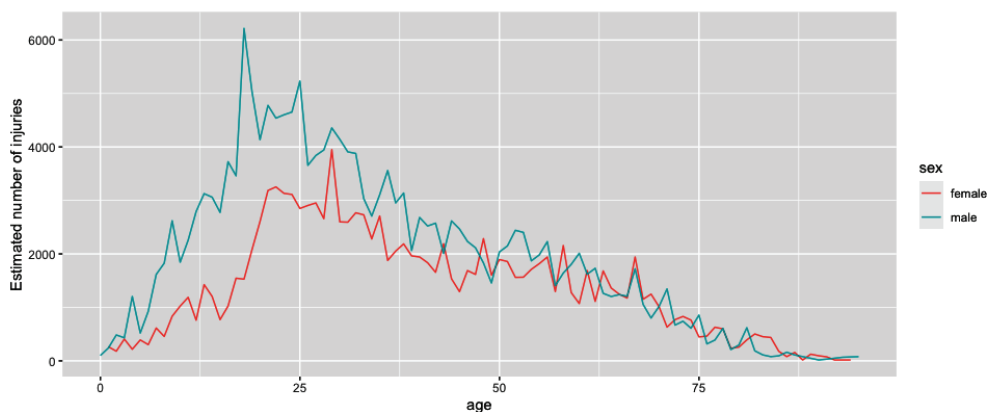


Рис. 4.4 ❖ Во второй итерации приложения мы оставили только первые несколько элементов в таблицах. Опробовать приложение на этой стадии можно по адресу <https://hadley.shinyapps.io/ms-polish-tables>

По умолчанию я установил выбор показа процентов (rate) – так будет лучше, поскольку вам не нужно будет знать распределение численности населения, чтобы правильно интерпретировать график.

После этого необходимо скорректировать вывод графика в зависимости от выбранного варианта:

```
output$age_sex <- renderPlot({
  if (input$y == "count") {
    summary() %>%
      ggplot(aes(age, n, colour = sex)) +
      geom_line() +
      labs(y = "Estimated number of injuries")
  } else {
    summary() %>%
      ggplot(aes(age, rate, colour = sex)) +
      geom_line(na.rm = TRUE) +
      labs(y = "Injuries per 10,000 people")
  }
}, res = 96)
```

Внешний вид очередной итерации приложения показан на рис. 4.5. Исходный код лежит на GitHub по адресу <https://github.com/hadley/mastering-shiny/blob/master/neiss/rate-vs-count.R>.



Рис. 4.5 ❖ На этой итерации пользователь может сам выбрать шкалу отображения оси Y. Проверить работу данного приложения онлайн можно по адресу <https://hadley.shinyapps.io/ms-rate-vs-count>

ИСТОРИИ ПОЛУЧЕНИЯ ТРАВМЫ

Наконец, давайте где-то выведем краткие истории, сопутствующие получению травм, ведь они представляют немалый интерес и позволяют проверить догадки, которые можно построить, исходя из графика. В коде R я выбирал несколько историй случайным образом, но в приложении нет никакой необходимости так делать – здесь можно перебирать истории по нажатию на кнопку.

Наше решение будет состоять из двух частей. Сначала давайте добавим строку в нижнюю часть экрана приложения. В эту строку поместим кнопку для получения очередной истории и текстовый элемент вывода для ее отображения:

```
fluidRow(
  column(2, actionButton("story", "Tell me a story")),
  column(10, textOutput("narrative"))
)
```

В таком виде приложение показано на рис. 4.6, а его исходный код можно скачать с GitHub по адресу <https://github.com/hadley/mastering-shiny/blob/master/neiss/narrative.R>.

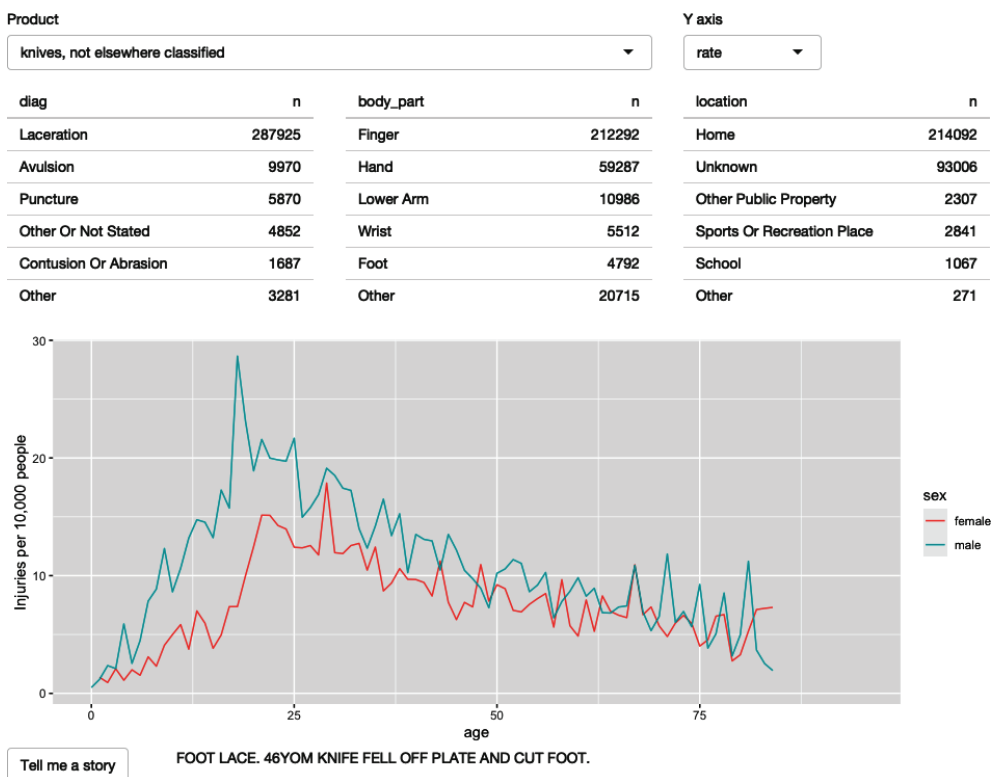


Рис. 4.6 ❖ В заключительной итерации приложения мы дали пользователю возможность просматривать истории получения травм. Посмотреть финальное приложение в интернете можно по адресу <https://hadley.shinyapps.io/ms-narrative>

После этого используем функцию `eventReactive()` для создания реактивного выражения, которое будет обновляться только по нажатии на кнопку или в результате изменения данных, лежащих в его основе:

```
narrative_sample <- eventReactive(
  list(input$story, selected()),
  selected() %>% pull(narrative) %>% sample(1)
)
output$narrative <- renderText(narrative_sample())
```

УПРАЖНЕНИЯ

Упражнение 1

Нарисуйте реактивные графики для всех приложений.

Упражнение 2

Что случится, если поменять местами функции `fct_infreq()` и `fct_lump()` в коде формирования таблиц?

Упражнение 3

Добавьте в приложение элемент ввода, позволяющий пользователю выбрать количество отображаемых строк в таблицах.

Упражнение 4

Реализуйте метод циклического переключения по историям получения травм с помощью кнопок движения вперед и назад. При нахождении на последней истории кнопка движения вперед должна перебрасывать вас на первую запись.

ЗАКЛЮЧЕНИЕ

Теперь, когда вы усвоили структурные основы фреймворка Shiny, пришло время переходить к изучению разных полезных техник, чему и будут посвящены следующие семь глав. После того как вы прочитаете следующую главу по организации рабочего процесса, я бы рекомендовал вам бегло просмотреть содержание оставшихся глав, чтобы понимать, что в них описывается, а читать их можете по мере необходимости того или иного инструмента или метода.

Часть II

SHINY В ДЕЙСТВИИ

В следующих главах вы познакомитесь со множеством полезных техник. Но начнем мы с главы 5, в которой представим важные инструменты для разработки и отладки приложений, – они помогут вам, когда почувствуете, что сбились с пути. Следующие главы в этой части будут очень слабо коррелировать между собой. В связи с этим вы можете быстро пролистать их, чтобы сразу найти то, что вам нужно, когда понадобится. Ниже приведено краткое описание глав:

- в главе 6 мы посмотрим, как можно компоновать элементы ввода и вывода на странице приложения, а также научимся настраивать их внешний вид при помощи тем;
- глава 7 будет посвящена добавлению интерактивных взаимодействий на графики и выводу изображений;
- в главе 8 мы опишем техники взаимодействия с пользователем во время работы приложения, включая оповещения об ошибках, индикаторы хода выполнения задач, диалоговые окна и т. д.;
- глава 9 будет посвящена загрузке и сохранению файлов из приложения;
- в главе 10 вы научитесь динамически менять внешний вид пользовательского интерфейса приложения прямо во время его работы;
- в главе 11 мы поговорим о сохранении текущего состояния приложений, чтобы пользователи могли делать закладки;
- в главе 12 вы узнаете, как позволить пользователям выбирать переменные при работе с пакетами `tidyverse`.

Давайте начнем с настройки рабочего процесса для разработки приложений.

Глава 5

Рабочий процесс

Если вы собираетесь связать свою профессиональную деятельность с написанием приложений Shiny (а раз вы читаете эту книгу, я надеюсь, это так и есть), вам придется потратить какое-то время на освоение *рабочего процесса* (workflow). Но это стоит того, и в итоге потраченное время окупится с лихвой. В итоге вы не просто будете писать больше кода на R, а станете гораздо лучше понимать все происходящее и быстрее получать результаты, что сделает процесс написания приложений Shiny более приятным и поможет быстрее развить нужные для этого навыки.

Цель этой главы – помочь вам в полной мере освоить три важных рабочих процесса, необходимых для разработки приложений Shiny:

- базовый цикл *разработки* (development cycle) приложения, внесение изменений и эксперименты с результатами;
- *отладка* (debugging) – рабочий процесс, связанный с попытками выяснить, что не так с вашим кодом, и принятием мер по исправлению ситуации;
- написание *воспроизводимых примеров* (reprex) – автономных блоков кода, иллюстрирующих проблему. Создание таких примеров представляет собой очень мощную технику отладки приложений, и это бывает довольно полезно, когда вы хотите получить помощь от других разработчиков.

РАБОЧИЙ ПРОЦЕСС РАЗРАБОТКИ ПРИЛОЖЕНИЯ

Целью оптимизации процесса разработки приложения является уменьшение времени между внесением корректировки и просмотром результата изменения. Чем быстрее вы сможете выполнять эти итерации, тем больше сможете экспериментировать и тем быстрее будете прогрессировать как разработчик. В области разработки приложения есть два рабочих процесса, требующих оптимизации: создание приложения с нуля и ускорение выполнения итеративных циклов по внесению изменений в код и просмотру результатов.

Создание приложения

Каждое новое приложение вы будете начинать одними и теми же строками кода:

```
library(shiny)

ui <- fluidPage(
)

server <- function(input, output, session) {
}

shinyApp(ui, server)
```

Скорее всего, очень скоро вам надоест писать этот шаблонный блок раз за разом, и в RStudio предусмотрена пара инструментов на этот случай:

- если ваш будущий файл приложения с именем *app.R* уже открыт, просто напишите слово **shinyapp**, после чего нажмите сочетание клавиш **Shift+Tab** для вставки *снимка* (snippet)¹ с заготовкой приложения Shiny;
- если вы хотите начать новый *проект* (project)², перейдите в меню **File**, выберите пункт **New Project** и в открывшемся диалоговом окне найдите вариант **Shiny Web Application**, как показано на рис. 5.1.

Вам может показаться, что изучать все эти премудрости будет излишне, ведь вы будете создавать по одному-два приложения в день. Но создавать небольшие приложения бывает очень полезно для освоения всех нюансов работы перед началом разработки действительно серьезного проекта. Кроме того, они могут стать хорошим подспорьем при отладке ваших приложений.

Отслеживание изменений

Вы будете *создавать* не так много приложений в день, но *запускать* их будете сотни раз, так что вам необходимо очень хорошо усвоить рабочий процесс разработки. Первое, что можно сделать, чтобы снизить время, требуемое для проверки новых итераций приложения, – это постараться избежать бесконечного нажатия на кнопку **Run App**, а вместо этого освоить комбинацию клавиш **Cmd/Ctrl+Shift+Enter**. Таким образом, ваш рабочий процесс может выглядеть следующим образом.

¹ Сниметы представляют собой текстовые макросы, которые можно использовать для быстрой вставки блока кода. Если вы являетесь поклонником использования этой технологии, то можете установить себе целую коллекцию специфических для Shiny сниметов от ThinkR по адресу <https://github.com/ThinkR-open/shinysnippets>.

² Под проектом подразумевается обособленная директория, изолированная от других проектов. Если вы применяете в работе RStudio, но пока не используете проекты, я очень рекомендую вам почитать о работе, ориентированной на проекты, по адресу <https://rstats.wtf/project-oriented-workflow.html>.

1. Пишете некоторый код.
2. Запускаете приложение при помощи комбинации клавиш **Cmd/Ctrl+Shift+Enter**.
3. Проверяете функционал приложения.
4. Закрываете приложение.
5. Переходите к пункту 1.

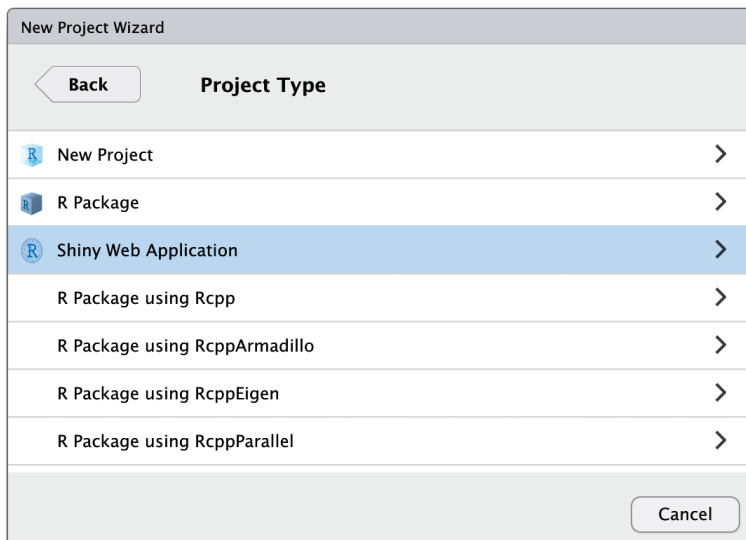


Рис. 5.1 ❖ Для создания приложения Shiny в RStudio выберите пункт Shiny Web Application

Еще одним способом ускорить итерации разработки приложения является включение *автоматической перезагрузки* (autoreload) и запуск приложения в *фоновой задаче* (background job). В этом случае после каждого сохранения файла ваше приложение будет перезапускаться без необходимости закрывать и открывать его заново. В результате рабочий процесс можно упростить до следующего.

1. Пишете некоторый код и нажимаете сочетание клавиш **Cmd/Ctrl+S** для сохранения файла.
2. Проверяете функционал приложения.
3. Переходите к пункту 1.

Главным недостатком такого подхода является относительная сложность отладки, поскольку приложение запускается в отдельном процессе.

С ростом сложности приложения вы будете понимать, что шаг с проверкой его функционала становится все более тяжелым и обременительным. Бывает непросто запомнить и проверить все изменения, которые вы произвели между итерациями. Позже, в главе 21, вы познакомитесь с инструментами для автоматического тестирования, позволяющими автоматизировать процесс проверки приложения. В результате вы сможете гораздо быстрее осу-

ществлять проверку изменений и не пропустите ни одного важного теста. Конечно, на разработку таких автоматизированных тестов придется потратить немного времени, но при разработке больших приложений эти временные затраты окупятся с лихвой.

Управление запуском приложения

По умолчанию при запуске приложения оно будет открываться во всплывающем окне. Также вы можете выбрать две другие опции показа приложения, нажав на выпадающую кнопку **Run App**, как показано на рис. 5.2:

- **Run in Viewer Pane** – в этом случае приложение будет открываться в панели **Viewer**, которая обычно располагается в среде разработки справа. Такой вариант может быть полезен при разработке небольших приложений, поскольку вы можете одновременно редактировать код и видеть результат;
- **Run External** – так вы сможете открыть свое приложение в браузере. Это может быть удобно при разработке больших приложений, а также если вам необходимо узнать, как будет выглядеть приложение у большинства пользователей.

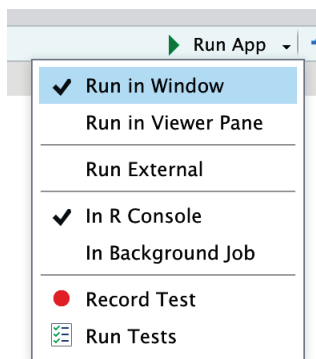


Рис. 5.2 ❖ Кнопка **Run App** позволяет выбирать, в каком виде будет запускаться ваше приложение

Отладка

В любом приложении, которое вы будете разрабатывать, будут возникать ошибки. Причиной возникновения большинства ошибок будет несоответствие между тем, как вы представляете себе устройство Shiny, и тем, как этот фреймворк на самом деле работает. В процессе чтения этой книги вы будете все больше узнавать о Shiny, так что количество ваших ошибок будет постепенно сокращаться, как и время на их поиск. К сожалению, как и в случае с любым другим языком программирования, здесь вам придется нарабаты-

вать опыт годами, чтобы писать приложения сразу практически безошибочно. Все это означает, что вам необходимо выработать четкий рабочий процесс для идентификации и исправления ошибок. В данном разделе мы будем говорить об *отладке* (debugging) применительно к фреймворку Shiny. Если у вас нет опыта отладки в R, я бы рекомендовал для начала посмотреть выступление Дженни Брайан (Jenny Bryan) на конференции *rstudio::conf(2020)* по адресу <https://www.rstudio.com/resources/rstudioconf-2020/object-of-type-closure-is-not-subsettable>.

Итак, мы главным образом будем говорить о трех следующих ситуациях:

- у вас возникла неожиданная ошибка. Это самый простой случай, поскольку в вашем распоряжении будет *трассировка* (traceback), которая поможет отследить причину возникновения ошибки. После обнаружения ошибки вам необходимо систематически проверять все предположения, пока не возникнет различие между ожиданием и реальностью. Интерактивный отладчик – прекрасный помощник в этом процессе;
- ошибки не возникают, но некоторые значения получаются неправильные. В этом случае вам придется использовать интерактивный отладчик и свои аналитические навыки для поиска первопричины появления этой неточности;
- все значения корректные, но они не обновляются так, как предполагалось. Это наиболее сложный случай, поскольку он может возникать только при работе с Shiny, и своими навыками отладки в R вы воспользоваться не сможете.

Всегда неприятно, когда в программе возникают ошибки, но без них вы бы не смогли развить свои навыки в отладке.

В следующем разделе мы поговорим о еще одной важной технике создания минимальных воспроизводимых примеров. Это бывает полезно, когда вы окончательно запутались и хотите, чтобы вам помогли в решении задачи. В то же время создание минимальных примеров может быть исключительно полезно и при отладке своего собственного кода. Чаще всего большая часть кода в программе работает нормально, а ошибки возникают только в определенной части программы. Если вам удастся выделить проблемный участок кода, вы сможете быстрее справиться с задачей путем запуска многократных итераций. Лично я использую эту технику практически каждый день.

Чтение трассировки

В языке R каждой ошибке соответствует своя *трассировка* (traceback) или *стек вызовов* (call stack), позволяющий пройти по вызовам в обратном направлении для поиска точки, в которой возникла ошибка. Представьте себе такую последовательность вызовов функций: `f()` вызывает `g()`, которая вызывает `h()`, а в ней выполняется операция умножения:

```
f <- function(x) g(x)
g <- function(x) h(x)
h <- function(x) x * 2
```

Если при вызове функции `f()` возникнет ошибка, как показано ниже:

```
f("a")
#> Error in x * 2: non-numeric argument to binary operator
```

вы сможете вызвать функцию `traceback()`, чтобы по сформированному стеку попытаться определить место возникновения ошибки:

```
traceback()
#> 3: h(x)
#> 2: g(x)
#> 1: f("a")
```

Мне кажется, эту трассировку легче будет понять, если развернуть ее в обратном направлении, как показано ниже:

```
1: f("a")
2: g(x)
3: h(x)
```

Таким образом, можно видеть, где именно возникла ошибка в стеке вызовов: функция `f()` вызвала функцию `g()`, которая вызвала функцию `h()`, и в ней уже возникла ошибка.

Трассировка в Shiny

К сожалению, вам не удастся воспользоваться функцией `traceback()` в Shiny, поскольку вы не можете выполнять код при запущенном приложении. Вместо этого Shiny будет автоматически выводить в консоль трассировку. Давайте для примера создадим простое приложение с вызовом указанной выше функции `f()`:

```
library(shiny)

f <- function(x) g(x)
g <- function(x) h(x)
h <- function(x) x * 2

ui <- fluidPage(
  selectInput("n", "N", 1:10),
  plotOutput("plot")
)

server <- function(input, output, session) {
  output$plot <- renderPlot({
    n <- f(input$n)
    plot(head(cars, n))
  }, res = 96)
}

shinyApp(ui, server)
```

При запуске этого приложения вы увидите ошибку и следующую трассировку в консоли:

```
Error in *: non-numeric argument to binary operator
169: g [app.R#4]
168: f [app.R#3]
167: renderPlot [app.R#13]
165: func
125: drawPlot
111: <reactive:plotObj>
95: drawReactive
82: renderFunc
81: output$plot
1: runApp
```

Давайте для удобства снова перевернем стек вызовов, чтобы они выстроились в хронологическом порядке:

```
Error in *: non-numeric argument to binary operator
1: runApp
81: output$plot
82: renderFunc
95: drawReactive
111: <reactive:plotObj>
125: drawPlot
165: func
167: renderPlot [app.R#13]
168: f [app.R#3]
169: g [app.R#4]
```

В данном стеке вызовов можно выделить три важных раздела:

- в первых нескольких вызовах осуществляется запуск приложения. В нашем случае вы видите только строку вызова функции `runApp()`, но в зависимости от того, как вы запускаете приложение, вы можете встретиться и с более замысловатыми комбинациями вызовов. Например, если вы используете функцию `source()` для запуска приложения, вы можете увидеть следующую последовательность в начале трассировки:

```
1: source
3: print.shiny.appobj
5: runApp
```

Чаще всего вы можете игнорировать все вызовы вплоть до функции `runApp()`, поскольку здесь просто производится запуск приложения;

- в следующих нескольких строках трассировки вы увидите код Shiny, отвечающий за создание реактивных выражений:

```
81: output$plot
82: renderFunc
95: drawReactive
```

```
111: <reactive:plotObj>
125: drawPlot
165: func
```

Здесь важно обратить внимание на вызов `output$plot`, поскольку он указывает на реактив (`plot`), ставший причиной возникновения ошибки. Следующие функции являются внутренними, и вы можете просто проигнорировать их;

- наконец, в конце трассировки вы можете увидеть написанный вами код:

```
167: renderPlot [app.R#13]
168: f [app.R#3]
169: g [app.R#4]
```

Этот код был вызван внутри функции `renderPlot()`. Вы должны обратить внимание на эти строки в трассировке еще и потому, что в них указан путь к файлу и номер строки, – так вы можете легко понять, что это ваш код.

Если в вашем приложении возникает ошибка, а трассировка не отображается, убедитесь, что вы запустили приложение с использованием комбинации клавиш **Cmd/Ctrl+Shift+Enter** (а если работаете не в RStudio, то при помощи функции **runApp()**) и сохранили файл, из которого его запускаете. При других способах запуска приложения не всегда происходит захват информации, необходимой для вывода трассировки.

Интерактивный отладчик

Определив источник ошибки, вы наверняка заинтересуетесь тем, что именно привело к ее появлению. И в этом вам, скорее всего, поможет очень мощный инструмент под названием *интерактивный отладчик* (interactive debugger). Во время запуска отладчика приложение ставится на паузу и открывается интерактивная консоль R, в которой вы можете выполнять любой код, способный помочь вам понять, что случилось с приложением. Запустить отладчик можно двумя способами:

- добавить вызов функции `browser()` в исходный код. Это стандартный способ запуска интерактивного отладчика в R, и он работает и применительно к приложениям Shiny. Еще одним преимуществом вызова функции `browser()` является то, что она представляет собой код на языке R, а значит, может быть использована в условных выражениях с применением конструкции `if`. Это позволяет запускать отладчик только для проблемных элементов ввода:

```
if (input$value == "a") {
  browser()
}
# Или так
if (my_reactive() < 0) {
```

```

    browser()
  }

```

- добавить *точку останова* (breakpoint) программы в RStudio, щелкнув мышью слева от номера строки. Удалить точку останова можно, щелкнув на появившемся красном кружке, как показано на рис. 5.3. Преимуществом точек останова является то, что они не внедряются в код программы, а значит, нет необходимости беспокоиться об их случайном сохранении в *системе контроля версий* (version control system).

```

23 ▾ server <- function(input, output, session) {
24 ▾   territory <- reactive({
25   req(input$territory)
26 ▾   if (input$territory == "NA") {

```

Рис. 5.3 ❖ Точка останова

Если вы работаете в RStudio, при запуске отладчика в верхней части консоли появится панель инструментов, показанная на рис. 5.4. При помощи панели инструментов можно легко запомнить команды отладки, доступные вам в данный момент. Эти команды доступны и за пределами RStudio: достаточно запомнить их односимвольные обозначения для запуска. Ниже приведены три наиболее популярные команды отладки:

- **Next** (нажмите на клавишу **n**) – переход на следующий шаг в функции. Обратите внимание, что если у вас есть переменная с именем *n*, вам необходимо использовать инструкцию `print(n)` для вывода ее значения;
- **Continue** (нажмите на клавишу **c**) – завершение интерактивной отладки и переход к нормальному выполнению функции. Эта команда бывает полезна, когда вы внесли необходимые изменения и хотите проверить, что функция отработает корректно;
- **Stop** (нажмите на клавишу **q**) – остановка отладки, прекращение выполнения функции и возврат в общее рабочее пространство. Используйте эту команду, если точно определили место возникновения ошибки, готовы исправить ее и запустить приложение заново.



Рис. 5.4 ❖ Панель инструментов отладки в RStudio

Вместе с перемещением по коду при помощи этих инструментов отладки вы также можете писать вспомогательные интерактивные фрагменты кода для отслеживания происходящего в приложении. Процесс отладки заключается в систематическом сравнении ожиданий с реальностью до тех пор, пока не будет обнаружено расхождение. Если вы новичок в R, возможно, вам будет полезно прочитать главу, посвященную отладке, из книги *Advanced R* по адре-

су <https://adv-r.hadley.nz/debugging.html#debugging-strategy>. Так вы сможете быстрее освоить базовые принципы отладки.

Практический пример

Отбросьте все невозможное, и то, что останется, и будет ответом, каким бы невероятным он ни казался.

Шерлок Холмс, «Знак четырех», Артур Конан Дойл

Для демонстрации базовых принципов отладки давайте рассмотрим проблему, с которой я столкнулся при написании приложения с иерархическими списками из главы 10. Сначала я покажу вам базовый проект и проблему, которую я решил без интерактивного отладчика, хотя она его требовала. Ну и посмотрите, что из этого вышло.

Изначальная цель была проста – есть набор данных с продажами, и мне нужно отфильтровать его по географическому признаку. Вот как выглядят исходные данные:

```
sales <- readr::read_csv("sales-dashboard/sales_data_sample.csv")
sales <- sales[c(
  "TERRITORY", "ORDERDATE", "ORDERNUMBER", "PRODUCTCODE",
  "QUANTITYORDERED", "PRICEEACH"
)]
sales

#> # A tibble: 2,823 x 6
#>   TERRITORY ORDERDATE   ORDERNUMBER PRODUCTCODE QUANTITYORDERED PRICEEACH
#>   <chr>      <chr>          <dbl> <chr>          <dbl>      <dbl>
#> 1 <NA>      2/24/2003 0:00      10107 S10_1678         30        95.7
#> 2 EMEA      5/7/2003 0:00      10121 S10_1678         34        81.4
#> 3 EMEA      7/1/2003 0:00      10134 S10_1678         41        94.7
#> 4 <NA>      8/25/2003 0:00      10145 S10_1678         45        83.3
#> # ... with 2,819 more rows
```

А вот и территории:

```
unique(sales$TERRITORY)
#> [1] NA      "EMEA" "APAC" "Japan"
```

Когда я только начал работать над этой задачей, мне казалось, что она довольно простая и я смогу сразу написать приложение без проведения дополнительных исследований:

```
ui <- fluidPage(
  selectInput("territory", "territory", choices = unique(sales$TERRITORY)),
  tableOutput("selected")
)

server <- function(input, output, session) {
  selected <- reactive(sales[sales$TERRITORY == input$territory, ])
```

```
output$selected <- renderTable(head(selected(), 10))
}
```

Я подумал: *это простое приложение из восьми строк, откуда здесь взяться ошибкам?* Но когда я запустил приложение, то обнаружил массу пропущенных значений, какую бы территорию я ни выбрал. Скорее всего, источник ошибки мог скрываться в реактивном выражении, выбирающем данные для показа: `sales[sales$TERRITORY == input$territory,]`. Я закрыл приложение и быстро проверил, что подмножество работает так, как я и предполагал:

```
sales[sales$TERRITORY == "EMEA", ]
#> # A tibble: 2,481 x 6
#>   TERRITORY ORDERDATE      ORDERNUMBER PRODUCTCODE QUANTITYORDERED PRICEEACH
#>   <chr>      <chr>          <dbl> <chr>          <dbl>      <dbl>
#> 1 <NA>      <NA>                NA <NA>              NA         NA
#> 2 EMEA      5/7/2003 0:00        10121 S10_1678      34         81.4
#> 3 EMEA      7/1/2003 0:00        10134 S10_1678      41         94.7
#> 4 <NA>      <NA>                NA <NA>              NA         NA
#> # ... with 2,477 more rows
```

Вот это да! Я забыл, что в поле `TERRITORY` содержатся пустые значения, а значит, выражение `sales$TERRITORY == "EMEA"` также выдаст ряд пропущенных значений:

```
head(sales$TERRITORY == "EMEA", 25)
#> [1] NA TRUE TRUE NA NA NA TRUE TRUE NA TRUE FALSE NA
#> [13] NA NA TRUE NA TRUE TRUE NA NA TRUE FALSE TRUE NA
#> [25] TRUE
```

Эти пропущенные значения превращаются в пропущенные строки, и когда я использую их для ограничения датафрейма `sales` с использованием квадратных скобок, все отсутствующие значения на входе сохраняются и на выходе. Есть множество способов решить эту проблему. Я решил использовать функцию `subset()`¹, поскольку она автоматически удаляет пропущенные значения, и мне не придется много раз писать слово `sales`. Дважды проверим, что все работает:

```
subset(sales, TERRITORY == "EMEA")
#> # A tibble: 1,407 x 6
#>   TERRITORY ORDERDATE      ORDERNUMBER PRODUCTCODE QUANTITYORDERED PRICEEACH
#>   <chr>      <chr>          <dbl> <chr>          <dbl>      <dbl>
#> 1 EMEA      5/7/2003 0:00        10121 S10_1678      34         81.4
#> 2 EMEA      7/1/2003 0:00        10134 S10_1678      41         94.7
#> 3 EMEA     11/11/2003 0:00        10180 S10_1678      29         86.1
#> 4 EMEA     11/18/2003 0:00        10188 S10_1678      48         100
#> # ... with 1,403 more rows
```

¹ Я использую функцию `subset()`, чтобы моему приложению не нужны были другие пакеты. В большом приложении я наверняка применил бы функцию `dplyr::filter()`, просто потому что я лучше знаком с ее поведением.

Это позволило мне решить большинство проблем, но у меня по-прежнему осталась неувязочка с выбором значения NA в выпадающем списке с территориями: строки не появлялись. Я снова проверил в консоли:

```
subset(sales, TERRITORY == NA)
#> # A tibble: 0 x 6
#> # ... with 6 variables: TERRITORY <chr>, ORDERDATE <chr>, ORDERNUMBER <dbl>,
#> #   PRODUCTCODE <chr>, QUANTITYORDERED <dbl>, PRICEEACH <dbl>
```

Тогда я вспомнил, что это, конечно же, не сработает, поскольку пропущенные значения заразные:

```
head(sales$TERRITORY == NA, 25)
#> [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
```

Есть еще один трюк, который позволит решить эту проблему, – можно перейти от использования `==` к `%in%`:

```
head(sales$TERRITORY %in% NA, 25)
#> [1] TRUE FALSE FALSE TRUE TRUE TRUE FALSE FALSE TRUE FALSE FALSE TRUE
#> [13] TRUE TRUE FALSE TRUE FALSE FALSE TRUE TRUE FALSE FALSE FALSE TRUE
#> [25] FALSE
subset(sales, TERRITORY %in% NA)
#> # A tibble: 1,074 x 6
#>   TERRITORY ORDERDATE ORDERNUMBER PRODUCTCODE QUANTITYORDERED PRICEEACH
#>   <chr>      <chr>          <dbl> <chr>                <dbl>    <dbl>
#> 1 <NA>      2/24/2003 0:00      10107 S10_1678          30      95.7
#> 2 <NA>      8/25/2003 0:00      10145 S10_1678          45      83.3
#> 3 <NA>     10/10/2003 0:00      10159 S10_1678          49      100
#> 4 <NA>     10/28/2003 0:00      10168 S10_1678          36      96.7
#> # ... with 1,070 more rows
```

Итак, я обновил приложение и запустил его снова. Но оно по-прежнему не работало! Когда я выбирал в выпадающем списке вариант NA, ни одна строка не выводилась.

К тому моменту я решил, что перепробовал все возможные способы решения проблемы с использованием консоли, осталось поэкспериментировать и понять, почему внутри Shiny код работает не так, как я ожидал. Я предполагал, что наиболее вероятным источником проблемы будет реактивное выражение `selected`, так что поместил в него вызов функции `browser()`. При этом выражение стало многострочным, поэтому пришлось заключить его в фигурные скобки:

```
server <- function(input, output, session) {
  selected <- reactive({
    browser()
    subset(sales, TERRITORY %in% input$territory)
  })
  output$selected <- renderTable(head(selected(), 10))
}
```

После запуска приложения передо мной сразу открылась интерактивная консоль. Первое, что я решил проверить, – это само наличие проблемы, поэтому я ввел следующую строку: `subset(sales, TERRITORY %in% input$territory)`. На выходе я получил пустой датафрейм, что подтвердило присутствие проблемы. Если бы проблема не возникла, я бы нажал на клавишу `c`, чтобы приложение продолжило работу, после чего постарался бы воссоздать ошибку заново, взаимодействуя с приложением.

После этого я проверил, что с наборами данных, поступающими в `subset()`, все в порядке. Сначала я проверил набор `sales`. Не то чтобы я сомневался в его целостности, поскольку этот набор данных не изменялся внутри программы, но лучше лишний раз все внимательно проверить. С `sales` все оказалось в порядке, так что проблема должна была крыться в выражении `TERRITORY %in% input$territory`. Я стал исследовать выражение `input$territory`, поскольку `TERRITORY` – это часть `sales`:

```
input$territory
#> [1] "NA"
```

Какое-то время я смотрел на экран в недоумении, так как с этим выражением тоже вроде все было в порядке. А затем меня осенило: я ожидал, что выведется значение `NA`, а вывелось `"NA"`! Теперь я мог воссоздать проблему за пределами приложения Shiny:

```
subset(sales, TERRITORY %in% "NA")
#> # A tibble: 0 x 6
#> # ... with 6 variables: TERRITORY <chr>, ORDERDATE <chr>, ORDERNUMBER <dbl>,
#> #   PRODUCTCODE <chr>, QUANTITYORDERED <dbl>, PRICEEACH <dbl>
```

Затем я написал измененный код, представленный ниже, перенес его в функцию `server` и запустил приложение снова:

```
server <- function(input, output, session) {
  selected <- reactive({
    if (input$territory == "NA") {
      subset(sales, is.na(TERRITORY))
    } else {
      subset(sales, TERRITORY == input$territory)
    }
  })
  output$selected <- renderTable(head(selected(), 10))
}
```

Ура! Проблема была решена! Но сама ситуация меня очень удивила: Shiny молча преобразовал `NA` в `"NA"`, и я немедленно создал отчет об ошибке, который можно посмотреть по адресу <https://github.com/rstudio/shiny/issues/2884>.

Несколькими неделями позже я вновь взглянул на этот пример на предмет наличия разных территорий. У нас есть территория, включающая Европу, Ближний и Средний Восток и Африку (Europe, Middle East, and Africa – EMEA), а также Азиатско-Тихоокеанский регион (Asia-Pacific – APAC). А где же Северная Америка (North America)? Тогда до меня дошло – вероятно, в исходных

данных для Северной Америки была использована аббревиатура NA, и R воспринял ее как отсутствующее значение. Так что на самом деле вносить изменения в данном случае необходимо еще на этапе загрузки данных следующим образом:

```
sales <- readr::read_csv("sales-dashboard/sales_data_sample.csv", na = "")
unique(sales$TERRITORY)
#> [1] "NA"      "EMEA"    "APAC"    "Japan"
```

Это существенно облегчит нам жизнь!

Здесь мы представили довольно типичный процесс отладки: зачастую вам приходится снимать с луковицы кожуру за кожурой, чтобы добраться до истины.

Отладка реактивных выражений

Одной из наиболее сложных ситуаций для отладки является активизация реактивных выражений в неожиданное для вас время. На данном этапе чтения книги я могу порекомендовать вам не так много средств для эффективной отладки подобных проблем. В следующем разделе вы научитесь создавать минимальные воспроизводимые примеры, хорошо подходящие для исправления таких ситуаций, а позже в этой книге узнаете о таких удобных инструментах отладки, как *реактивный лог* (reactive log). В данный же момент сосредоточимся на классической технике отладки с выводом на печать.

Техника отладки с выводом на печать заключается в вызове функции `print()` всякий раз, когда выполняется тот или иной блок кода, с показом текущих значений, важных для хода выполнения программы переменных. Мы называем эту технику *принт-отладкой*, поскольку в большинстве языков программирования для вывода на печать используется функция `print()`. В то же время в R лучше для этих целей применять функцию `message()`:

- функция `print()` предназначена для отображения векторов данных, поэтому она заключает строки в кавычки, а вывод начинается с конструкции `[1]`;
- функция `message()` посылает результат в так называемый *стандартный вывод ошибок* (standard error), а не в *стандартный вывод* (standard output). Этими техническими терминами описываются потоки вывода, которые вы обычно не замечаете, поскольку при интерактивном запуске приложения они отображаются одинаково. Но когда ваше приложение где-то развернуто, поток стандартного вывода ошибок будет записываться в логи.

Я также рекомендую сочетать функции `message()` и `glue::glue()` для вывода текста и значений в удобном формате. Если вы ранее не встречались с функцией `glue()`, ее основная идея состоит в том, что любой текст, заключенный в фигурные скобки (`{}`), будет вычисляться и вставляться в поток вывода:

```
library(glue)
name <- "Hadley"
message(glue("Hello {name}"))
#> Hello Hadley
```

Также очень полезным инструментом является функция *str()*, позволяющая вывести на экран детальную структуру любого объекта. Это бывает крайне полезно, когда вам необходимо убедиться в том, что вы работаете с объектом ожидаемого типа.

Приведем пример приложения, в котором продемонстрированы все базовые идеи. Обратите внимание на то, как я использую функцию *message()* внутри *reactive()*: мне необходимо произвести вычисление, послать сообщение и вернуть значение предыдущего вычисления:

```
ui <- fluidPage(
  sliderInput("x", "x", value = 1, min = 0, max = 10),
  sliderInput("y", "y", value = 2, min = 0, max = 10),
  sliderInput("z", "z", value = 3, min = 0, max = 10),
  textOutput("total")
)

server <- function(input, output, session) {
  observeEvent(input$x, {
    message(glue("Updating y from {input$y} to {input$x * 2}"))
    updateSliderInput(session, "y", value = input$x * 2)
  })

  total <- reactive({
    total <- input$x + input$y + input$z
    message(glue("New total is {total}"))
    total
  })

  output$total <- renderText({
    total()
  })
}
```

После запуска приложения на консоли появились следующие строки:

```
Updating y from 2 to 2
New total is 6
```

Сдвинув ползунок *x* на значение 3, я получил следующий вывод:

```
Updating y from 2 to 6
New total is 8
New total is 12
```

Не беспокойтесь, если вывод показался вам немного странным. Прочитав главу 8 и вспомнив, что читали о реактивных графиках в главе 3, вы без труда разберетесь в происходящем.

Получение помощи

Если вы, перепробовав все перечисленные техники отладки, так и не смогли решить проблему, то можете попросить помощи у более опытных коллег. На сайте *RStudio Community*, располагающемся по адресу <https://community.rstudio.com/c/shiny/8>, вы всегда сможете отыскать ответ на свой вопрос. На этом сайте обитает много разработчиков приложений Shiny, а также самого фреймворка. Кроме того, вы можете и сами помогать менее опытным посетителям сайта, тем самым развивая и свои навыки.

Для получения максимально быстрой и эффективной помощи вам необходимо создать так называемый *воспроизводимый пример* (reprex – от **re**producible **e**xample). Цель такого примера – предоставить минимальный фрагмент кода R, который может быть запущен на другом компьютере для воспроизведения возникшей ошибки. Создавать воспроизводимые примеры при обращении за помощью – это признак хорошего тона и кратчайший путь к решению вашей проблемы: если вы хотите, чтобы люди вам помогли, максимально облегчите им эту задачу.

Воспроизводимые примеры позволяют оппонентам ухватить самую суть вопроса и быстро проверить предлагаемые решения – это поможет вам быстро и эффективно выработать план по устранению проблемы.

Основы воспроизводимых примеров

Воспроизводимый пример – не что иное, как фрагмент кода на R, без проблем работающий на другом компьютере. Ниже показан простой воспроизводимый пример приложения Shiny:

```
library(shiny)

ui <- fluidPage(
  selectInput("n", "N", 1:10),
  plotOutput("plot")
)

server <- function(input, output, session) {
  output$plot <- renderPlot({
    n <- input$n * 2
    plot(head(cars, n))
  })
}

shinyApp(ui, server)
```

В представленном коде не делается никаких предположений о том, на каком компьютере запускается приложение, за исключением того, что на нем должен быть установлен фреймворк Shiny. Таким образом, любой может воспроизвести у себя возникшую проблему и попытаться разобраться с ней локально. Показанный выше код выдает следующую ошибку: `non-numeric argument to binary operator`.

Понятно донести информацию об ошибке – значит сделать первый шаг к ее исправлению, а поскольку с помощью воспроизводимого примера любой может с легкостью проверить свои идеи, приемлемое решение будет найдено достаточно быстро. В случае с приведенной задачей нам необходимо было написать `as.numeric(input$n)`, поскольку на выходе элемента `selectInput()` получается `input$n` строкового типа.

Создание воспроизводимого примера

Первым делом для получения минимального воспроизводимого примера необходимо создать отдельный файл, содержащий все необходимое для запуска вашего кода. Убедитесь, что вы не забыли загрузить пакеты¹, нужные для работы приложения.

Обычно самой большой сложностью с запуском приложения на стороннем компьютере является то, что в нем используются ваши данные. И в этом случае есть три возможных решения:

- зачастую сами данные никак не связаны с возникающей в приложении ошибкой, и вместо них вы легко можете использовать встроенные наборы данных вроде `mtcars` или `iris`;
- также вы можете написать вспомогательный код на R, воссоздающий минимальный набор данных, необходимый для воспроизведения возникшей проблемы, например:

```
mydata <- data.frame(x = 1:5, y = c("a", "b", "c", "d", "e"))
```

- если ни одна из приведенных выше техник вам не подходит, вы можете включить свои данные в код при помощи функции `dput()`. Например, инструкция `dput(mydata)` поможет вам сгенерировать код, необходимый для воссоздания данных из объекта `mydata`:

```
dput(mydata)
#> structure(list(x = 1:5, y = c("a", "b", "c", "d", "e")),
#> class = "data.frame", row.names = c(NA, -5L))
```

Получив этот код, вы можете вставить его в свой воспроизводимый пример для создания объекта `mydata`, как показано ниже:

```
mydata <- structure(list(x = 1:5, y = structure(1:5, .Label = c("a", "b", "c", "d", "e"), class = "factor")), class = "data.frame", row.names = c(NA, -5L))
```

Зачастую вызов функции `dput()` для воссоздания ваших данных может приводить к образованию кода огромного размера, и вам лучше будет найти минимальное подмножество данных, способное проиллюстри-

¹ Вне зависимости от того, как вы обычно загружаете пакеты, здесь я настоятельно рекомендую использовать множественные команды `library()`. Это поможет легче разобраться с вашим кодом тем, кто незнаком с используемыми вами инструментами.

ровать возникающую проблему. Чем меньший набор сопутствующих данных вы предоставите, тем легче будет людям решить вашу задачу.

Если прочитать данные, необходимые для воспроизведения проблемы, с диска не представляется возможным, в качестве последней меры можно предоставить людям полный проект с исходным файлом *app.R* и всеми нужными файлами с данными. Лучше всего реализовать это при помощи проекта RStudio, размещенного на GitHub. Также вы можете создать архив zip, который может быть запущен локально. Убедитесь, что вы используете относительные пути, а не абсолютные, например `read.csv("my-data.csv")`, а не `read.csv("c:\\my-user-name\\files\\my-data.csv")`, чтобы ваш код мог работать на стороннем компьютере.

Кроме того, хорошим тоном будет позаботиться о том, чтобы ваш код был хорошо отформатирован и легко читался. Если вы будете придерживаться стиля, прописанного в руководстве по *tidyverse* по адресу <https://style.tidyverse.org>, то сможете применить автоматическое форматирование при помощи пакета *styler* (<https://styler.r-lib.org>). Это значительно облегчит для понимания ваш код.

Минимальный воспроизводимый пример

Создание воспроизводимого примера – поистине первый шаг к решению возникшей проблемы, поскольку он дает возможность любому точно воссоздать вашу ошибку. В то же время проблемный фрагмент кода может за просто затеряться в глубине программы, выполняющейся нормально. Таким образом, вы должны постараться по максимуму исключить нормально работающий код из воспроизводимого примера, чтобы он не отвлекал людей, желающих вам помочь.

Создание действительно минимального воспроизводимого примера особенно важно для приложений Shiny, которые зачастую бывают достаточно сложными. Вы сможете гораздо быстрее получить квалифицированную помощь, если выделите проблемный код самостоятельно, а не будете ждать, что потенциальный помощник станет разбираться во всем вашем приложении. Кроме того, в процессе поиска проблемных участков кода вы можете и сами понять, где кроется ошибка, – в этом случае вам даже не придется дожидаться помощи от других, вы сами сможете решить возникшую проблему.

Однако выделение из целой программы блока кода с ошибкой – это настоящее искусство, и у вас может не сразу получиться это сделать идеально. Но это нормально. Даже небольшое сокращение кода может помочь человеку лучше разобраться в вашем приложении, а со временем вы научитесь все более искусно выделять минимально возможные фрагменты кода для проверки.

Если вы не знаете, какая именно часть кода приводит к возникновению ошибки, можно попробовать исключать фрагмент за фрагментом, пока ошибка не исчезнет. В этом случае с большой долей вероятности можно будет сказать, что именно последний удаленный фрагмент приводит к появлению ошибки. Также можно попробовать создать приложение заново и постепенно расширять его, пока не проявится проблема.

Завершив создание минимального воспроизводимого примера, пройдите по предложенному ниже списку и проверьте, что не забыли ничего важного:

- относятся ли к возникающей ошибке все оставшиеся в интерфейсе пользователя элементы ввода и вывода?
- если у вашего приложения сложный дизайн, все ли вы сделали, чтобы упростить его и оставить только область, относящуюся к возникновению ошибки? Не забыли ли вы убрать из приложения все украшательства, не относящиеся напрямую к возникновению ошибки?
- остались ли в функции `server()` реактивные выражения, от которых можно избавиться?
- если вы самостоятельно пытались решить проблему, убрали ли вы из кода все вспомогательные отладочные фрагменты?
- все ли пакеты, которые вы загружаете, требуются для воспроизведения ошибки? Можете ли вы убрать пакеты, заменив использующие их функции на временные заглушки?

Все перечисленное может предполагать немало работы, но вы будете щедро вознаграждены за свои усилия. Зачастую вы будете сами находить ошибку в коде в процессе его «очищения». А если не найдете, вам в этом наверняка помогут, – в сокращенном приложении людям разобраться будет куда проще.

Практический пример

В качестве эталона создания первоклассного воспроизводимого примера я приведу пост Скотта Новогораца (Scott Novogoratz) на сайте RStudio Community по адресу <https://community.rstudio.com/t/error-in-getslidertype-type-mismatch-for-min-max-and-value-each-must-be-date-posixt-or-number-dynamic-min-max-values-for-date-time-for-inputslider-r/37982>. Изначально представленный автором фрагмент кода был близок к воспроизводимому примеру, но он не мог быть использован для воссоздания проблемы по причине отсутствия пары пакетов. Что я сделал:

- добавил недостающие строки загрузки пакетов: `library(lubridate)` и `library(xts)`;
- разделил `ui` и `server` на обособленные объекты;
- переформатировал код с помощью функции `styler::style_selection()`.

В результате получился следующий воспроизводимый пример:

```
library(xts)
library(lubridate)
library(shiny)

ui <- fluidPage(
  uiOutput("interaction_slider"),
  verbatimTextOutput("breaks")
)

server <- function(input, output, session) {
  df <- data.frame(
```

```

    dateTime = c(
      "2019-08-20 16:00:00",
      "2019-08-20 16:00:01",
      "2019-08-20 16:00:02",
      "2019-08-20 16:00:03",
      "2019-08-20 16:00:04",
      "2019-08-20 16:00:05"
    ),
    var1 = c(9, 8, 11, 14, 16, 1),
    var2 = c(3, 4, 15, 12, 11, 19),
    var3 = c(2, 11, 9, 7, 14, 1)
  )

timeSeries <- as.xts(df[, 2:4],
  order.by = strptime(df[, 1], format = "%Y-%m-%d %H:%M:%S")
)
print(paste(min(time(timeSeries)), is.POSIXt(min(time(timeSeries))), sep = " "))
print(paste(max(time(timeSeries)), is.POSIXt(max(time(timeSeries))), sep = " "))

output$interaction_slider <- renderUI({
  sliderInput(
    "slider",
    "Select Range:",
    min = min(time(timeSeries)),
    max = max(time(timeSeries)),
    value = c(min, max)
  )
})

brks <- reactive({
  req(input$slider)
  seq(input$slider[1], input$slider[2], length.out = 10)
})

output$breaks <- brks
}

shinyApp(ui, server)

```

Если вы запустите этот пример, то увидите ту же ошибку, что и в исходном посте, а именно: «*Type mismatch for min, max, and value. Each must be Date, POSIXt, or number*». Это почти идеальный воспроизводимый пример – вы можете легко и просто запустить его на своем компьютере и получить искомую ошибку. В то же время он получился довольно объемным, в связи с чем не сразу понятно, что именно приводит к возникновению ошибки.

Для упрощения данного примера мы можем пройти по каждой строке кода и подумать, нужна ли она здесь. В процессе этого я обнаружил следующее:

- две строки кода, начиная с функции `print()`, никак не влияют на появление ошибки. А в этих строках вызывается функция `lubridate::is.POSIXt()`, принадлежащая пакету `lubridate`. Больше в коде нет обращения к функциям этого пакета, а значит, избавившись от этих строк, мы можем убрать и строку загрузки пакета `lubridate`;

- `df` представляет собой датафрейм, который был преобразован в датафрейм `xts` и присвоен переменной `timeSeries`. Но единственное место, где дальше используется переменная `timeSeries`, – это вызов функции `time(timeSeries)`, возвращающей тип даты и времени. Таким образом, я просто создал переменную `datetime` с датами. Ошибка сохранилась, так что я просто избавился от переменных `timeSeries` и `df`, а поскольку больше нигде пакет `xts` не используется, я убрал и строку `library(xts)`.

В результате обновленная функция `server()` получила следующий вид:

```
datetime <- Sys.time() + (86400 * 0:10)

server <- function(input, output, session) {
  output$interaction_slider <- renderUI({
    sliderInput(
      "slider",
      "Select Range:",
      min  = min(datetime),
      max  = max(datetime),
      value = c(min, max)
    )
  })

  brks <- reactive({
    req(input$slider)
    seq(input$slider[1], input$slider[2], length.out = 10)
  })

  output$breaks <- brks
}
```

Также я обратил внимание, что в данном примере используется довольно продвинутая техника Shiny с генерированием пользовательского интерфейса непосредственно в серверной функции. Но в данном случае функция `renderUI()` не использует реактивные элементы ввода, так что ничего не изменится, если мы перенесем этот блок в функцию интерфейса.

В результате мы получили очень простой пример, и ошибка теперь возникает намного раньше – еще до старта приложения:

```
ui <- fluidPage(
  sliderInput("slider",
    "Select Range:",
    min  = min(datetime),
    max  = max(datetime),
    value = c(min, max)
  ),
  verbatimTextOutput("breaks")
)
#> Error: Type mismatch for `min`, `max`, and `value`.
#> i All values must have same type: either numeric, Date, or POSIXt.
```

После этого можно внимательно присмотреться к сообщению об ошибке и вывести входные параметры для переменных `min`, `max` и `value`, чтобы понять, в чем проблема:

```
min(datetime)
#> [1] "2021-03-05 16:38:02 CST"
max(datetime)
#> [1] "2021-03-15 17:38:02 CDT"
c(min, max)
#> [[1]]
#> function (... , na.rm = FALSE) .Primitive("min")
#>
#> [[2]]
#> function (... , na.rm = FALSE) .Primitive("max")
```

Теперь проблема стала очевидна. Мы не присвоили значение переменным `min` и `max`, так что по ошибке передали функции `min()` и `max()` в `sliderInput()`. Одним из способов решения этой проблемы является использование функции `range()`:

```
ui <- fluidPage(
  sliderInput("slider",
    "Select Range:",
    min = min(datetime),
    max = max(datetime),
    value = range(datetime)
  ),
  verbatimTextOutput("breaks")
)
```

Это типичный результат приведения кода к виду минимального воспроизводимого примера – как только мы упростили фрагмент кода до предела, причина ошибки стала более чем очевидна. Создание минимальных воспроизводимых примеров – одна из мощнейших техник отладки приложений.

Стоит отметить, что в процессе упрощения исходного кода мне пришлось немного поэкспериментировать и почитать о функциях, с которыми до этого не имел дела¹. Свой код обычно сокращать проще, чем чужой, поскольку его вы знаете куда лучше. Но вам все равно может понадобится поэкспериментировать, чтобы понять, где возникает ошибка. На это может уйти какое-то время, но без преимуществ вы не останетесь:

- это позволит вам создать описание проблемы, доступное для всех, кто знает Shiny, а не только для тех, кто еще и разбирается в вашей области;
- вы сможете лучше понять, как работает ваш код, и в будущем будете допускать меньше таких или похожих ошибок;
- со временем вы будете все быстрее создавать воспроизводимые примеры, и эта техника станет одной из ваших любимых при отладке приложений;

¹ К примеру, я даже понятия не имел, что функция `is.POSIXt()` является составной частью пакета `lubridate`.

- даже если вам не удастся создать идеальный воспроизводимый пример, любая работа по его упрощению одновременно облегчит задачу и тем, кто попытается вам помочь. Это особенно важно, когда вы обращаетесь за помощью к разработчикам пакетов, поскольку у них обычно не так много свободного времени.

Когда я пытаюсь кому-то помочь на сайте RStudio Community, я первым делом завожу речь о минимальном воспроизводимом примере. Нет, это не потому, что я не хочу помогать, это для меня действительно очень важно.

ЗАКЛЮЧЕНИЕ

В данной главе мы обсудили разные рабочие процессы, составляющие основу разработки приложений, их отладки и получения помощи от сообщества. Эти процессы могут показаться довольно размытыми и необязательными к выполнению, поскольку они не влияют напрямую на качество вашего приложения. Но я лично рассматриваю их как какую-то особую суперсилу, и тем, что я так многого добился в области разработки, я отчасти обязан тому, что всегда уделял немало времени анализу и улучшению качества моих рабочих процессов. И я настоятельно рекомендую вам делать то же самое!

В следующей главе мы перейдем к обсуждению макетов и тем приложений и приступим к освоению полезных техник и приемов при проектировании программ. Нет никакой необходимости читать все эти главы подряд, вы можете запросто пропускать темы, которые вам не так интересны, и читать только о том, что требуется вам для разработки конкретного приложения.

Глава 6

Макеты, темы, HTML

ВВЕДЕНИЕ

В данной главе вы откроете для себя новые инструменты для управления внешним видом ваших приложений. Мы начнем разговор с описания *макетов* (layout) страницы (одностраничного и многостраничного) для размещения ваших элементов ввода и вывода. После этого вы познакомитесь с фреймворком Bootstrap, представляющим собой набор инструментов CSS, используемый в Shiny, и узнаете, как применять темы. Завершим мы главу кратким экскурсом в то, что происходит «под капотом», – вы увидите, что при наличии знаний в области HTML и CSS можно еще лучше настроить приложения Shiny под свои требования. Как обычно, начнем с загрузки пакета Shiny:

```
library(shiny)
```

ОДНОСТРАНИЧНЫЕ МАКЕТЫ

В главе 2 вы узнали про элементы ввода и вывода, составляющие основу интерактивного интерфейса приложения Shiny. При этом мы практически не касались темы их размещения на странице, а просто использовали функцию `fluidPage()`, позволяющую быстро раскидать элементы по странице. На стадии обучения Shiny это вполне приемлемо, но создать визуально привлекательное приложение при помощи одной лишь этой функции будет весьма проблематично. В этой главе вы познакомитесь с другими функциями размещения элементов в макете.

Функции создания макета определяют высокоуровневую визуальную архитектуру приложения. Сам макет создается посредством *иерархии* (hierarchy) вызовов функций, при этом понятие иерархии в R практически совпадает с аналогичным термином в лексике HTML. Такая последовательность поможет вам лучше понять код макета. Взгляните на код макета, показанный ниже:

```
fluidPage(  
  titlePanel("Hello Shiny!"),
```

```

    sidebarLayout(
      sidebarPanel(
        sliderInput("obs", "Observations:", min = 0, max = 1000, value = 500)
      ),
      mainPanel(
        plotOutput("distPlot")
      )
    )
  )
)

```

Обратите внимание на иерархию вызовов функций:

```

fluidPage(
  titlePanel(),
  sidebarLayout(
    sidebarPanel(),
    mainPanel()
  )
)

```

И хотя мы пока не изучали все эти функции, вы легко можете догадаться об их предназначении по именам. Таким образом, можно предположить, что в результате запуска этого кода будет создано приложение с классическим дизайном: *строка заголовка* (title bar) в верхней части, *боковая панель* (sidebar) с ползунком и *основная панель* (main panel) с графиком. Легкость, с которой можно отследить иерархию вызовов функций по отступам в тексте, является достаточной причиной писать код единообразно.

В данном разделе мы поговорим о функциях, составляющих основу одностраничного макета приложения, после чего – в следующем разделе – перейдем к рассмотрению многостраничных приложений.

Также я настоятельно рекомендую вам ознакомиться с руководством по созданию макетов приложений Shiny по адресу <https://shiny.rstudio.com/articles/layout-guide.html> – пусть оно немного устарело, но содержит массу полезных советов.

Функции страницы

Наиболее важной, но в то же время наименее интересной функцией определения макета является `fluidPage()`, которую вы видели практически в каждом примере этой книги. Но что она делает? И что происходит при ее обособленном использовании? На рис. 6.1 показано такое приложение – согласитесь, крайне удручающее зрелище. В то же время за сценой здесь происходит очень много всего, поскольку функция `fluidPage()` обуславливает формирование кода HTML, CSS и JavaScript, необходимого для нужд Shiny.

Помимо функции `fluidPage()`, Shiny предоставляет еще пару функций, которые могут оказаться полезными в разных ситуациях. Это `fixedPage()` и `fillPage()`. Функция `fixedPage()` работает подобно `fluidPage()`, за исключением того, что страница в случае ее использования будет обладать фиксированной шириной, что не даст вашим приложениям бесосновательно разрастаться

в ширину на больших экранах. Функция *fillPage()* позволяет приложению распространяться на всю доступную высоту страницы и полезна при выводе графиков, которые должны занимать все доступное место. Подробности об этих функциях вы можете найти в документации.

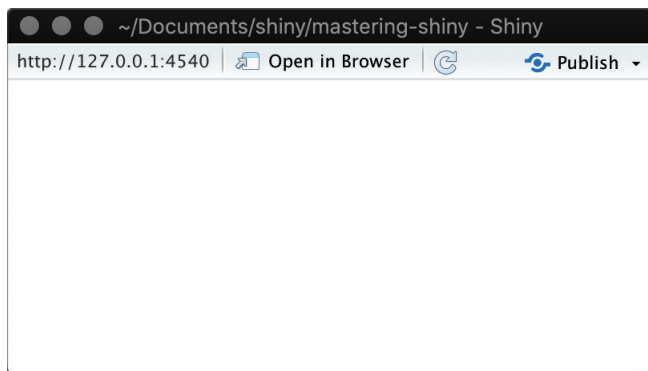


Рис. 6.1 ❖ Интерфейс пользователя, состоящий из одной функции *fluidPage()*

Страница с боковой панелью

Для создания более сложных страниц необходимо внутри функции *fluidPage()* встраивать другие функции макетов. К примеру, если вы хотите спроектировать макет, состоящий из двух колонок, с элементами ввода слева и элементами вывода справа, вы можете использовать функцию *sidebarLayout()* вместе с ее извечными спутниками *titlePanel()*, *sidebarPanel()* и *mainPanel()*. Фрагмент кода ниже демонстрирует базовую структуру такого приложения. Схематично внешний вид приложения показан на рис. 6.2:

```
fluidPage(
  titlePanel(
    # заголовок/описание приложения
  ),
  sidebarLayout(
    sidebarPanel(
      # элементы ввода
    ),
    mainPanel(
      # элементы вывода
    )
  )
)
```

Для создания более реалистичного приложения давайте снабдим его парой элементов ввода и вывода для визуализации центральной предельной теоремы, как показано на рис. 6.3. Запустив это приложение, вы сможете изменять *число выборов* (Number of samples) и наблюдать за тем, что рас-

пределение на гистограмме будет приближаться к нормальному по мере увеличения показателя:

```
ui <- fluidPage(
  titlePanel("Central limit theorem"),
  sidebarLayout(
    sidebarPanel(
      numericInput("m", "Number of samples:", 2, min = 1, max = 100)
    ),
    mainPanel(
      plotOutput("hist")
    )
  )
)

server <- function(input, output, session) {
  output$hist <- renderPlot({
    means <- replicate(1e4, mean(runif(input$m)))
    hist(means, breaks = 20)
  }, res = 96)
}
```

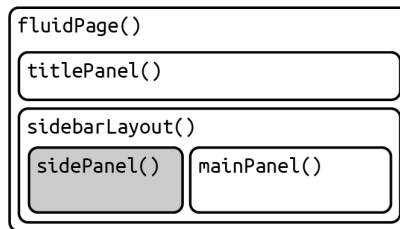


Рис. 6.2 ❖ Базовая структура приложения с боковой панелью

Central limit theorem

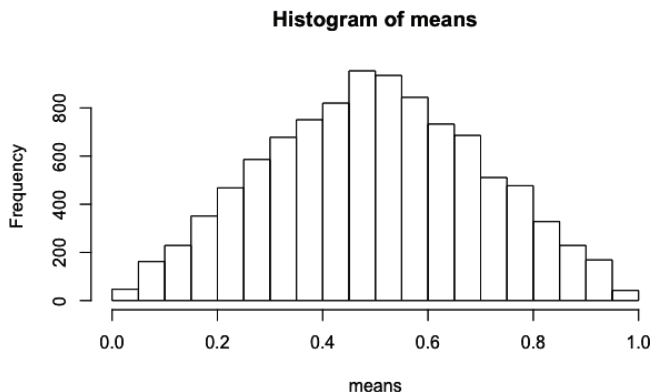


Рис. 6.3 ❖ Распространенный вид приложений: элементы управления в боковой панели, вывод результата – в основной

Многострочный вывод

По сути, функция `sidebarLayout()` является надстройкой над гибким многострочным макетом, который вы можете использовать напрямую для создания более сложных приложений. Для этого необходимо в обрамляющую функцию `fluidPage()` встроить нужное количество строк при помощи функции `fluidRow()` и столбцов при помощи функции `column()`. Показанный ниже шаблон генерирует приложение, схематично показанное на рис. 6.4:

```
fluidPage(
  fluidRow(
    column(4,
      ...
    ),
    column(8,
      ...
    )
  ),
  fluidRow(
    column(6,
      ...
    ),
    column(6,
      ...
    )
  )
)
```

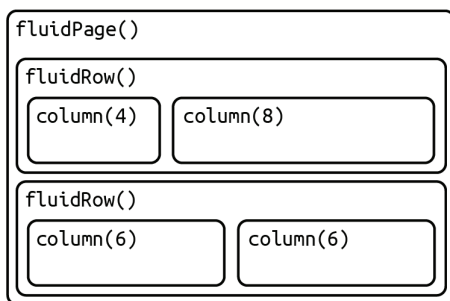


Рис. 6.4 ❖ Структура простого многострочного приложения

Каждая строка здесь формально состоит из 12 колонок, и первым аргументом функция `column()` принимает инструкцию о том, сколько колонок отвести под ячейку. 12-колоночный макет предоставляет достаточную гибкость, поскольку позволяет спокойно разместить на странице два, три или четыре столбца, а также использовать узкие колонки в качестве разделителей. Пример такого макета мы уже видели на рис. 4.3.

Если вы хотите больше узнать о дизайне приложений с использованием сетки, я настоятельно рекомендую прочитать классическую книгу *Grid*

systems in graphic design (<https://www.amazon.com/dp/3721201450>) от Джозефа Мюллера-Брокманна (Josef Müller-Brockmann).

Упражнения

Упражнение 1

Прочитайте документацию к функции `sidebarLayout()` на предмет определения ширины (в колонках) боковой и основной панелей. Можно ли воссоздать такой вид при помощи функций `fluidRow()` и `column()`?

Упражнение 2

Измените пример с визуализацией центральной предельной теоремы таким образом, чтобы боковая панель располагалась справа, а не слева.

Упражнение 3

Создайте приложение с двумя графиками, делящими ширину страницы пополам. Разместите элементы управления в контейнере под графиками, занимающем всю ширину страницы.

МНОГОСТРАНИЧНЫЕ МАКЕТЫ

С ростом уровня сложности вашего приложения вам постепенно может не хватить одной страницы для размещения всех необходимых элементов. В данном разделе мы рассмотрим разные способы использования функции `tabPanel()`, создающей иллюзию многостраничности. Да, речь идет именно об иллюзии, поскольку в результате применения этой функции вы получите приложение с единым файлом HTML в основе. При этом визуально приложение будет разделено на несколько частей, из которых видимой в любой момент времени будет только одна.

Многостраничные приложения (multipage app) особенно хорошо взаимодействуют с модулями, с которыми мы познакомимся в главе 19. Модули позволяют разделить серверную функцию в той же манере, в которой вы разделяете функцию пользовательского интерфейса, создавая независимые компоненты, взаимодействующие посредством хорошо отлаженных связей.

Наборы вкладок

Самый простой способ разбить страницу на части – использовать функцию `tabsetPanel()` и ее верную подругу – `tabPanel()`. Как вы увидите в приведенном ниже фрагменте кода, функция `tabsetPanel()` создает контейнер для любого количества функций `tabPanel()`, которые могут содержать любые другие компоненты HTML. На рис. 6.5 показан простой пример такого приложения:

```
ui <- fluidPage(  
  tabsetPanel(  
    
```

```

tabPanel("Import data",
  fileInput("file", "Data", buttonLabel = "Upload..."),
  textInput("delim", "Delimiter (leave blank to guess)", ""),
  numericInput("skip", "Rows to skip", 0, min = 0),
  numericInput("rows", "Rows to preview", 10, min = 1)
),
tabPanel("Set parameters"),
tabPanel("Visualise results")
)
)

```

The screenshot shows a web application with a tabset. The first tab, 'Import data', is selected and contains the following elements:

- Data** section: Includes an 'Upload...' button and a 'No file selected' message.
- Delimiter (leave blank to guess)**: A text input field.
- Rows to skip**: A numeric input field with the value '0'.
- Rows to preview**: A numeric input field with the value '10'.

The other two tabs, 'Set parameters' and 'Visualise results', are currently inactive.

Рис. 6.5 ❖ Функция `tabsetPanel()` позволяет пользователю выбрать нужную вкладку для отображения

Если вы хотите контролировать, какую вкладку открыл пользователь, то можете передать функции `tabsetPanel()` аргумент `id`, и она станет элементом ввода. На рис. 6.6 показан внешний вид следующего приложения:

```

ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      textOutput("panel")
    ),
    mainPanel(
      tabsetPanel(
        id = "tabset",
        tabPanel("panel 1", "one"),
        tabPanel("panel 2", "two"),
        tabPanel("panel 3", "three")
      )
    )
  )
)

```

```
server <- function(input, output, session) {
  output$panel <- renderText({
    paste("Current panel: ", input$tabset)
  })
}
```

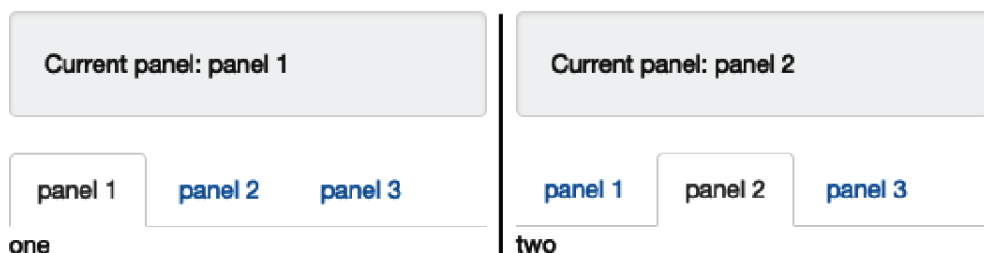


Рис. 6.6 ❖ При использовании аргумента `id` набор вкладок превращается в элемент ввода. Это позволяет изменить поведение приложения в зависимости от того, какая вкладка в данный момент видима

Обратите внимание, что функция `tabsetPanel()` может быть использована в любом месте вашего приложения. Вполне нормальной практикой является включение наборов вкладок в другие компоненты, в том числе в другие наборы вкладок.

Навигационный список и навигационная панель

По причине горизонтального расположения вкладок всегда есть определенное разумное ограничение на их максимальное количество, особенно если названия вкладок длинные. Функции `navBarPage()` и `navBarMenu()` предлагают два альтернативных способа размещения вкладок, не обладающих этим недостатком.

Функция `navlistPanel()` аналогична `tabsetPanel()`, за тем лишь исключением, что отображает вкладки не горизонтально, а вертикально в боковой панели. Также в этом случае вы можете добавлять заголовки в виде простого текста, как показано в следующем коде и на рис. 6.7:

```
ui <- fluidPage(
  navlistPanel(
    id = "tabset",
    "Heading 1",
    tabPanel("panel 1", "Panel one contents"),
    "Heading 2",
    tabPanel("panel 2", "Panel two contents"),
    tabPanel("panel 3", "Panel three contents")
  )
)
```

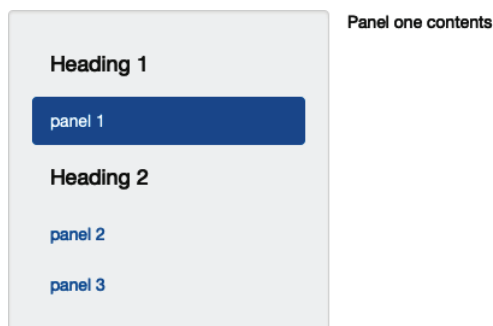


Рис. 6.7 ❖ Функция `navlistPanel()` помогает отобразить вкладки не горизонтально, а вертикально

Еще один подход заключается в использовании функции `navbarPage()`: в этом случае вкладки будут располагаться горизонтально, но вы сможете дополнять свою навигационную панель выпадающими пунктами меню при помощи функции `navbarMenu()`. Ниже представлен код примера, а на рис. 6.8 – его визуальное воплощение:

```
ui <- navbarPage(
  "Page title",
  tabPanel("panel 1", "one"),
  tabPanel("panel 2", "two"),
  tabPanel("panel 3", "three"),
  navbarMenu("subpanels",
    tabPanel("panel 4a", "four-a"),
    tabPanel("panel 4b", "four-b"),
    tabPanel("panel 4c", "four-c")
  )
)
```

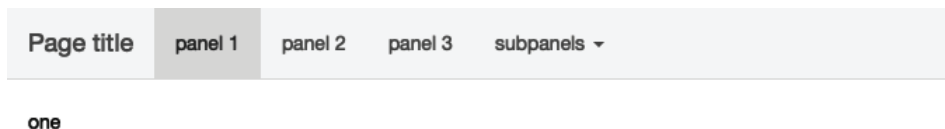


Рис. 6.8 ❖ Горизонтальная навигационная панель с выпадающим пунктом меню

Представленные в этом разделе инструменты позволят вам создать полноценное приложение с комфортной навигацией. Прежде чем двигаться дальше, вам необходимо чуть больше узнать о системе, лежащей в основе Shiny.

BOOTSTRAP

Осваивая различные способы настройки приложений под себя, вы просто обязаны познакомиться с фреймворком Bootstrap (<https://getbootstrap.com>),

который активно используется в Shiny. *Bootstrap* представляет из себя коллекцию правил HTML, стилей CSS и сниппетов JavaScript, объединенную под одной крышей в удобном виде. *Bootstrap* появился на основе фреймворка, изначально разработанного для Twitter, и за последнее десятилетие дорос до одной из наиболее популярных программных платформ CSS. *Bootstrap* также пользуется большой популярностью в R – вы наверняка видели немало документов, построенных при помощи функции `markdown::html_document()`, и использовали сайты, созданные при помощи пакета `pkgdown`. В обоих случаях используется также фреймворк *Bootstrap*.

Будучи разработчиком Shiny, вам не нужно слишком уж много думать о фреймворке *Bootstrap*, поскольку функции Shiny автоматически генерируют код HTML, совместимый с *Bootstrap*. Но о существовании этого фреймворка знать нужно, ведь это позволит вам:

- использовать функцию `bslib::bs_theme()` для настройки внешнего представления вашего кода, о чем мы будем говорить далее;
- использовать аргумент `class` для настройки некоторых макетов, а также элементов ввода и вывода, с применением названий классов *Bootstrap*, о чем мы уже говорили в главе 2 применительно к кнопкам в приложениях;
- создавать собственные функции для генерирования компонентов *Bootstrap*, не представленных в Shiny, как описано в документации *bslib* по адресу <https://rstudio.github.io/bslib/articles/utility-classes.html>.

Также вы можете использовать и другие фреймворки CSS. В R есть множество пакетов, облегчающих этот процесс за счет оборачивания наиболее популярных альтернатив *Bootstrap*:

- *shiny.semantic* (<https://appsilon.github.io/shiny.semantic>) от *Appsilon* (<https://appsilon.com>), основанный на *Fomantic-UI* (<https://fomantic-ui.com>);
- *shinyMobile* (<https://github.com/RinterRface/shinyMobile>) от *RinterRface* (<https://rinterface.com>), основанный на *Framework7* (<https://framework7.io>) и разработанный специально для мобильных приложений;
- *shinymaterial* (<https://ericrayanderson.github.io/shinymaterial>) от *Eric Anderson* (<https://github.com/ericrayanderson>), основанный на фреймворке *Material design* (<https://material.io/design>) от Google;
- *shinydashboard* (<https://rstudio.github.io/shinydashboard>) от RStudio, представляет собой систему макетов, разработанную для создания дашбордов.

Полный и наиболее актуальный список пакетов можно найти на GitHub по адресу <https://github.com/nanxstats/awesome-shiny-extensions>.

ТЕМЫ

Фреймворк *Bootstrap* настолько заполонил все сообщество R, что от этих стилей можно просто устать, – через какое-то время все приложения Shiny и документы Rmd становятся похожи один на другой. Решение состоит в использовании пакета *bslib* (<https://rstudio.github.io/bslib>) для создания собственных

тем. Пакет *bslib* является относительно новым и позволяет переопределить множество параметров Bootstrap по умолчанию, что открывает возможности для создания своих уникальных стилей. На момент написания книги *bslib* был применим, по сути, только к Shiny, но ведутся работы и по внедрению мощи этого пакета в пакетах *RMarkdown*, *pkgdown* и других.

Если вы разрабатываете приложение для компании, я настоятельно рекомендую потратить какое-то время на создание уникальной темы, соответствующей стилю организации.

Подготовка

Создайте тему (theme) при помощи функции `bslib::bs_theme()` и примените ее к приложению посредством передачи аргумента `theme` в функцию макета страницы:

```
fluidPage(
  theme = bslib::bs_theme(...)
)
```

Если не указать тему, Shiny будет применять классическую тему Bootstrap v3, которая использовалась практически всегда с момента появления фреймворка. По умолчанию функция `bslib::bs_theme()` применяет тему Bootstrap v4. Использование темы Bootstrap v4 вместо v3 не вызовет никаких проблем, если ваш выбор ограничивается встроенными компонентами. Неприятности могут возникнуть при использовании пользовательского кода HTML – в этом случае вы можете принудительно указать версию темы v3, передав в качестве аргумента строку `version = 3`.

Темы Shiny

Самый простой способ изменить внешний вид приложения – выбрать свою тему при помощи аргумента *bootswatch* (<https://bootswatch.com>) функции `bslib::bs_theme()`. На рис. 6.9 показано, как будет выглядеть приложение при выборе других тем, помимо *darkly*:

```
ui <- fluidPage(
  theme = bslib::bs_theme(bootswatch = "darkly"),
  sidebarLayout(
    sidebarPanel(
      textInput("txt", "Text input:", "text here"),
      sliderInput("slider", "Slider input:", 1, 100, 30)
    ),
    mainPanel(
      h1(paste0("Theme: darkly")),
      h2("Header 2"),
      p("Some text")
    )
  )
)
```



Рис. 6.9 ❖ Приложение, стилизованное при помощи четырех тем bootswatch: darkly, flatly, sandstone и united

Вы также можете настроить собственную тему с использованием таких аргументов функции `bs_theme()`, как *bg* (цвет фона), *fg* (основной цвет) и *base_font*¹:

```
theme <- bslib::bs_theme(
  bg = "#0b3d91",
  fg = "white",
  base_font = "Source Sans Pro"
)
```

Чтобы осуществить предварительный просмотр темы и настроить ее, используйте функцию `bslib::bs_theme_preview(theme)`. Ее вызов приведет к запуску приложения Shiny с отображением внешнего вида темы со множеством стандартных элементов управления. Кроме того, вы сможете интерактивно изменить ключевые параметры темы.

Темы графиков

Изменив внешний вид своего приложения, вы наверняка захотите соответствующим образом настроить и графики. К счастью, сделать это очень легко при помощи пакета *thematic* (<https://rstudio.github.io/thematic>), который позволяет стилизовать базовые графики, а также диаграммы, построенные при помощи пакетов *ggplot2* и *lattice*. Для этого просто вызовите функцию `thematic_shiny()` в серверной функции, и все параметры графиков будут адаптированы под вашу тему, как показано на рис. 6.10:

¹ Со шрифтами дело обстоит чуть сложнее, чем с цветами, поскольку вам нужно убедиться, что на клиенте установлен нужный вам шрифт. Подробнее об этом можно почитать в документации к функции `bs_theme()`.

```
library(ggplot2)

ui <- fluidPage(
  theme = bslib::bs_theme(bootswatch = "darkly"),
  titlePanel("A themed plot"),
  plotOutput("plot"),
)

server <- function(input, output, session) {
  thematic::thematic_shiny()

  output$plot <- renderPlot({
    ggplot(mtcars, aes(wt, mpg)) +
      geom_point() +
      geom_smooth()
  }, res = 96)
}
```

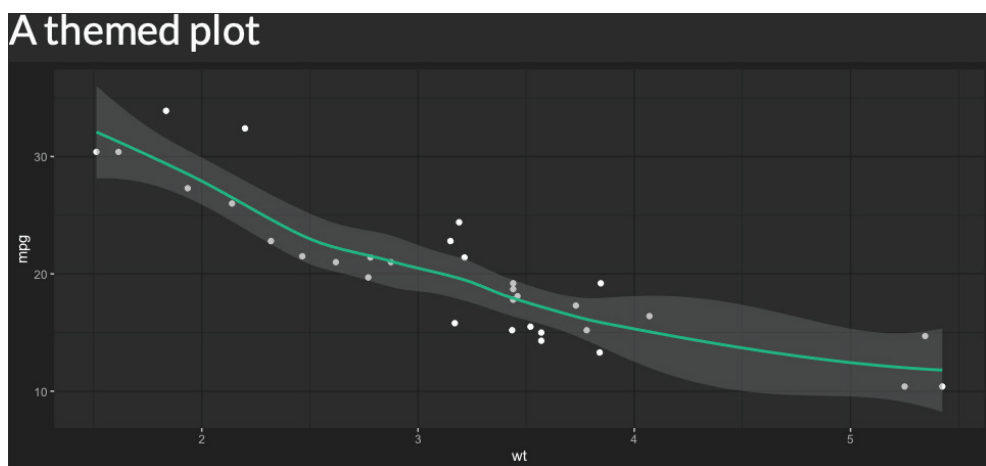


Рис. 6.10 ❖ Вызов функции `thematic::thematic_shiny()` гарантирует автоматическую стилизацию графиков *ggplot2*

Упражнения

Упражнение 1

Используйте функцию `bslib::bs_theme_preview()` для создания самой уродливой темы, какой только возможно.

За кулисами

Фреймворк Shiny спроектирован таким образом, что разработчику совсем не обязательно понимать все тонкости языка разметки HTML. Но при определенном знании HTML и CSS вы сможете выполнить более тонкую настройку

своих приложений Shiny. К сожалению, изучение языков HTML и CSS выходит за рамки данной книги, но вы можете самостоятельно пройтись по их основам на сайте MDN по ссылкам https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/HTML_basics и https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/CSS_basics соответственно.

Очень важно понять, что все эти функции ввода, вывода и построения макетов не таят в себе никакой магии, они просто генерируют HTML-код¹. Увидеть этот код вы можете, запустив функции интерфейса пользователя непосредственно в консоли, как показано ниже:

```
fluidPage(
  textInput("name", "What's your name?")
)

<div class="container-fluid">
  <div class="form-group shiny-input-container">
    <label for="name">What's your name?</label>
    <input id="name" type="text" class="form-control" value=""/>
  </div>
</div>
```

Обратите внимание, что здесь мы получили содержимое тега `<body>`. За наполнение тега `<head>` в Shiny отвечают другие функции. Если вы хотите включить в код дополнительные зависимости CSS или JavaScript, вам придется изучить функцию `htmltools::htmlDependency()`. Начать можете с поста *JavaScript for the R Package Developer* на *R-hub* по адресу <https://blog.r-hub.io/2020/08/25/js-r/#web-dependency-management> и четвертой главы книги *Outstanding User Interfaces with Shiny* по адресу <https://unleash-shiny.rinterface.com/htmltools-dependencies.html>.

У вас есть возможность добавить свой собственный HTML-код в интерфейс пользователя. Один из способов сделать это – включить в код разметку HTML при помощи функции `HTML()`. В следующем примере я использовал символьную константу² `r"()"` для безопасного включения кавычек в код:

```
ui <- fluidPage(
  HTML(r"(
    <h1>This is a heading</h1>
    <p class="my-class">This is some text!</p>
    <ul>
      <li>First bullet</li>
      <li>Second bullet</li>
    </ul>
  )")
)
```

Если вы эксперт в области HTML/CSS, вам будет приятно узнать, что функцию `fluidPage()` можно опустить, оставив только код HTML. Подробнее об этом приеме можно почитать по адресу <https://shiny.rstudio.com/articles/html-ui.html>.

¹ Настоящая магия происходит при связывании элементов ввода и вывода в скриптах JavaScript, но эта тема выходит за рамки данной книги.

² Была введена в R версии 4.0.0.

Также вы можете использовать вспомогательные функции Shiny для создания разметки HTML. Для наиболее важных элементов разметки предусмотрены отдельные функции вроде `h1()` и `p()`, а к остальным вы можете обратиться посредством объекта `tags`, как показано ниже. При этом именованные аргументы становятся атрибутами, а неименованные – вложенными элементами. Таким образом, предыдущий пример мы можем воссоздать следующим образом:

```
ui <- fluidPage(
  h1("This is a heading"),
  p("This is some text", class = "my-class"),
  tags$sul(
    tags$li("First bullet"),
    tags$li("Second bullet")
  )
)
```

Одним из преимуществ генерирования разметки HTML с помощью кода является возможность вплетения компонентов Shiny в вашу собственную структуру. Например, в следующем фрагменте мы создаем абзац с текстом, содержащим два элемента вывода, один из которых будет выделен жирным шрифтом:

```
tags$p(
  "You made ",
  tags$b("$", textOutput("amount", inline = TRUE)),
  " in the last ",
  textOutput("days", inline = TRUE),
  " days "
)
```

Обратите внимание, как мы использовали аргумент `inline = TRUE` – по умолчанию функция `textOutput()` выводится в виде абзаца.

Если вы хотите больше узнать об использовании HTML, CSS и JavaScript для обогащения ваших приложений, я очень рекомендую почитать книгу *Outstanding User Interfaces with Shiny* от Дэвида Гранжона (David Granjon) по адресу <https://unleash-shiny.rinterface.com/index.html>.

ЗАКЛЮЧЕНИЕ

В данной главе мы рассмотрели множество важнейших инструментов для создания богатых и сложных приложений Shiny. Вы узнали, как в Shiny проектировать одностраничные и многостраничные макеты при помощи функций `fluidPage()` и `tabsetPanel()` и менять внешний вид приложений посредством тем. Также мы немного заглянули за кулисы фреймворка и посмотрели, как Shiny использует Bootstrap, а функции ввода и вывода генерируют разметку HTML, которую вы можете создавать и самостоятельно.

В следующей главе мы подробно поговорим о еще одной важной составляющей любого приложения Shiny – графиках.

Глава 7

Графики

В главе 2 мы вскользь коснулись функции `renderPlot()`, представляющей мощный инструмент для построения графиков в приложениях. Сейчас же мы поговорим о ней более подробно и посмотрим, как при помощи нее можно строить интерактивные диаграммы, реагирующие на события мыши. Попутно мы освоим пару полезных техник, включая создание графиков переменной ширины и высоты и вывода изображений посредством функции `renderImage()`.

В данной главе мы будем, помимо Shiny, использовать пакет *ggplot2*, поскольку его я применяю для построения большинства графиков:

```
library(shiny)
library(ggplot2)
```

ИНТЕРАКТИВНОСТЬ

Одна из самых примечательных характеристик функции `plotOutput()` состоит в том, что она определяет элемент вывода, который одновременно может быть и элементом ввода, отвечающим на действия пользователя. Эта особенность позволяет нам строить *интерактивные графики* (interactive plot), с которыми пользователь может взаимодействовать напрямую. Интерактивные графики представляют собой невероятно мощный инструмент, присутствующий во множестве приложений. В этой главе я, конечно, не смогу дать вам подробное описание данного инструмента, так что мы сосредоточимся на основах, но я подскажу ресурсы, где вы сможете получить наиболее полную информацию.

Основы

График может реагировать на четыре следующих события мыши¹: `click` (щелчок мыши), `dblclick` (двойное нажатие кнопки мыши), `hover` (удержа-

¹ На момент написания книги в Shiny не поддерживались события касания, что не позволяло использовать интерактивные графики на мобильных устройствах. Надеюсь, после выхода книги эти события уже будут поддерживаться.

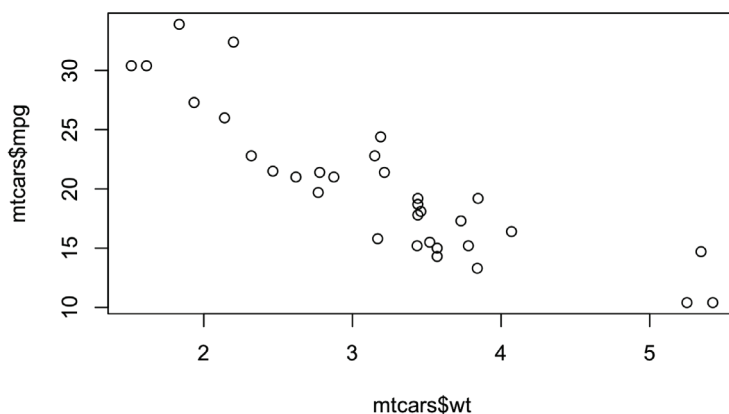
ние мыши на одном месте в течение какого-то времени) и `brush` (выделение прямоугольной области). Чтобы преобразовать эти события в элементы ввода Shiny, мы передаем соответствующую строку в виде аргумента функции `plotOutput()`, например `plotOutput("plot", click = "plot_click")`. В результате будет создан элемент ввода `input$plot_click`, который может быть использован для перехвата щелчков мыши на графике.

Ниже представлен простой пример перехвата такого события. Мы регистрируем элемент ввода `plot_click` и впоследствии используем его для обновления элемента вывода путем передачи ему координат щелчка мыши. На рис. 7.1 показан результат работы приложения:

```
ui <- fluidPage(
  plotOutput("plot", click = "plot_click"),
  verbatimTextOutput("info")
)

server <- function(input, output) {
  output$plot <- renderPlot({
    plot(mtcars$wt, mtcars$mpg)
  }, res = 96)

  output$info <- renderPrint({
    req(input$plot_click)
    x <- round(input$plot_click$x, 2)
    y <- round(input$plot_click$y, 2)
    cat("[", x, ", ", y, "]", sep = "")
  })
}
```



[1.51, 30.4]

Рис. 7.1 ❖ Щелчок по левой верхней точке данных привел к выводу ее координат.

Посмотреть это приложение онлайн можно по адресу
<https://hadley.shinyapps.io/ms-click>

Обратите внимание на использование функции `req()`, чтобы убедиться, что приложение не выполняло никаких действий перед первым щелчком, а координаты выражены через соответствующие переменные `wt` и `mpg`.

В следующем разделе мы рассмотрим события более детально. Начнем с самого простого события щелчка мышью (`click`), после чего поговорим о смежных событиях двойного щелчка и наведения мыши (`dblclick` и `hover` соответственно). Далее мы коснемся события `brush`, определяемого по четырём сторонам (`xmin`, `xmax`, `ymin` и `ymax`). После этого рассмотрим несколько примеров обновления графиков в результате действий пользователя и в завершение перечислим некоторые ограничения интерактивных графиков в Shiny.

Щелчки мыши

События, связанные с нажатием кнопки мыши, возвращают довольно богатый список значений, содержащий массу полезной информации. Наиболее важными компонентами этого списка, конечно, являются `x` и `y`, отвечающие за координаты события. Но мы в основном не будем обращаться к этой структуре данных напрямую, поскольку она бывает нужна относительно редко (если вы хотите разобраться в этой структуре подробно, можете обратиться к приложению в галерее Shiny по адресу <https://gallery.shinyapps.io/095-plot-interaction-advanced>). Вместо этого мы будем применять вспомогательную функцию `nearPoints()`, возвращающую датафрейм со строками, находящимися неподалеку¹ от места щелчка.

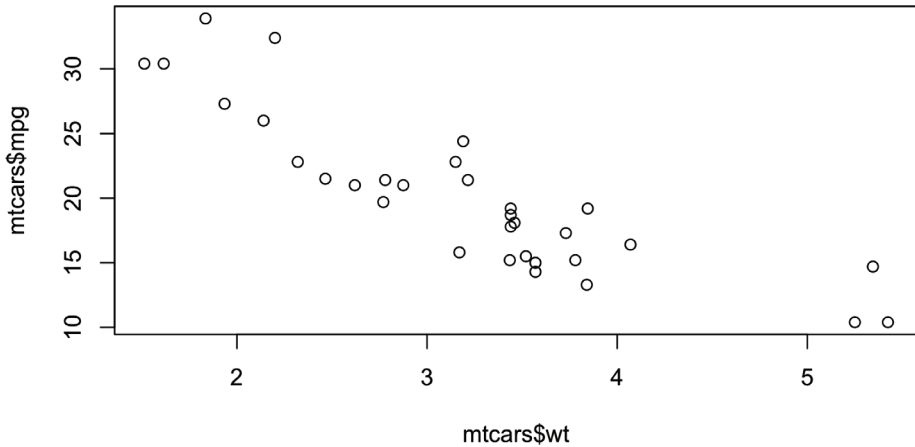
Ниже приведен простой пример работы функции `nearPoints()` с выводом списка точек данных, находящихся в непосредственной близости от щелчка мыши. На рис. 7.2 показан результат работы этого приложения:

```
ui <- fluidPage(
  plotOutput("plot", click = "plot_click"),
  tableOutput("data")
)

server <- function(input, output, session) {
  output$plot <- renderPlot({
    plot(mtcars$wt, mtcars$mpg)
  }, res = 96)

  output$data <- renderTable({
    nearPoints(mtcars, input$plot_click, xvar = "wt", yvar = "mpg")
  })
}
```

¹ Обратите внимание, что функция называется не `nearestPoints()`, так что она вполне может вернуть пустой список, если вы произведете щелчок мыши на большом удалении от точек.



mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
30.40	4.00	95.10	113.00	3.77	1.51	16.90	1.00	1.00	5.00	2.00

Рис. 7.2 ❖ Функция `nearPoints()` переводит координаты графика в строки данных, облегчая отображение информации о точках, по которым вы щелкаете.

Посмотреть это приложение онлайн можно по адресу
<https://hadley.shinyapps.io/ms-nearPoints>

Здесь мы передали функции `nearPoints()` четыре аргумента: датафрейм, лежащий в основе графика, входное событие и имена переменных на осях. При использовании пакета *ggplot2* вам необходимо будет передать только два аргумента, поскольку переменные `xvar` и `yvar` можно автоматически вывести из структуры данных на графике. По этой причине я буду до конца главы использовать этот пакет для построения графиков. Ниже представлена реализация предыдущего примера с помощью пакета *ggplot2*:

```
ui <- fluidPage(
  plotOutput("plot", click = "plot_click"),
  tableOutput("data")
)

server <- function(input, output, session) {
  output$plot <- renderPlot({
    ggplot(mtcars, aes(wt, mpg)) + geom_point()
  }, res = 96)

  output$data <- renderTable({
    req(input$plot_click)
    nearPoints(mtcars, input$plot_click)
  })
}
```

Наверняка вам интересно, что же именно возвращает функция `nearPoints()`. Чтобы это узнать, лучше всего использовать функцию `browser()`, которую мы обсуждали ранее в этой книге:

```
...
  output$data <- renderTable({
    req(input$plot_click)
    browser()
    nearPoints(mtcars, input$plot_click)
  })
...
```

После запуска приложения и щелчка мышью по графику откроется интерактивное окно отладки, в котором можно посмотреть, что возвращает функция `nearPoints()`:

```
nearPoints(mtcars, input$plot_click)
#>      mpg  cyl disp  hp drat   wt  qsec vs  am gear carb
#> Datsun 710 22.8   4 108 93 3.85 2.32 18.61 1  1   4    1
```

Также можно использовать функцию `nearPoints()` с аргументами `allRows = TRUE` и `addDist = TRUE`. В результате будет возвращен исходный датафрейм с двумя новыми столбцами:

- `dist_` – дистанция между строкой и событием (в пикселях);
- `selected_` – говорит о том, находится ли эта точка данных в непосредственной близости от события, то есть вернулась ли бы она, если бы аргументу `allRows` было передано значение `FALSE`.

Чуть позже мы посмотрим на примере, как это работает.

Другие события мыши

Одинаковый подход работает для событий `click`, `dblclick` и `hover`: нужно только менять название аргумента. При необходимости можно получить дополнительный контроль над событиями, передав функции `clickOpts()`, `dblclickOpts()` или `hoverOpts()` вместо строки с идентификатором элемента ввода. Это требуется не так часто, так что мы не будем останавливаться на этом приеме подробно – если нужно, вы можете обратиться за разъяснениями к документации. Применительно к графикам вы можете отслеживать самые разные типы взаимодействия. Главное – поставить в известность пользователя о том, что он может и чего не может делать с графиком. Одним из недостатков использования интерактивных графиков с отслеживанием событий мыши является их слабая информативность¹.

Выделение прямоугольной области

Еще одним способом выделения точек на графике является использование *кисти* (`brush`) для обозначения прямоугольной области с четырьмя граница-

¹ Считается, что необходимость дополнять приложение текстовыми описаниями говорит о его излишней сложности, так что лучше этого избегать. Именно эта идея лежит в основе *аффордансов* (*affordances*) – предположений о том, что объект должен естественным образом побуждать пользователя к правильным действиям с ним, как описал в своей книге *The Design of Everyday Things* Дон Норман (Don Norman).

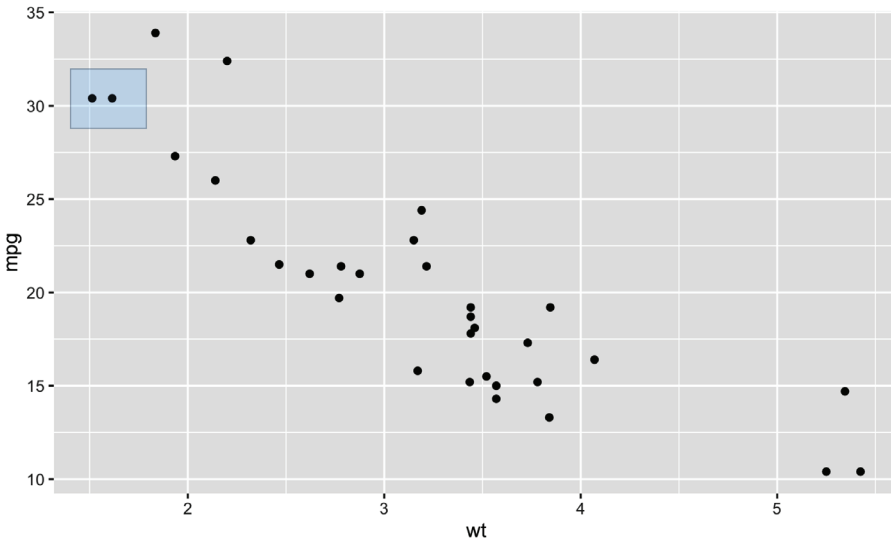
ми. В Shiny применение операции *выделения прямоугольной области* реализовано довольно понятно – после освоения события `click` и функции `nearPoints()` вам достаточно будет сменить аргумент на `brush` и использовать вспомогательную функцию `brushedPoints()`.

Давайте рассмотрим простой пример использования области выделения для обнаружения точек данных, попавших в ее пределы. На рис. 7.3 показан результат работы приложения:

```
ui <- fluidPage(
  plotOutput("plot", brush = "plot_brush"),
  tableOutput("data")
)

server <- function(input, output, session) {
  output$plot <- renderPlot({
    ggplot(mtcars, aes(wt, mpg)) + geom_point()
  }, res = 96)

  output$data <- renderTable({
    brushedPoints(mtcars, input$plot_brush)
  })
}
```



mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
30.40	4.00	75.70	52.00	4.93	1.61	18.52	1.00	1.00	4.00	2.00
30.40	4.00	95.10	113.00	3.77	1.51	16.90	1.00	1.00	5.00	2.00

Рис. 7.3 ❖ Передача аргумента `brush` позволила пользователю выделять прямоугольную область на графике.

Точки данных, попавшие в выделенную область, вынесены в таблицу.

Посмотреть это приложение онлайн можно по адресу
<https://hadley.shinyapps.io/ms-brushedPoints>

Используйте функцию `brushOpts()` для контроля за цветом выделения (с аргументами `fill` и `stroke`) или ограничения области выделения одним направлением (аргумент `direction` может принимать значения "x" или "y"). Это может быть полезно, в частности, при выделении временных диапазонов.

Изменение графика

До сих пор мы отображали результаты взаимодействия пользователя с графиком в других элементах вывода. Но настоящая магия происходит, когда вид графика, с которым пользователь непосредственно взаимодействует, динамически меняется. К сожалению, эта техника требует знания и понимания работы реактивной функции `reactiveVal()`, которую мы еще не проходили. Более подробно мы будем говорить о ней в главе 16, а здесь посмотрим на пример ее использования – уж больно она хороша. Возможно, вам захочется вернуться к этому примеру после прочтения главы 16, но я надеюсь, что и без дополнительной теории вы сможете понять всю мощь этого функционала.

Как понятно из названия, функция `reactiveVal()` весьма близка к `reactive()`. Вы создаете *реактивное значение* (reactive value) с определенным содержанием по умолчанию при помощи вызова функции `reactiveVal()` и извлекаете это значение так же, как и в случае с реактивным выражением:

```
val <- reactiveVal(10)
val()
#> [1] 10
```

Разница заключается в том, что вы можете менять реактивное значение, в результате чего все реактивные потребители, связанные с ним, будут пересчитаны. В реактивном значении используется специальный синтаксис для изменения его содержимого – вы вызываете его как функцию, передавая в качестве параметра новое значение:

```
val(20)
val()
#> [1] 20
```

Соответственно, изменить реактивное значение при помощи его текущего содержимого можно следующим образом:

```
val(val() + 1)
val()
#> [1] 21
```

Если вы попытаетесь выполнить представленный код в консоли, то увидите ошибку, поскольку он должен запускаться в рамках реактивного окружения. Эта особенность существенно осложняет процесс отладки приложения, ведь вам придется вызывать функцию вроде `browser()` для остановки приложения внутри функции `shinyApp()`. Об этом ограничении мы подробно поговорим в главе 16.

Сейчас же мы отложим все сложности изучения функции `reactiveVal()` и посмотрим, как можно ее использовать. Представьте, что вам необходимо визуализировать расстояние в пикселях от места щелчка мышью до точек на графике. В показанном ниже приложении мы сначала создадим реактивное значение, в котором будем хранить искомые расстояния, и инициализируем его константой, которая будет использоваться вплоть до первого щелчка мыши. После этого мы воспользуемся функцией `observeEvent()` для обновления реактивного значения и визуализируем дистанцию до точек данных на графике *ggplot* посредством изменения размера этих точек. Результат работы приложения показан на рис. 7.4:

```
set.seed(1014)
df <- data.frame(x = rnorm(100), y = rnorm(100))

ui <- fluidPage(
  plotOutput("plot", click = "plot_click", )
)

server <- function(input, output, session) {
  dist <- reactiveVal(rep(1, nrow(df)))
  observeEvent(input$plot_click,
    dist(nearPoints(df, input$plot_click, allRows = TRUE, addDist = TRUE)$dist_)
  )

  output$plot <- renderPlot({
    df$dist <- dist()
    ggplot(df, aes(x, y, size = dist)) +
      geom_point() +
      scale_size_area(limits = c(0, 1000), max_size = 10, guide = NULL)
  }, res = 96)
}
```

Здесь важно отметить два подхода, которые я использовал:

- добавил расстояния в датафрейм перед отрисовкой графика. Мне кажется хорошей практикой объединять связанные переменные в датафрейме перед визуализацией;
- установил в функции `scale_size_area()` аргумент `limits`, чтобы обеспечить сопоставимые размеры кружков для любых щелчков мыши. Для поиска подходящего диапазона мне пришлось немного поэкспериментировать, но при необходимости вы можете произвести точные расчеты.

Теперь рассмотрим более сложный пример. Допустим, я хочу использовать выделение прямоугольной области для постепенного добавления точек к своему выбору. Здесь я буду помечать выбранные элементы цветом, но можно использовать любую другую реализацию. Для начала я инициализирую реактивное значение `selected` вектором из `FALSE`, а затем использую функцию `brushedPoints()` и оператор `|` для добавления выделенных точек к своему выбору. Чтобы пользователь мог начать делать выбор заново, назовем сброс выделения на двойной щелчок мыши. На рис. 7.5 показана пара скриншотов работающего приложения:

```

ui <- fluidPage(
  plotOutput("plot", brush = "plot_brush", dblclick = "plot_reset")
)

server <- function(input, output, session) {
  selected <- reactiveVal(rep(FALSE, nrow(mtcars)))

  observeEvent(input$plot_brush, {
    brushed <- brushedPoints(mtcars, input$plot_brush, allRows = TRUE)$selected_
    selected(brushed | selected())
  })
  observeEvent(input$plot_reset, {
    selected(rep(FALSE, nrow(mtcars)))
  })

  output$plot <- renderPlot({
    mtcars$sel <- selected()
    ggplot(mtcars, aes(wt, mpg)) +
      geom_point(aes(colour = sel)) +
      scale_colour_discrete(limits = c("TRUE", "FALSE"))
  }, res = 96)
}

```

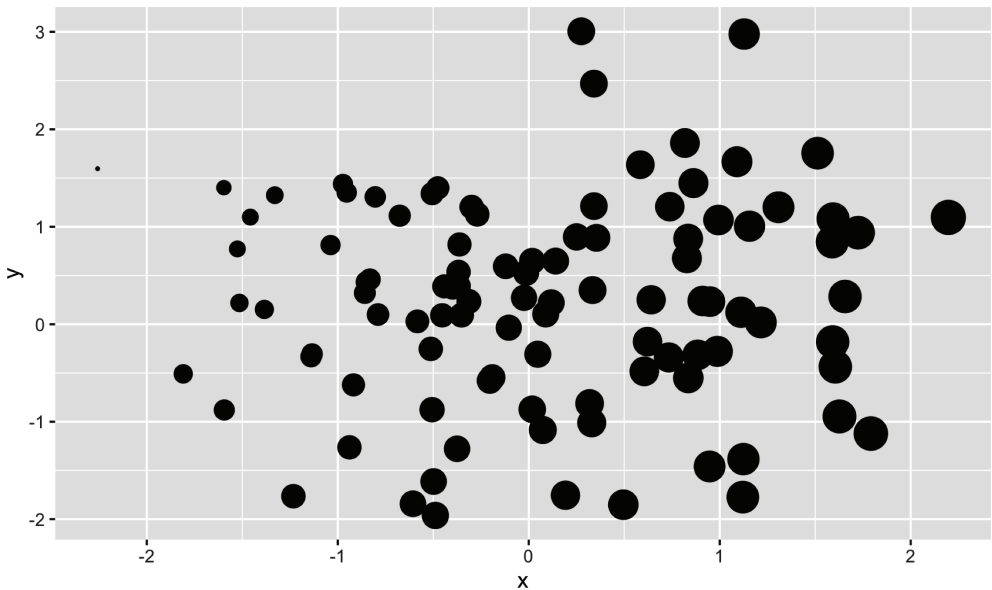


Рис. 7.4 ❖ Приложение использует функцию `reactiveVal()` для хранения расстояний до точки последнего щелчка мыши, которые впоследствии преобразуются в размеры кружков на графике. Здесь показано состояние приложения после щелчка на левой верхней точке данных.

Посмотреть это приложение онлайн можно по адресу
<https://hadley.shinyapps.io/ms-modifying-size>

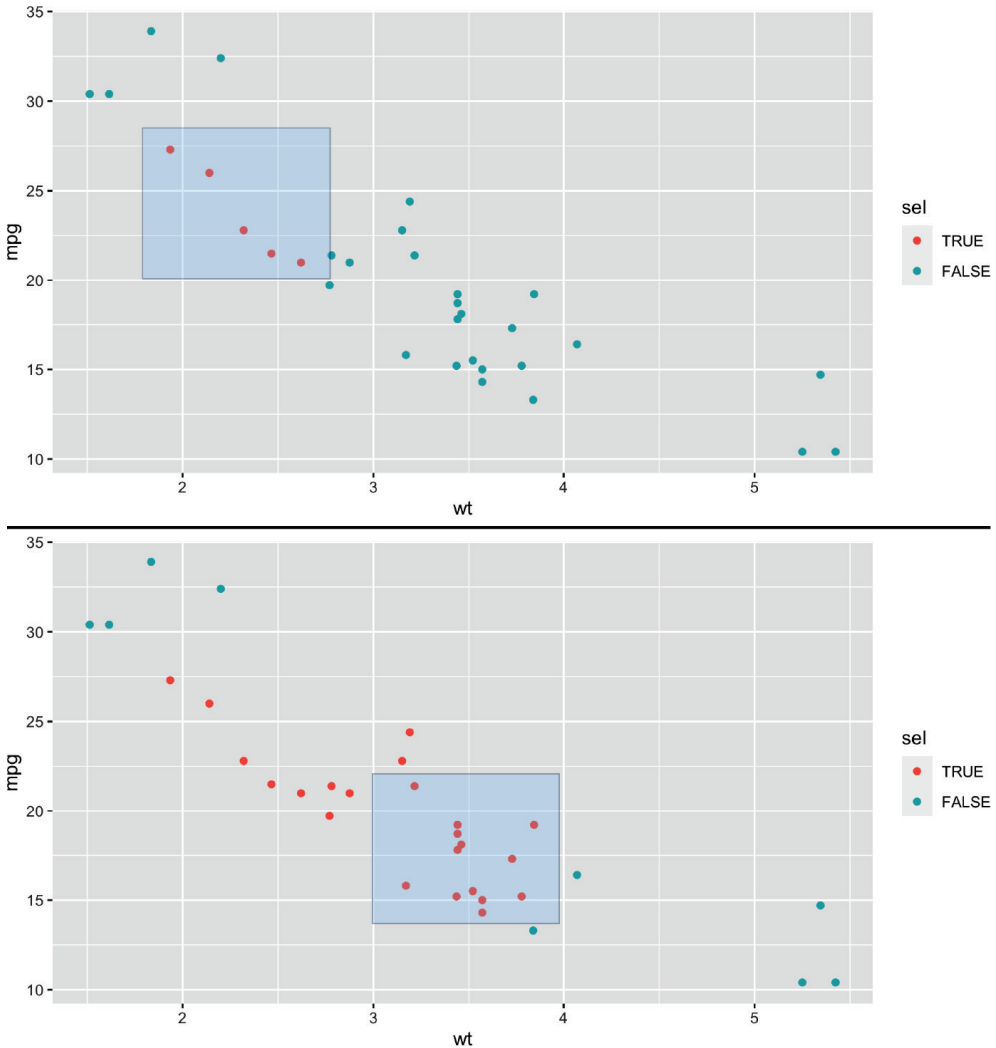


Рис. 7.5 ❖ Выделение точек добавляет их к уже существующему выбору

Здесь я снова установил аргумент `limits`, чтобы легенда и цвета не изменились после первого нажатия на кнопку мыши.

Ограничения интерактивности

Чтобы выяснить, какие есть ограничения у интерактивных графиков, необходимо понять, как в них организованы потоки данных. А все происходит примерно так.

1. JavaScript перехватывает событие мыши.
2. Shiny посылает данные, связанные с событием мыши, обратно в R, сообщая приложению, что элемент ввода утратил актуальность.

3. Все нижестоящие реактивные потребители пересчитываются.
4. Функция `plotOutput()` генерирует новое изображение PNG и посылает его браузеру.

Для локальных приложений узким местом в плане производительности будет время, необходимое для прорисовки графика. В зависимости от их сложности на это может потребоваться несколько долей секунды. В случае с приложениями, размещенными в интернете, нужно также учитывать время, необходимое для передачи события из браузера в R и отрисованного графика – обратно из R в браузер.

В целом это ведет к невозможности применения приложений Shiny в тех областях, где действия и ответ должны происходить практически одновременно. Если вам необходима такая скорость реакции приложения, придется больше вычислений выполнять в JavaScript. Один из способов – использовать пакет R, являющийся надстройкой над графической библиотекой JavaScript. На момент написания книги, полагаю, лучшим выбором является пакет *plotly*, и про это много написано в книге *Interactive Web-Based Data Visualization with R, Plotly, and Shiny* (<https://plotly-r.com>) Карсона Зиверта (Carson Sievert).

ДИНАМИЧЕСКАЯ ШИРИНА И ВЫСОТА

Оставшаяся часть главы будет не такой захватывающей, как интерактивные графики, но без этого материала в книге просто не обойтись.

Первое, о чем хочется рассказать, – так это о том, что размеры графиков сами по себе могут быть реактивными, то есть меняться в ответ на действия пользователя. Для этого необходимо передать функции `renderPlot()` в виде аргументов `width` и `height` функции без параметров, и сделать это нужно в серверной функции, а не в функции интерфейса пользователя, поскольку мы определяем изменяющиеся параметры. Эти функции не должны принимать аргументов, а возвращать должны желаемый размер в пикселях. Вычисления будут производиться в реактивном окружении, а значит, вы можете сделать размеры своих графиков динамическими.

В следующем примере мы продемонстрируем эту идею. Расположим в приложении два ползунка для управления размерами графика. Внешний вид приложения показан на рис. 7.6. Обратите внимание, что при изменении ширины и высоты графика данные остаются прежними, их обновления не происходит:

```
ui <- fluidPage(
  sliderInput("height", "height", min = 100, max = 500, value = 250),
  sliderInput("width", "width", min = 100, max = 500, value = 250),
  plotOutput("plot", width = 250, height = 250)
)

server <- function(input, output, session) {
  output$plot <- renderPlot(
```



```

width = function() input$width,
height = function() input$height,
res = 96,
{
  plot(rnorm(20), rnorm(20))
}
)
}

```

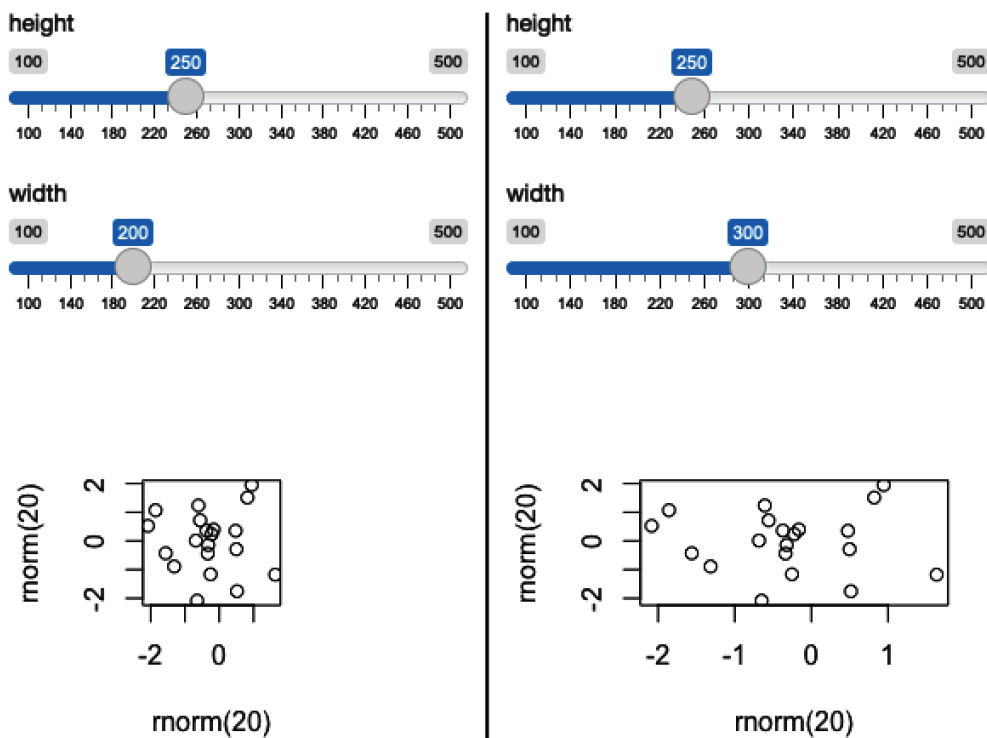


Рис. 7.6 ❖ График с динамическими размерами, зависящими от действий пользователя.

На этом рисунке видно, что мы изменили ширину диаграммы.

Посмотреть приложение онлайн можно по адресу

<https://hadley.shinyapps.io/ms-resize>

В реальных приложениях вы можете использовать более сложные выражения в функциях `width` и `height`. Например, если будете строить множественные фасетированные графики в *ggplot2*, вы можете попробовать использовать эту технику для увеличения размера диаграммы, чтобы примерно сохранить размеры каждого отдельного графика прежними¹.

¹ К сожалению, нет простого способа сохранить их размеры в точности такими же, поскольку мы не можем узнать размеры фиксированных элементов за границами графика.

ИЗОБРАЖЕНИЯ

Для вывода в приложение существующего изображения вы можете использовать функцию `renderImage()`. К примеру, у вас на диске может находиться директория с фотографиями, которые вы хотите показать пользователю.

В следующем фрагменте кода мы продемонстрируем основы работы с функцией `renderImage()` на примере фотографий со щеночками. Фотографии я взял со своего любимого бесплатного сайта <https://unsplash.com>. Результат работы приложения и, конечно, фотографии щенков представлены на рис. 7.7:

```
puppies <- tibble::tribble(
  ~breed, ~ id, ~author,
  "corgi", "eoqnr8ikwFE", "alvannee",
  "labrador", "KCdYn0xu2fU", "shanequymon",
  "spaniel", "TzjMd7i5WQI", "_redo_"
)

ui <- fluidPage(
  selectInput("id", "Pick a breed", choices = setNames(puppies$id, puppies$breed)),
  htmlOutput("source"),
  imageOutput("photo")
)

server <- function(input, output, session) {
  output$photo <- renderImage({
    list(
      src = file.path("puppy-photos", paste0(input$id, ".jpg")),
      contentType = "image/jpeg",
      width = 500,
      height = 650
    )
  }, deleteFile = FALSE)

  output$source <- renderUI({
    info <- puppies[puppies$id == input$id, , drop = FALSE]
    HTML(glue::glue("<p>
      <a href='https://unsplash.com/photos/{info$id}'>original</a> by
      <a href='https://unsplash.com/@{info$author}'>{info$author}</a>
    </p>"))
  })
}
```

Функции `renderImage()` необходимо возвращать список. При этом единственным важным аргументом является `src`, представляющий локальный путь к файлу с изображением. Также вы можете предоставить следующие параметры:

- `contentType` – определяет MIME-тип изображения. В случае его отсутствия Shiny сделает предположение о типе файла по его расширению, так что явно указывать этот параметр необходимо только при отсутствии расширения;

- width и height – ширина и высота изображения, если известны;
- другие аргументы, такие как class или alt, будут добавлены в тег в разметке HTML в виде атрибутов.

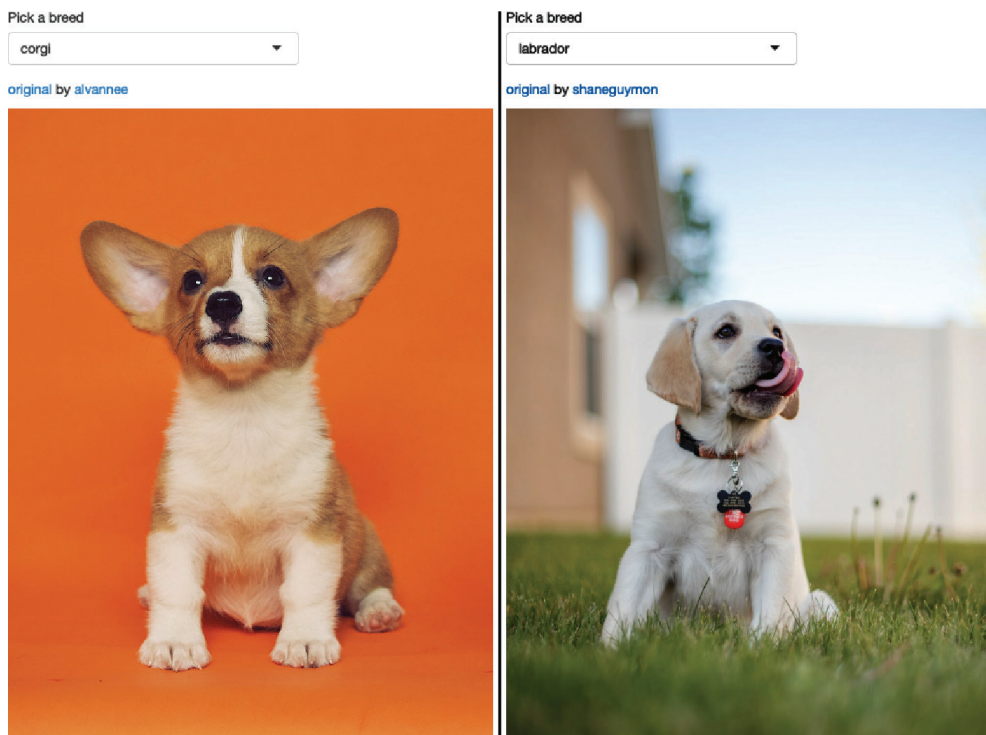


Рис. 7.7 ❖ Приложение со щеночками и функцией `renderImage()`.
Посмотреть приложение онлайн можно по адресу
<https://hadley.shinyapps.io/ms-puppies>

Также вы *должны* передать в функцию `renderImage()` аргумент `deleteFile`. К сожалению, изначально эта функция предназначалась для работы с временными файлами, так что она удаляла фотографии после их вывода. Очевидно, что такое поведение функции было неприемлемым и опасным, и в версии Shiny 1.5.0 оно было изменено. Теперь Shiny не удаляет показанные файлы, но требует при этом, чтобы вы явно указывали желаемое действие.

Подробно о функции `renderImage()` и способах ее использования можно почитать на официальном сайте Shiny по адресу <https://shiny.rstudio.com/articles/images.html>.

ЗАКЛУЧЕНИЕ

Визуализация данных – одна из важнейших составляющих приложений, призванная донести до пользователя нужную ему информацию. В данной главе мы рассмотрели несколько приемов и техник для визуального обогащения ваших приложений. В следующей главе мы коснемся не менее важной темы и поговорим о способах предоставления пользователю обратной связи о том, что происходит в приложении. Это особенно важно для действий, которые могут занимать определенное время.

Глава 8

Обратная связь с пользователем

Сделать приложение более дружелюбным и удобным в использовании можно, добавив оповещения о том, что в данный момент происходит. Это могут быть сообщения, предупреждающие о некорректном вводе, или индикаторы хода выполнения длительных задач. Определенную связь с пользователем можно поддерживать при помощи стандартных элементов вывода, и вы уже знаете, как это делать. Но иногда нам требуется чуть больше. В данной главе мы рассмотрим разные способы обеспечить пользователя понятной обратной связью.

Начнем с применения техники *проверки* (validation) значений, помогающей известить пользователя о некорректном вводе в одном или нескольких полях. Затем перейдем к *сообщениям* (notification), позволяющим держать пользователя в курсе происходящего, после чего рассмотрим применение *индикаторов хода выполнения задач* (progress bar), с которыми пользователю будет легче ориентироваться в приложении. В конце главы мы посмотрим, как можно выводить диалоговые окна с подтверждением действий пользователя и даже предоставить ему возможность отменить свое действие.

В этой главе мы будем пользоваться пакетами *shinyFeedback* (<https://github.com/merlinoa/shinyFeedback>) от Энди Мерлино (Andy Merlino) и *waiter* (<https://waiter.john-coene.com/#>) от Джона Коэна (John Coene). Также я порекомендовал бы вам следить за пакетом *shinyvalidate* от Джо Ченга (Joe Cheng), который в данный момент находится в разработке. Как и всегда, начнем с загрузки нашего основного пакета:

```
library(shiny)
```

ПРОВЕРКА ЗНАЧЕНИЙ

Первое, о чем пользователю можно и нужно сообщать, – так это о том, что он ввел некорректные значения в поля ввода. Это похоже на написание хороших и правильных функций в R, которые в удобном виде оповещают разработчика о том, какие аргументы должны быть переданы на вход и что вы сделали не так. Предположения о том, насколько неправильно пользователь

может обращаться с вашим приложением, позволят вам создать полезные информативные подсказки, вместо того чтобы заставлять пользователя пробираться через дебри непонятных сообщений об ошибках.

Проверка ввода

Пакет *shinyFeedback* великолепно подходит для обеспечения необходимой обратной связи с пользователем. Использование этого пакета состоит из двух последовательных шагов. Сначала необходимо добавить в интерфейс пользователя функцию *useShinyFeedback()*. Это приведет к созданию соответствующей разметки HTML и кода на языке JavaScript для сообщений об ошибках:

```
ui <- fluidPage(
  shinyFeedback::useShinyFeedback(),
  numericInput("n", "n", value = 10),
  textOutput("half")
)
```

Затем в функции *server()* вы вызываете одну из функций обратной связи: *feedback()*, *feedbackWarning()*, *feedbackDanger()* или *feedbackSuccess()*. У этих функций есть три ключевых аргумента:

- *inputId* – идентификатор элемента ввода, для которого добавляется проверка;
- *show* – логическое значение, определяющее, показывать проверку или нет;
- *text* – текст для отображения.

Также этим функциям можно передать аргументы *color* и *icon*, позволяющие настроить внешний вид сообщений. За более подробной информацией можете обратиться к документации.

Давайте посмотрим на примере, как это все работает. Допустим, нам нужно осуществлять проверку на ввод четных чисел:

```
server <- function(input, output, session) {
  half <- reactive({
    even <- input$n %% 2 == 0
    shinyFeedback::feedbackWarning("n", !even, "Please select an even number")
    input$n / 2
  })
  output$half <- renderText(half())
}
```

На рис. 8.1 показан результат работы приложения.

Обратите внимание, что, хоть ошибка и отображается, значение элемента вывода все равно обновляется. Обычно такое поведение приложения не приветствуется, поскольку ошибочные значения могут только сбивать с толку пользователей. Чтобы не позволить элементу ввода повлечь за собой реактивные изменения, вам потребуется использовать функцию *req()* (сокращенно от *required* – требуемый). Это может выглядеть так:

```
server <- function(input, output, session) {
  half <- reactive({
    even <- input$n %% 2 == 0
    shinyFeedback::feedbackWarning("n", !even, "Please select an even number")
    req(even)
    input$n / 2
  })

  output$half <- renderText(half())
}
```

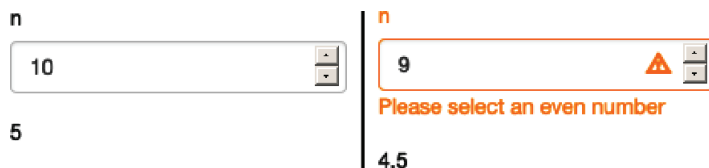


Рис. 8.1 ❖ Функция `feedbackWarning()` помогает отобразить ошибку в случае ошибочного ввода. На рисунке слева показан корректный ввод, а справа – ошибочный. Посмотреть приложение онлайн можно по адресу <https://hadley.shinyapps.io/ms-feedback/>

Если входящий аргумент `req()` не истина, функция посылает Shiny специальный сигнал о том, что реактивное выражение получило не все требуемые входные данные и должно быть *приостановлено* (paused). Мы сделаем небольшое отступление и поговорим об этом более обстоятельно, прежде чем вернемся к использованию `req()` совместно с функцией `validate()`.

Отмена выполнения при помощи функции `req()`

Проще всего понять работу функции `req()` будет за пределами операции проверки ввода. Как вы могли заметить, сразу после запуска приложения происходит выполнение всего реактивного графика – еще до любых действий от пользователя. С этим нет никаких проблем, если вы можете задать для элементов ввода соответствующие значения по умолчанию. Но это не всегда возможно, и иногда вам просто необходимо дождаться каких-то действий от пользователя, прежде чем запустится цепочка вычислений. Это бывает, например, в следующих случаях:

- если для элемента ввода `textInput()` установлено пустое значение по умолчанию (`value = ""`) и вы не хотите ничего предпринимать, пока пользователь не введет какое-либо значение;
- если в списочном элементе ввода `selectInput()` присутствует пустой элемент в аргументе выбора `choice` и вы не можете ничего сделать, пока пользователь не выберет из списка другой элемент;
- при наличии файлового элемента ввода `fileInput()` с пустым результатом – пока пользователь ничего не загрузил (подробнее мы будем говорить об этом в следующей главе).

Таким образом, нам необходим какой-то механизм для остановки реактивных выражений, чтобы ничего не происходило до выполнения определенного условия. Именно эти действия берет на себя функция *req()* – она проверяет требуемые значения перед продолжением работы реактивного поставщика.

Рассмотрим следующее приложение, которое будет приветствовать пользователя на двух языках: английском и маори. Если запустить его, вы увидите ошибку, показанную на рис. 8.2, поскольку в векторе *greetings* нет значения, соответствующего выбору в списке по умолчанию (пустой строке):

```
ui <- fluidPage(
  selectInput("language", "Language", choices = c("", "English", "Maori")),
  textInput("name", "Name"),
  textOutput("greeting")
)

server <- function(input, output, session) {
  greetings <- c(
    English = "Hello",
    Maori = "Ki ora"
  )
  output$greeting <- renderText({
    paste0(greetings[[input$language]], " ", input$name, "!")
  })
}
```

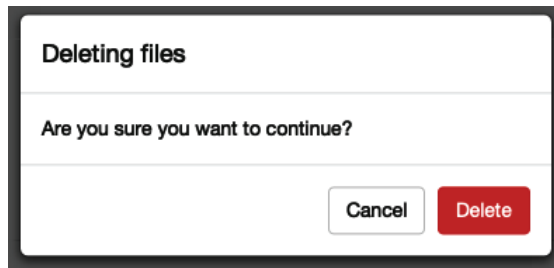


Рис. 8.2 ❖ Приложение выводит неинформативную ошибку после запуска из-за отсутствия выбора языка приветствия

Избавиться от этой проблемы можно, используя функцию *req()*. Теперь программа не будет продолжать свое выполнение, пока пользователь не выберет язык и не введет имя, как показано в следующем фрагменте кода и на рис. 8.3:

```
server <- function(input, output, session) {
  greetings <- c(
    English = "Hello",
    Maori = "Ki ora"
  )
  output$greeting <- renderText({
```



```

    req(input$language, input$name)
    paste0(greetings[[input$language]], " ", input$name, "!")
  })
}

```

The figure consists of three vertically stacked screenshots of a web application interface. Each screenshot shows a form with two input fields: 'Language' and 'Name'.
 - The first screenshot shows the 'Language' field as a dropdown menu and the 'Name' field as a text input, both of which are empty.
 - The second screenshot is identical to the first, showing the empty form.
 - The third screenshot shows the 'Language' dropdown menu with 'English' selected and the 'Name' text input containing the word 'Hadley'. Below the form, the text 'Hello Hadley!' is displayed.

Рис. 8.3 ❖ Использование функции `req()` позволяет программе дожидаться выбора пользователя перед продолжением выполнения. Посмотреть приложение онлайн можно по адресу <https://hadley.shinyapps.io/ms-require-simple2/>

Фактически функция `req()` посылает приложению *специальное сообщение*¹ (*condition*). Это специальное сообщение приводит к остановке выполнения всех зависимых реактивных выражений и элементов вывода. Чисто технически все нижестоящие реактивные потребители просто становятся недействительными. В главе 16 мы еще вернемся к этой терминологии.

Реализация функции `req()` предполагает, что выражение `req(input$x)` будет выполнено только в случае предоставления пользователем входного значе-

¹ Термин *специальные сообщения* используется в R для описания целой группы событий, включая *ошибки* (*error*), *предупреждения* (*warning*) и *сообщения* (*message*). За дополнительной информацией вы можете обратиться к главе 8 книги *Advanced R* по адресу <https://adv-r.hadley.nz/conditions.html>.

ния для элемента ввода вне зависимости от типа элемента¹. При необходимости вы можете использовать функцию `req()` и со своими собственными логическими выражениями. Например, инструкция `req(input$a > 0)` позволит вычислениям продолжаться только в случае, если значение элемента ввода `a` больше нуля. Это типичный пример выполнения проверки в приложениях, как вы увидите далее.

Функция `req()` и проверка данных

Давайте попробуем объединить функцию `req()` с `shinyFeedback` для решения более сложных задач. Вернемся к простейшему виду приложения из первой главы для выбора набора данных и просмотра его содержимого. Я собираюсь немного изменить приложение, заменив элемент ввода `selectInput()` на `textInput()`. Клиентская часть при этом значительно не изменится и будет выглядеть следующим образом:

```
ui <- fluidPage(
  shinyFeedback::useShinyFeedback(),
  textInput("dataset", "Dataset name"),
  tableOutput("data")
)
```

Что касается серверной функции, то она претерпит существенные изменения. В частности, мы будем использовать функцию `req()` двумя способами:

- мы хотим, чтобы вычисления продолжались только в случае ввода пользователем имени набора данных, поэтому использовали в начале реактивного выражения инструкцию `req(input$dataset)`;
- затем мы проверяем, существует ли набор данных с указанным именем. Если нет, выводим сообщение об ошибке и используем функцию `req()` для отмены дальнейших вычислений. При этом мы передали в качестве аргумента функции `cancelOutput` значение `TRUE`. Обычно отмена реактивного выражения приводит к сбросу значений всех зависимых элементов вывода, а аргумент `cancelOutput = TRUE` позволяет им сохранить последний успешный вывод. Это особенно важно при использовании текстового поля, поскольку оно может посылать сигналы на обновление зависимых элементов даже в процессе ввода текста.

Результат работы приложения показан на рис. 8.4.

```
server <- function(input, output, session) {
  data <- reactive({
    req(input$dataset)

    exists <- exists(input$dataset, "package:datasets")
    shinyFeedback::feedbackDanger("dataset", !exists, "Unknown dataset")
  })
}
```

¹ Если быть максимально точными, то функция `req()` выполняется только в случае действительности или *истинности* (truthy) соответствующих элементов ввода, то есть при условии, что их значения не равны `FALSE`, `NULL`, `""` и другим особым вариантам, перечисленным в справке `?isTruthy`.

```

    req(exists, cancelOutput = TRUE)

    get(input$dataset, "package:datasets")
  })

  output$data <- renderTable({
    head(data())
  })
}

```

Dataset name

Dataset name

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.10	3.50	1.40	0.20	setosa
4.90	3.00	1.40	0.20	setosa
4.70	3.20	1.30	0.20	setosa
4.60	3.10	1.50	0.20	setosa
5.00	3.60	1.40	0.20	setosa
5.40	3.90	1.70	0.40	setosa

Dataset name

Unknown dataset

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.10	3.50	1.40	0.20	setosa
4.90	3.00	1.40	0.20	setosa
4.70	3.20	1.30	0.20	setosa
4.60	3.10	1.50	0.20	setosa
5.00	3.60	1.40	0.20	setosa
5.40	3.90	1.70	0.40	setosa

Рис. 8.4 ❖ При запуске приложения таблица остается пустой, поскольку мы не ввели имя набора данных. Таблица заполняется данными при вводе корректного имени набора (*iris*) и сохраняет заполнение при нажатии на клавишу **Backspace** для ввода другого имени.

Посмотреть приложение онлайн можно по адресу
<https://hadley.shinyapps.io/ms-require-cancel>

Проверка элементов вывода

Пакет *shinyFeedback* прекрасно справляется со своими обязанностями, когда речь идет о единственном элементе ввода. Но иногда неправильный вывод является следствием комбинации некорректного ввода сразу в нескольких полях. В этих случаях нет никакого смысла размещать сообщение об ошибке рядом с элементами ввода (кто из них больше виноват – неизвестно). Гораздо логичнее будет соответствующим образом пометить элемент вывода.

Это можно сделать при помощи функции *validate()*, встроенной в Shiny. При вызове внутри реактивного выражения или элемента вывода функция *validate(message)* останавливает выполнение дальнейшего кода и отображает переданное сообщение рядом со всеми зависимыми элементами вывода. Ниже показан код приложения, в котором мы предотвращаем расчет квадратного корня и логарифма в случае ввода отрицательного числа:

```
ui <- fluidPage(
  numericInput("x", "x", value = 0),
  selectInput("trans", "transformation",
    choices = c("square", "log", "square-root")
  ),
  textOutput("out")
)

server <- function(input, output, server) {
  output$out <- renderText({
    if (input$x < 0 && input$trans %in% c("log", "square-root")) {
      validate("x can not be negative for this transformation")
    }

    switch(input$trans,
      square = input$x ^ 2, "square-root" = sqrt(input$x),
      log = log(input$x)
    )
  })
}
```

Результат работы приложения показан на рис. 8.5.

State	x	transformation	Output
Correct	0	square	0
Error	-1	log	x can not be negative for this transformation

Рис. 8.5 ❖ При корректном вводе приложение произведет нужные вычисления. Если комбинация значений в элементах ввода не соответствует нашим ожиданиям, пользователь увидит сообщение об ошибке

Оповещения

Если в приложении никакой ошибки не произошло, а вам просто необходимо уведомить пользователя о чем-либо, вы можете использовать *оповещения* (notification). В Shiny оповещения создаются при помощи функции `showNotification()` и выводятся в виде стека в правой нижней части окна. Существует три основных способа использования функции `showNotification()`:

- для вывода временного уведомления, исчезающего через определенный промежуток времени;
- для оповещения о начале и окончании выполнения процесса;
- для вывода единственного оповещения, обновляющегося с течением времени.

В следующих разделах мы подробно рассмотрим все перечисленные техники.

Временные оповещения

Простейший способ использования функции `showNotification()` – ее вызов с единственным аргументом, представляющим собой сообщение для вывода. Поведение приложения в этом случае трудно показать при помощи скриншотов, легче будет открыть его онлайн по адресу <https://hadley.shinyapps.io/ms-notification-transient/> и посмотреть, как оно работает:

```
ui <- fluidPage(
  actionButton("goodnight", "Good night")
)

server <- function(input, output, session) {
  observeEvent(input$goodnight, {
    showNotification("So long")
    Sys.sleep(1)
    showNotification("Farewell")
    Sys.sleep(1)
    showNotification("Auf Wiedersehen")
    Sys.sleep(1)
    showNotification("Adieu")
  })
}
```

По умолчанию сообщения будут исчезать через пять секунд, но это поведение можно переопределить при помощи аргумента `duration`. Кроме того, пользователь может и сам закрыть сообщение до истечения срока его видимости, нажав на соответствующую кнопку. Если вы хотите, чтобы ваше уведомление как-то выделялось, можете присвоить аргументу `type` одно из следующих значений: «*message*», «*warning*» или «*error*»:

```

server <- function(input, output, session) {
  observeEvent(input$goodnight, {
    showNotification("So long")
    Sys.sleep(1)
    showNotification("Farewell", type = "message")
    Sys.sleep(1)
    showNotification("Auf Wiedersehen", type = "warning")
    Sys.sleep(1)
    showNotification("Adieu", type = "error")
  })
}

```

На рис. 8.6 показан внешний вид нашего приложения.

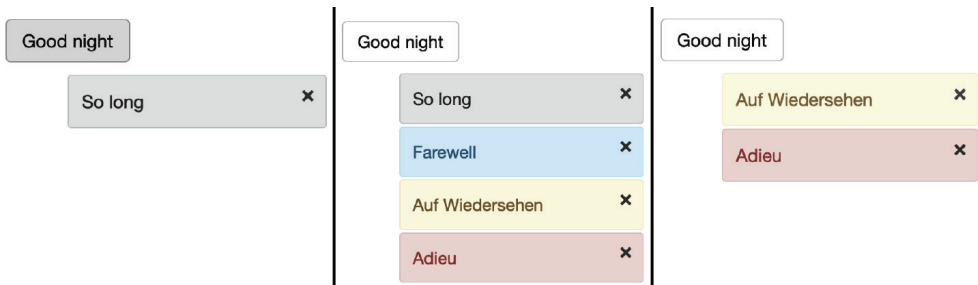


Рис. 8.6 ❖ Последовательность оповещений после нажатия на кнопку. Сначала появляется первое сообщение, через три секунды видны уже все они, после чего уведомления по очереди исчезают. Посмотреть приложение онлайн можно по адресу <https://hadley.shinyapps.io/ms-notify-persistent>

Оповещения о выполнении процесса

Зачастую бывает необходимо привязать оповещения к длительной операции. Допустим, вы хотите, чтобы уведомление появлялось в начале выполнения задачи и исчезало по его окончании. Для этого вам необходимо сделать следующее:

- установить аргументы `duration = NULL` и `closeButton = FALSE`, чтобы оповещение оставалось видимым вплоть до окончания выполнения операции;
- сохранить идентификатор (`id`), возвращенный функцией `showNotification()`, чтобы передать его впоследствии функции `removeNotification()`. Наиболее надежный способ сделать это – использовать функцию `on.exit()`, гарантирующую, что оповещение будет закрыто вне зависимости от того, как завершится операция (успешно или с ошибкой). Чтобы больше узнать о функции `on.exit()`, прочитайте статью *Changing and Restoring State* по адресу <https://withr.r-lib.org/articles/changing-and-restoring-state.html>.

В следующем фрагменте кода показано, как можно оповестить пользователя о загрузке большого файла CSV¹:

```
server <- function(input, output, session) {
  data <- reactive({
    id <- showNotification("Reading data...", duration = NULL, closeButton = FALSE)
    on.exit(removeNotification(id), add = TRUE)

    read.csv(input$file$datapath)
  })
}
```

Обычно такие оповещения прописываются внутри реактивных выражений, чтобы длительные операции могли выполняться только при необходимости.

Обновляемые оповещения

Как вы видели в первом примере этого раздела, множественные вызовы функции `showNotification()` могут приводить к появлению на экране сразу нескольких оповещений. Но вы всегда можете перехватить идентификатор (`id`) первого вызова функции `showNotification()` и использовать его впоследствии для обновления текста оповещения:

```
ui <- fluidPage(
  tableOutput("data")
)

server <- function(input, output, session) {
  notify <- function(msg, id = NULL) {
    showNotification(msg, id = id, duration = NULL, closeButton = FALSE)
  }

  data <- reactive({
    id <- notify("Reading data...")
    on.exit(removeNotification(id), add = TRUE)
    Sys.sleep(1)

    notify("Reticulating splines...", id = id)
    Sys.sleep(1)

    notify("Herding llamas...", id = id)
    Sys.sleep(1)

    notify("Orthogonalizing matrices...", id = id)
    Sys.sleep(1)
  })
}
```

¹ Если чтение файлов CSV является узким местом в вашем приложении, подумайте об использовании функций `data.table::fread()` и `vroom::vroom()`; они могут работать на порядок быстрее, чем `read.csv()`.

```

      mtcars
    })

    output$data <- renderTable(head(data()))
  }

```

Такое использование оповещений бывает удобно, если ваша длительная операция может быть разбита на стадии. Посмотреть на это приложение онлайн можно по адресу <https://hadley.shinyapps.io/ms-notification-updates>.

ИНДИКАТОР ХОДА ВЫПОЛНЕНИЯ ЗАДАЧИ

Для продолжительных операций ничего лучше *индикатора хода выполнения задачи* (progress bar) еще не придумали. Помимо того, что такие индикаторы помогают понять, в каком месте операции вы находитесь в данный момент, они еще и позволяют примерно определить, сколько времени осталось до завершения процесса: успеете ли вы только глазом моргнуть, сможете ли налить чашечку кофе или вам лучше сразу отправляться спать. В данном разделе мы рассмотрим две техники отображения индикатора выполнения: одна из них будет использована с применением встроенных средств Shiny, а вторая – с помощью пакета *waiter* (<https://waiter.john-coene.com/#>), разработанного Джоном Козном.

К сожалению, обе техники, которые здесь будут показаны, страдают от одного и того же недостатка. Чтобы использовать индикатор выполнения, необходимо прежде всего иметь возможность разбить общую задачу на составляющие отрезки, приблизительно равные по длительности. Но это бывает сделать проблематично, особенно с учетом того, что сопутствующий код часто написан на С, и контролировать обновления прогресса бывает невозможно. Мы работаем над инструментарием пакета *progress* (<https://github.com/r-lib/progress>), чтобы пакеты вроде *dplyr*, *readr* и *vroom* получили возможность оповещать пользователей о работе при помощи индикаторов выполнения, которые можно было бы перенести в приложения Shiny.

Shiny

Для создания индикатора прогресса в Shiny необходимо воспользоваться функциями *withProgress()* и *incProgress()*. Представьте, что у вас есть фрагмент медленно работающего кода вроде показанного ниже¹:

```

for (i in seq_len(step)) {
  x <- function_that_takes_a_long_time(x)
}

```

¹ Если в вашем коде не используются циклы *for* или функции *apply/map*, вам будет очень проблематично встроить в свое приложение индикатор выполнения.

Начнем с заключения этого блока кода в функцию `withProgress()`. Это поможет отобразить индикатор выполнения в начале задачи и скрыть – в конце:

```
withProgress({
  for (i in seq_len(step)) {
    x <- function_that_takes_a_long_time(x)
  }
})
```

Затем, после каждой итерации, вам необходимо вызывать функцию `incProgress()` следующим образом:

```
withProgress({
  for (i in seq_len(step)) {
    x <- function_that_takes_a_long_time(x)
    incProgress(1 / length(step))
  }
})
```

Первым параметром в функцию `incProgress()` передается инкремент для приращения индикатора. По умолчанию шкала индикатора начинается на отметке 0 и заканчивается на 1, так что приращение в виде единицы, деленной на количество шагов, обеспечит заполнение индикатора к концу цикла.

Ниже показана полная реализация индикатора выполнения в виде приложения. А результат работы программы продемонстрирован на рис. 8.7:

```
ui <- fluidPage(
  numericInput("steps", "How many steps?", 10),
  actionButton("go", "go"),
  textOutput("result")
)

server <- function(input, output, session) {
  data <- eventReactive(input$go, {
    withProgress(message = "Computing random number", {
      for (i in seq_len(input$steps)) {
        Sys.sleep(0.5)
        incProgress(1 / input$steps)
      }
      runif(1)
    })
  })

  output$result <- renderText(round(data(), 2))
}
```

Дадим лишь пару замечаний:

- я использовал необязательный аргумент `message` для снабжения нашего индикатора выполнения текстовым описанием;
- я воспользовался функцией `Sys.sleep()` для симуляции выполнения длительной операции. В реальном коде здесь будет настоящая функция, требующая времени;

- я позволил пользователю самому запустить длительную операцию, воспользовавшись функцией `eventReactive()`. Это хорошая практика, когда дело касается выполнения продолжительных задач.

How many steps?

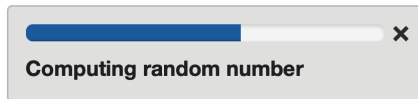


Рис. 8.7 ❖ Индикатор помогает понять ход выполнения процесса.
Проверить работу этого приложения онлайн можно по адресу
<https://hadley.shinyapps.io/ms-progress>

Waiter

Встроенный индикатор выполнения очень неплохо подходит для простых базовых задач, но если вы хотите получить больше визуальных эффектов, попробуйте воспользоваться пакетом *waiter* (<https://waiter.john-coene.com>). Адаптировать наш предыдущий фрагмент кода для использования этого пакета очень просто. Для этого достаточно добавить в функцию пользовательского интерфейса строку `use_waitress()`, как показано ниже:

```
ui <- fluidPage(
  waiter::use_waitress(),
  numericInput("steps", "How many steps?", 10),
  actionButton("go", "go"),
  textOutput("result")
)
```

Интерфейс для работы с индикаторами выполнения из пакета *waiter* немного отличается от того, что мы видели ранее. Пакет *waiter* использует *объект R6* для объединения всех необходимых для работы функций. Если вы никогда не использовали объекты *R6*, не беспокойтесь. Вам достаточно будет скопировать приведенный ниже шаблон. Базовый жизненный цикл выглядит следующим образом:

```
# Создаем новый индикатор выполнения
waitress <- waiter::Waitress$new(max = input$steps)
# Автоматически закрываем его по выполнении задачи
on.exit(waitress$close())

for (i in seq_len(input$steps)) {
```

```

    Sys.sleep(0.5)
    # Инкремент на один шаг
    waitress$inc(1)
  }

```

В приложении Shiny этот шаблон можно использовать следующим образом:

```

server <- function(input, output, session) {
  data <- eventReactive(input$go, {
    waitress <- waiter::Waitress$new(max = input$steps)
    on.exit(waitress$close())

    for (i in seq_len(input$steps)) {
      Sys.sleep(0.5)
      waitress$inc(1)
    }

    runif(1)
  })

  output$result <- renderText(round(data(), 2))
}

```

По умолчанию индикатор будет выглядеть как тонкая полоска в верхней части страницы, но вы всегда можете изменить его внешний вид. Для этого необходимо переопределить аргумент `theme`, присвоив ему одно из следующих значений:

- `overlay` – непрозрачный индикатор, скрывающий всю страницу;
- `overlay-opacity` – полупрозрачный индикатор, скрывающий всю страницу;
- `overlay-percent` – непрозрачный индикатор с отображением процента выполнения.

Если вы не хотите, чтобы индикатор выполнения занимал всю страницу, можете наложить его на конкретный элемент ввода или вывода, передав параметр `selector` следующим образом:

```
waitress <- Waitress$new(selector = "#steps", theme = "overlay")
```

Вращающийся индикатор прогресса (спиннер)

Иногда вы не можете знать заранее, сколько продлится та или иная операция. В таких случаях обычно пользователю показывают *вращающийся индикатор прогресса* (спиннер или песочные часы), говорящий о том, что процесс выполняется. Для отображения такого индикатора вы также можете использовать упомянутый выше пакет `waiter`, переключившись с объекта `Waitress` на `Waiter`:

```

ui <- fluidPage(
  waiter::use_waiter(),

```

```

    actionButton("go", "go"),
    textOutput("result")
  )
server <- function(input, output, session) {
  data <- eventReactive(input$go, {
    waiter <- waiter::Waiter$new()
    waiter$show()
    on.exit(waiter$hide())

    Sys.sleep(sample(5, 1))
    runif(1)
  })
  output$result <- renderText(round(data(), 2))
}

```

На рис. 8.8 показан внешний вид индикатора в приложении.

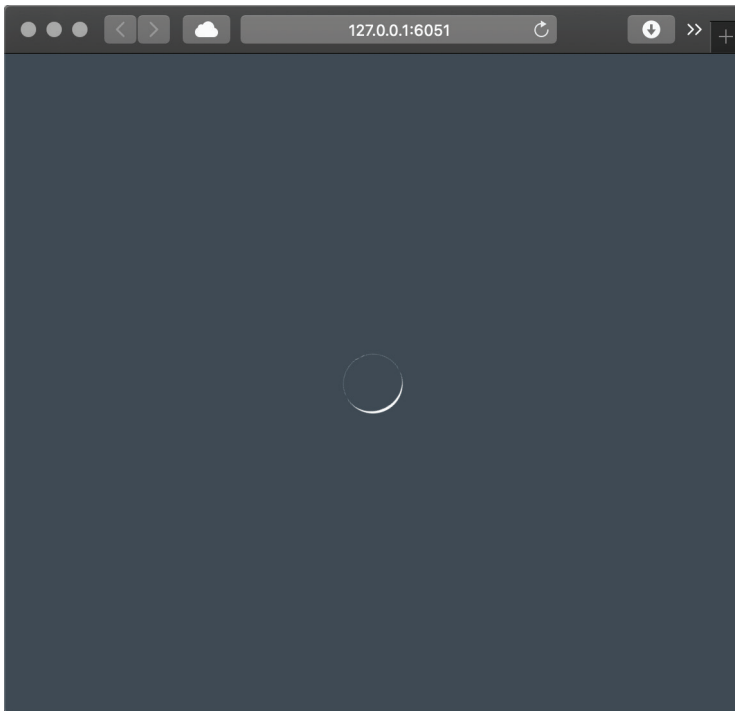


Рис. 8.8 ❖ Пакет *waiter* позволяет отображать вращающийся индикатор прогресса.
Посмотреть приложение с таким индикатором онлайн
можно по адресу <https://hadley.shinyapps.io/ms-spinner-1/>

Как и *Waitress*, вы можете использовать объекты *Waiter* применительно к конкретному элементу вывода. В этом случае вращающийся индикатор будет исчезать после обновления элемента, а код будет выглядеть еще проще:

```

ui <- fluidPage(
  waiter::use_waiter(),
  actionButton("go", "go"),
  plotOutput("plot"),
)

server <- function(input, output, session) {
  data <- eventReactive(input$go, {
    waiter::Waiter$new(id = "plot")$show()

    Sys.sleep(3)
    data.frame(x = runif(50), y = runif(50))
  })

  output$plot <- renderPlot(plot(data()), res = 96)
}

```

Результат работы приложения показан на рис. 8.9.

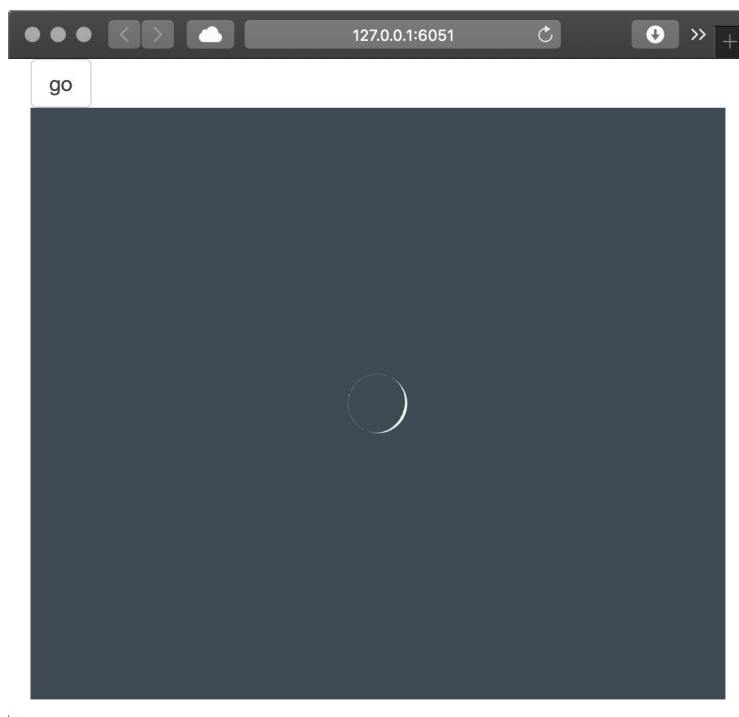


Рис. 8.9 ❖ Вы можете отображать вращающийся индикатор для конкретного элемента. Посмотреть приложение можно по адресу <https://hadley.shinyapps.io/ms-spinner-2>

Пакет *waiter* предлагает на выбор большое разнообразие вращающихся индикаторов. Посмотреть их все можно, запросив подсказку `?waiter::spinners`, а чтобы применить понравившийся, используйте, к примеру, инструкцию `Waiter$new(html = spin_ripple())`.

Еще более простой альтернативой отображения вращающихся индикаторов прогресса является использование пакета *shinycssloaders* (<https://github.com/daattali/shinycssloaders>) от Дина Аттали (Dean Attali). В данном пакете используется JavaScript для отслеживания событий Shiny, так что вам даже не понадобится писать код на серверной стороне. Вместо этого вы оборачиваете в функцию `shinycssloaders::withSpinner()` элементы вывода, для которых хотите отображать вращающиеся индикаторы при переходе в недействительное состояние:

```
library(shinycssloaders)

ui <- fluidPage(
  actionButton("go", "go"),
  withSpinner(plotOutput("plot")),
)

server <- function(input, output, session) {
  data <- eventReactive(input$go, {
    Sys.sleep(3)
    data.frame(x = runif(50), y = runif(50))
  })

  output$plot <- renderPlot(plot(data()), res = 96)
}
```

ПОДТВЕРЖДЕНИЕ И ОТМЕНА ДЕЙСТВИЙ

Иногда при выполнении пользователем критических или необратимых действий уместно дать ему возможность подтвердить свой выбор или отменить его. Давайте рассмотрим три разные техники, которые позволят вам реализовать описанный функционал.

Явное подтверждение

Простейший способ дать возможность пользователю одуматься и отменить выполняемое действие – показать ему дополнительное диалоговое окно с подтверждением выбора. В Shiny такое окно можно создать при помощи функции *modalDialog()*. Диалоговое окно будет открыто в модальном режиме, то есть будет создан новый режим взаимодействия с пользователем, и вернуться в основное приложение без закрытия диалогового окна он не сможет.

Представьте, что ваше приложение выполняет удаление файлов из папки, строки из базы данных и т. д. Подобные действия трудно обратить, а значит, будет неплохо запросить у пользователя дополнительное подтверждение его действий. Вы можете создать диалоговое окно, показанное на рис. 8.10, при помощи следующего кода:

```
modal_confirm <- modalDialog(
  "Are you sure you want to continue?",
  title = "Deleting files",
  footer = tagList(
    actionButton("cancel", "Cancel"),
    actionButton("ok", "Delete", class = "btn btn-danger")
  )
)
```

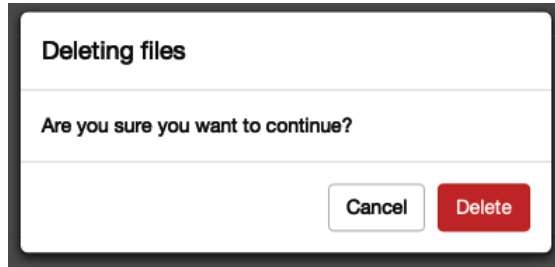


Рис. 8.10 ❖ Диалоговое окно, подтверждающее намерение удалить файлы

Стоит отдельно сказать о небольших, но важных нюансах при создании таких диалоговых окон:

- как называть кнопки? Лучше использовать описательные названия для кнопок и избегать коротких Да/Нет, Продолжить/Отменить;
- как выстраивать кнопки? Размещать ли первой кнопку отмены действия, как на Mac, или его подтверждения, как на Windows? Желательно выбрать вариант, подходящий большинству пользователей, которые предположительно будут работать с вашим приложением;
- как дать понять, что выполняемое действие является опасным или критическим? В показанном выше приложении я использовал особый стиль для кнопки подтверждения действия при помощи аргумента `class = "btn btn-danger"`.

Якоб Нильсен (Jakob Nielsen) в своей статье, расположенной по адресу <https://www.nngroup.com/articles/ok-cancel-or-cancel-ok>, дает хорошие советы по оформлению диалоговых окон в приложениях.

Давайте используем созданное ранее диалоговое окно в настоящем, пусть и довольно простом приложении. В нашем интерфейсе пользователя будет всего одна кнопка для удаления файлов:

```
ui <- fluidPage(
  actionButton("delete", "Delete all files?")
)
```

В функции `server()` мы реализуем следующие две идеи:

- будем использовать функции `showModal()` и `removeModal()` для отображения и закрытия диалогового окна;

- будем перехватывать события, генерируемые пользовательским интерфейсом диалогового окна `modal_confirm`. Эти объекты не создаются в нашем интерфейсе статически, а добавляются динамически в функции `server()` посредством вызова функции `showModal()`:

```
server <- function(input, output, session) {
  observeEvent(input$delete, {
    showModal(modal_confirm)
  })

  observeEvent(input$ok, {
    showNotification("Files deleted")
    removeModal()
  })

  observeEvent(input$cancel, {
    removeModal()
  })
}
```

Более подробно эту идею мы рассмотрим в главе 10.

Отмена действия

Явное подтверждение, описанное ранее, требуется только при выполнении критических действий, которые составляют не такой большой процент всех действий в приложениях. Вы должны избегать этого подхода при выполнении пользователями обычных рутинных действий, чтобы снизить количество ошибок. К примеру, эта техника не подойдет для отправки сообщений в Twitter. Если показывать пользователям диалоговое окно с подтверждением каждый раз, когда они хотят опубликовать сообщение, вы вскоре заметите, что они жмут кнопку подтверждения своего действия чисто автоматически, а потом, когда через десять секунд заметят опечатку в своем сообщении, начинают жалеть об этом.

В таких случаях лучше перед выполнением действия давать пользователю несколько секунд, в течение которых у него будет возможность отменить операцию. По своей сути это не является отменой действия, поскольку оно фактически еще не выполнилось, но пользователи привыкли к этому слову, так что лучше продолжать его использовать.

Проиллюстрируем описанный подход на примере сайта для отправки сообщений в Twitter с возможностью их отмены. Пользовательский интерфейс у нас будет очень простым и будет состоять из текстового поля и кнопки для отправки сообщения:

```
ui <- fluidPage(
  textAreaInput("message",
    label = NULL,
    placeholder = "What's happening?",
    rows = 3
```



```
),  
  actionBar("tweet", "Tweet")  
)
```

Что касается серверной функции, то она будет намного более сложной, и в ней мы используем технику, о которой раньше не говорили. Не беспокойтесь, если не поймете весь код досконально. Сосредоточьтесь на базовой идее, состоящей в использовании специальных аргументов при вызове функции *observeEvent()* для запуска определенного кода через несколько секунд. Основной же замысел состоит в том, чтобы перехватывать результат работы функции *observeEvent()* и сохранять его в переменной. Это позволит нам впоследствии отменить действие наблюдателя, чтобы данные в Twitter отправлены не были:

```
runLater <- function(action, seconds = 3) {  
  observeEvent(  
    invalidateLater(seconds * 1000), action,  
    ignoreInit = TRUE,  
    once = TRUE,  
    ignoreNULL = FALSE,  
    autoDestroy = FALSE  
  )  
}  
  
server <- function(input, output, session) {  
  waiting <- NULL  
  last_message <- NULL  
  
  observeEvent(input$tweet, {  
    notification <- glue::glue("Tweeted '{input$message}')"  
    last_message <- input$message  
    updateTextAreaInput(session, "message", value = "")  
  
    showNotification(  
      notification,  
      action = actionBar("undo", "Undo?"),  
      duration = NULL,  
      closeButton = FALSE,  
      id = "tweeted",  
      type = "warning"  
    )  
  
    waiting <- runLater({  
      cat("Actually sending tweet...\n")  
      removeNotification("tweeted")  
    })  
  })  
  
  observeEvent(input$undo, {  
    waiting$destroy()  
    showNotification("Tweet retracted", id = "tweeted")  
    updateTextAreaInput(session, "message", value = last_message)  
  })  
}
```

Увидеть приложение в действии можно по адресу <https://hadley.shinyapps.io/ms-undo>.

Корзина

Для действий, о которых пользователь может пожалеть по прошествии нескольких дней, можно организовать нечто вроде привычной *корзины* на вашем компьютере. Например, удаление файла может приводить не к его физическому уничтожению, а к переносу в так называемую *резервную ячейку* (holding cell), удаление из которой может потребовать дополнительных действий. Это что-то вроде продвинутой кнопки отмены действия с возможностью перенестись в прошлое спустя несколько дней. Одновременно это похоже и на подтверждение действия – просто вам нужно выполнить два не зависящих друг от друга действия, чтобы удалить файл безвозвратно.

Главным недостатком этой техники является ее значительная сложность в реализации, заключающаяся в необходимости поддержания резервной ячейки с информацией для возможной отмены предпринятого пользователем действия. К тому же для поддержания актуальности и во избежание излишнего накопления данных вам потребуется постоянное вмешательство со стороны пользователя. По этим причинам подобная техника реализуется далеко не во всех приложениях Shiny, а лишь в самых сложных, а значит, нет необходимости останавливаться на ней в данной книге более подробно.

ЗАКЛЮЧЕНИЕ

В данной главе мы рассмотрели разные приемы взаимодействия с пользователем на предмет его оповещения о происходящем в приложении. В общем случае все эти техники являются необязательными, но без них ваше задумчивое приложение может просто начать нервировать пользователя. Если приложением пользуетесь вы один, это можно как-то пережить, но чем больше людей работает с ним, тем более дружелюбным должен быть интерфейс в плане оповещения о выполнении длительных операций.

В следующей главе мы будем говорить о передаче файлов пользователю и от него.

Глава 9

Загрузка и скачивание файлов

Передача файлов (transferring files) в приложение и из приложения является очень распространенной задачей. Этот функционал можно использовать для загрузки данных для анализа или скачивания результатов в виде набора данных или отчета. В данной главе мы рассмотрим различные компоненты интерфейса пользователя и серверной части, необходимые для осуществления передачи файлов. А начнем, как и всегда, с загрузки пакета Shiny:

```
library(shiny)
```

ЗАГРУЗКА

Для начала обсудим варианты *загрузки* (upload) файлов из приложения с перечислением клиентских и серверных компонентов, после чего попытаемся применить полученные знания в простом приложении.

Интерфейс пользователя

Настройка поддержки передачи файлов в пользовательском интерфейсе реализуется при помощи элемента ввода `fileInput()`, как показано ниже:

```
ui <- fluidPage(  
  fileInput("upload", "Upload a file")  
)
```

Как и в случае с большинством других элементов ввода, элемент `fileInput` принимает на вход два обязательных аргумента: `id` и `label`. Аргументы `width`, `buttonLabel` и `placeholder` позволят вам настроить внешний вид элемента. Мы не будем здесь касаться этих аргументов, но вы можете больше узнать о них в справке, выполнив инструкцию `?fileInput`.

Серверная часть

Управление элементом `fileInput()` в функции `server` осуществляется чуть более сложно по сравнению с другими элементами ввода. Большинство элементов ввода возвращают простой вектор, тогда как `fileInput()` возвращает датафрейм с четырьмя следующими столбцами:

- `name` – оригинальное имя файла на компьютере пользователя;
- `size` – размер файла в байтах. По умолчанию пользователь может загружать файлы размером до 5 Мб. Вы можете изменить этот порог путем обновления опции `shiny.maxRequestSize` перед стартом Shiny. К примеру, чтобы увеличить ограничение до 10 Мб, выполните инструкцию `options(shiny.maxRequestSize = 10 * 1024^2)`;
- `type` – MIME-тип файла¹. MIME представляет собой формальную спецификацию типа файла, зачастую извлекаемую из его расширения и редко необходимую для приложений Shiny;
- `datapath` – путь, по которому была осуществлена загрузка файла на сервере. Особенно полагаться на этот путь не стоит: при загрузке пользователем очередных файлов этот путь может стать недействительным. Данные всегда сохраняются во временной директории под временными именами.

Полагаю, легче всего разобраться с этой структурой данных будет на конкретном примере. Запустите следующий код и загрузите несколько файлов, чтобы понять, какие данные и в каком виде возвращает Shiny:

```
ui <- fluidPage(
  fileInput("upload", NULL, buttonLabel = "Upload...", multiple = TRUE),
  tableOutput("files")
)

server <- function(input, output, session) {
  output$files <- renderTable(input$upload)
}
```

На рис. 9.1 приведены результаты после загрузки пары фотографий со щенками.

Обратите внимание, как я изменил внешний вид элемента при помощи аргументов `label` и `buttonLabel`, а также разрешил одновременную загрузку нескольких файлов, передав в качестве аргумента `multiple` значение `TRUE`.

¹ MIME расшифровывается как *multipurpose internet mail extensions* (многоцелевые расширения почты интернет). Как ясно из названия, изначально этот термин был введен для почтовых систем, но сегодня используется во множестве сетевых инструментов. MIME-тип выражается как тип/подтип. Вот некоторые распространенные примеры таких типов: `text/csv`, `text/html`, `image/png`, `application/pdf`, `application/vnd.ms-excel` (файл Excel).

Upload...	2 files		
Upload complete			
name	size	type	datapath
eoqnr8ikwFE.jpg	183186	image/jpeg	/tmp/RtmpdtmeUj/022d21f6d4342af5aa0cfd39/0.jpg
KCdYn0xu2fU.jpg	116343	image/jpeg	/tmp/RtmpdtmeUj/022d21f6d4342af5aa0cfd39/1.jpg

Рис. 9.1 ❖ Данные, возвращаемые Shiny в ответ на загрузку файлов.
Посмотреть приложение онлайн и опробовать его в действии
можно по адресу <https://hadley.shinyapps.io/ms-upload/>

Загрузка данных

Если пользователь выполняет загрузку набора данных, стоит помнить о двух важных моментах:

- при запуске страницы `input$upload` инициализируется значением `NULL`, так что вам необходимо будет написать инструкцию `req(input$file)`, чтобы было выполнено ожидание окончания загрузки первого файла;
- аргумент асепт позволяет ограничить выбор файлов. Простейший вариант – передать в качестве этого аргумента символьный вектор с допустимыми расширениями файлов, например `accept = ".csv"`. В то же время значение аргумента асепт является лишь рекомендацией для браузера и не всегда срабатывает, так что нелишним будет выполнять надлежащую проверку вручную. О технике проверки данных мы говорили в главе 8. Чтобы извлечь расширение файла в R, можно, к примеру, воспользоваться функцией `tools::file_ext()`, но помните, что она возвращает расширение без ведущей точки.

В результате обсуждений мы получили следующее приложение, в котором пользователь имеет возможность загружать файлы с расширениями `.csv` и `.tsv` и на выходе получает первые `n` строк:

```
ui <- fluidPage(
  fileInput("file", NULL, accept = c(".csv", ".tsv")),
  numericInput("n", "Rows", value = 5, min = 1, step = 1),
  tableOutput("head")
)

server <- function(input, output, session) {
  data <- reactive({
    req(input$file)

    ext <- tools::file_ext(input$file$name)
    switch(ext,
      csv = vroom::vroom(input$file$datapath, delim = ","),
      tsv = vroom::vroom(input$file$datapath, delim = "\t"),
      validate("Invalid file; Please upload a .csv or .tsv file")
    )
  })
}
```

```
output$head <- renderTable({
  head(data(), input$n)
})
}
```

Посмотреть приложение онлайн можно по адресу <https://hadley.shinyapps.io/ms-upload-validate/>.

Обратите внимание, что поскольку `multiple = FALSE` (значение по умолчанию), `input$file` будет представлен датафреймом с единственной строкой, а `input$file$name` и `input$file$datapath` будут символьными векторами длиной `length-1`.

СКАЧИВАНИЕ

Теперь обратимся к операции *скачивания* (download) файлов, рассмотрим необходимые для этого клиентские и серверные компоненты, после чего продемонстрируем эти инструменты на примере загрузки пользователем данных и отчетов.

ОСНОВЫ

Как и в случае с загрузкой файлов из приложения, для скачивания клиентский инструментарий богатством выбора не отличается: вам достаточно добавить в интерфейс элементы вывода `downloadButton(id)` или `downloadLink(id)`, чтобы дать пользователю возможность скачивать файлы:

```
ui <- fluidPage(
  downloadButton("download1"),
  downloadLink("download2")
)
```

Результат работы приложения показан на рис. 9.2.



Рис. 9.2 ❖ Кнопка и ссылка для скачивания файлов

Вы можете настраивать внешний вид этих элементов при помощи аргументов `class` и `icon` аналогично элементу `actionButton()`, о котором мы говорили в главе 2.

В отличие от других элементов вывода, элемент `downloadButton()` не применяется совместно с функцией отображения. Вместо этого вы будете использовать функцию `downloadHandler()`, как показано ниже:

```
output$download <- downloadHandler(
  filename = function() {
```

```

    paste0(input$dataset, ".csv")
  },
  content = function(file) {
    write.csv(data(), file)
  }
)

```

Функция `downloadHandler()` имеет два аргумента, в свою очередь тоже являющиеся функциями:

- `filename` – функция без аргументов, возвращающая имя файла в виде строки. Забота этой функции – создать имя файла, которое будет показано пользователю в диалоговом окне загрузки;
- `content` – функция с одним аргументом `file`, представляющим собой путь для сохранения файла. Эта функция предназначена для сохранения файла в месте, известном фреймворку Shiny, чтобы он потом мог переслать файл пользователю.

Этот интерфейс обычным не назовешь, но он позволяет Shiny контролировать местоположение файла (таким образом, он может быть сохранен в защищенном месте), в то время как у вас остается контроль за содержимым файла.

Теперь давайте соберем полученные знания воедино и посмотрим, как можно передавать файлы и отчеты пользователю.

Скачивание данных

В следующем приложении мы продемонстрируем основы скачивания данных, позволив пользователю загружать любой набор данных из пакета *datasets* в виде файла, разделенного знаками табуляции, как показано на рис. 9.3. Лично я рекомендую использовать файлы *.tsv* (разделенные табуляциями) вместо *.csv* (разделенные запятыми) по причине того, что во многих европейских странах запятые используются для разделения целой и десятичной частей в числах (1,23 вместо 1.23). В данном случае запятая не может быть использована для разграничения полей, а вместо этого применяется точка с запятой. Чтобы избежать путаницы, лучше отдавать предпочтение файлам, разделенным знаками табуляции, – по крайней мере, они везде работают одинаково:

```

ui <- fluidPage(
  selectInput("dataset", "Pick a dataset", ls("package:datasets")),
  tableOutput("preview"),
  downloadButton("download", "Download .tsv")
)

server <- function(input, output, session) {
  data <- reactive({
    out <- get(input$dataset, "package:datasets")
    if (!is.data.frame(out)) {
      validate(paste0("'", input$dataset, "' is not a data frame"))
    }
  })
}

```

```
    }
    out
  })

  output$preview <- renderTable({
    head(data())
  })

  output$download <- downloadHandler(
    filename = function() {
      paste0(input$dataset, ".tsv")
    },
    content = function(file) {
      vroom::vroom_write(data(), file)
    }
  )
}
```

Pick a dataset

iris ▼

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.10	3.50	1.40	0.20	setosa
4.90	3.00	1.40	0.20	setosa
4.70	3.20	1.30	0.20	setosa
4.60	3.10	1.50	0.20	setosa
5.00	3.60	1.40	0.20	setosa
5.40	3.90	1.70	0.40	setosa


 Download .tsv

Рис. 9.3 ❖ Приложение позволяет выбрать набор данных, просмотреть его и скачать на диск.
Посмотреть приложение онлайн можно по адресу <https://hadley.shinyapps.io/ms-download-data>

Обратите внимание, как мы использовали функцию `validate()`, чтобы позволить пользователю скачивать только наборы данных, являющиеся датафреймами. Лучше в данном случае было выполнить предварительную фильтрацию выпадающего списка, но мы решили лишний раз продемонстрировать вам работу функции `validate()`.

Скачивание отчетов

Помимо данных, ваши пользователи могут изъявить желание загружать отчеты с обобщенными результатами интерактивных исследований прямо

из приложения Shiny. Это не самая простая задача, поскольку она связана с представлением одних и тех же данных в разных форматах, но такой функционал бывает крайне полезен для серьезных приложений.

Один из способов генерирования таких отчетов состоит в создании параметризованных документов *RMarkdown*, о которых можно почитать по адресу <https://bookdown.org/yihui/rmarkdown/parameterized-reports.html>.

У параметризованного файла *RMarkdown* присутствует поле `params` в метаданных *YAML*:

```
title: My Document
output: html_document
params:
  year: 2018
  region: Europe
  printcode: TRUE
data: file.csv
```

Внутри документа вы можете ссылаться на эти значения, используя элементы списка `params` (например, `params$year`, `params$region`). Значения в метаданных *YAML* установлены по умолчанию. В основном вы будете перепределять их посредством передачи аргумента `params` при вызове функции `rmarkdown::render()`. Это позволяет легко генерировать различные отчеты на основании одного и того же файла *.Rmd*.

Ниже приведен простой пример, демонстрирующий эту технику более детально. Почитать о примере более подробно можно по адресу <https://shiny.rstudio.com/articles/generating-reports.html>. Ключевая идея состоит в вызове функции `rmarkdown::render()` внутри аргумента `content` функции `downloadHandler()`:

```
ui <- fluidPage(
  sliderInput("n", "Number of points", 1, 100, 50),
  downloadButton("report", "Generate report")
)

server <- function(input, output, session) {
  output$report <- downloadHandler(
    filename = "report.html",
    content = function(file) {
      params <- list(n = input$n)

      id <- showNotification(
        "Rendering report...",
        duration = NULL,
        closeButton = FALSE
      )
      on.exit(removeNotification(id), add = TRUE)

      rmarkdown::render("report.Rmd",
        output_file = file,
        params = params,
```

```

    enviro = new.env(parent = globalenv())
  )
}
)
}

```

Если вы хотите получать на выходе другие форматы, просто поменяйте формат вывода в *.Rmd* и не забудьте изменить расширение файла, например на *.pdf*. Посмотреть приложение онлайн можно по адресу <https://hadley.shinyapps.io/ms-download-rmd>. Обычно создание таких отчетов будет занимать как минимум несколько секунд, так что здесь очень уместно будет использовать оповещения пользователя, которые мы обсуждали в главе 8.

Есть также несколько нюансов, о которых важно помнить:

- RMarkdown оперирует в текущей рабочей директории, что может привести к проблемам при развертывании приложения (например, на *shinyapps.io*). Обойти это можно, скопировав отчет во временную директорию во время запуска приложения, т. е. за пределами функции `server`:

```

report_path <- tempfile(fileext = ".Rmd")
file.copy("report.Rmd", report_path, overwrite = TRUE)

```

После этого необходимо заменить название файла "report.Rmd" на переменную `report_path` при вызове функции `rmarkdown::render()`:

```

rmarkdown::render(report_path,
  output_file = file,
  params = params,
  enviro = new.env(parent = globalenv())
)

```

- по умолчанию RMarkdown будет отображать отчеты в текущем процессе, что приведет к наследованию многих настроек от приложения Shiny (таких как загруженные пакеты, опции и т. д.). Для большей надежности я рекомендую вызывать функцию `render()` в отдельной сессии R с помощью пакета *callr*:

```

render_report <- function(input, output, params) {
  rmarkdown::render(input,
    output_file = output,
    params = params,
    enviro = new.env(parent = globalenv())
  )
}

server <- function(input, output) {
  output$report <- downloadHandler(
    filename = "report.html",
    content = function(file) {
      params <- list(n = input$slider)
      callr::r(

```

```

        render_report,
        list(input = report_path, output = file, params = params)
      )
    }
  )
}

```

Файл приложения и отчет вы можете скачать по адресу <https://github.com/hadley/mastering-shiny/tree/master/rmarkdown-report>, находящемуся внутри репозитория GitHub.

Пакет *shinymeta* (<https://github.com/rstudio/shinymeta>) поможет вам с решением проблемы, состоящей в том, что иногда вам необходимо иметь возможность перевести текущее состояние приложения Shiny в вид воспроизводимого отчета, который может быть перезапущен в будущем. На конференции *useR! 2019* Джо Ченг посвятил этому отдельную тему, с которой можно ознакомиться по адресу <https://www.youtube.com/watch?v=5KByRC6eqC8>.

ПРАКТИЧЕСКИЙ ПРИМЕР

Для закрепления пройденного материала рассмотрим небольшой практический пример, в котором загрузим файл, предварительно выбрав разделитель, посмотрим данные, выполним некоторые преобразования с использованием пакета *janitor* (<http://sfirke.github.io/janitor>) от Сэма Фирка (Sam Firke), а затем дадим пользователю возможность скачать его в формате *.tsv*.

Для облегчения понимания работы приложения я использовал макет `sidebarLayout()`, чтобы можно было выделить три основных шага.

1. Загрузка и анализ файла:

```

ui_upload <- sidebarLayout(
  sidebarPanel(
    fileInput("file", "Data", buttonLabel = "Upload..."),
    textInput("delim", "Delimiter (leave blank to guess)", ""),
    numericInput("skip", "Rows to skip", 0, min = 0),
    numericInput("rows", "Rows to preview", 10, min = 1)
  ),
  mainPanel(
    h3("Raw data"),
    tableOutput("preview1")
  )
)

```

2. Очистка файла:

```

ui_clean <- sidebarLayout(
  sidebarPanel(
    checkboxInput("snake", "Rename columns to snake case?"),
    checkboxInput("constant", "Remove constant columns?"),
    checkboxInput("empty", "Remove empty cols?")
  )
)

```

```

    ),
    mainPanel(
      h3("Cleaner data"),
      tableOutput("preview2")
    )
  )
)

```

3. Скачивание файла:

```

ui_download <- fluidRow(
  column(width = 12, downloadButton("download", class = "btn-block"))
)

```

Теперь собираем все части в один макет при помощи функции `fluidPage()`:

```

ui <- fluidPage(
  ui_upload,
  ui_clean,
  ui_download
)

```

Подобное же разделение на операции реализуем и в функции `server`:

```

server <- function(input, output, session) {
  # Загрузка -----
  raw <- reactive({
    req(input$file)
    delim <- if (input$delim == "") NULL else input$delim
    vroom::vroom(input$file$datapath, delim = delim, skip = input$skip)
  })
  output$preview1 <- renderTable(head(raw(), input$rows))

  # Очистка -----
  tidied <- reactive({
    out <- raw()
    if (input$snake) {
      names(out) <- janitor::make_clean_names(names(out))
    }
    if (input$empty) {
      out <- janitor::remove_empty(out, "cols")
    }
    if (input$constant) {
      out <- janitor::remove_constant(out)
    }
  })
  out

  output$preview2 <- renderTable(head(tidied(), input$rows))

  # Скачивание -----
  output$download <- downloadHandler(
    filename = function() {

```

```

    paste0(tools::file_path_sans_ext(input$file$name), ".tsv")
  },
  content = function(file) {
    vroom::vroom_write(tidied(), file)
  }
)
}

```

Результат работы приложения показан на рис. 9.4.

Raw data

Data

Upload... No file selected

Delimiter (leave blank to guess)

Rows to skip

0

Rows to preview

10

Cleaner data

☐ Rename columns to snake case?

☐ Remove constant columns?

☐ Remove empty cols?

Download

Рис. 9.4 ❖ С помощью этого приложения пользователь может загружать файлы, выполнять их очистку и скачивать на компьютер. Посмотреть приложение онлайн можно по адресу <https://hadley.shinyapps.io/ms-case-study>

УПРАЖНЕНИЯ

Упражнение 1

Используйте пакет *ambient* (<https://ambient.data-imaginist.com>) от Томаса Лина Педерсена (Thomas Lin Pedersen) для генерирования шума Уорли (worley noise) и скачивания полученного изображения в виде файла PNG.

Упражнение 2

Разработайте приложение, с помощью которого вы сможете загрузить файл CSV, выбрать переменную и применить к ней функцию `t.test()` (Для расчета критерия Стьюдента. – *Прим. перев.*). После загрузки файла CSV вам необходимо будет воспользоваться функцией `updateSelectInput()` для заполнения списка доступными переменными.

Упражнение 3

Создайте приложение, с помощью которого пользователь сможет загрузить файл CSV, выбрать одну переменную, после чего скачать на диск построенную по этой переменной гистограмму. В качестве дополнения к заданию позвольте пользователю выбрать тип сохраняемого файла: *.png*, *.pdf* или *.svg*.

Упражнение 4

Разработайте приложение, с помощью которого пользователь сможет любой файл с расширением *.png* преобразовать в мозаику Lego. Воспользуйтесь для этого пакетом *brickr* (<https://github.com/ryantimpe/brickr>) от Райана Тимпа (Ryan Timpe). Выполнив основную часть задания, попробуйте добавить в приложение элементы управления, чтобы можно было задавать размер мозаики (в кирпичиках) и выбирать между *универсальной* (*universal*) и *обычной* (*generic*) цветовой палитрой.

Упражнение 5

Наш практический пример, рассмотренный ранее в этой главе, содержит приведенное ниже большое реактивное выражение:

```
tidied <- reactive({
  out <- raw()
  if (input$snake) {
    names(out) <- janitor::make_clean_names(names(out))
  }
  if (input$empty) {
    out <- janitor::remove_empty(out, "cols")
  }
  if (input$constant) {
    out <- janitor::remove_constant(out)
  }
  out
})
```

Разбейте это выражение на части таким образом, чтобы функция `janitor::make_clean_names()`, к примеру, не перезапускалась при изменении элемента ввода `input$empty`.

ЗАКЛЮЧЕНИЕ

В данной главе вы узнали, как можно выполнять передачу файлов пользователю и от него с помощью элементов `fileInput()` и `downloadButton()`. Большинство трудностей в этой области возникает при работе с загруженными файлами или создании файлов для скачивания. Я показал лишь пару пространенных примеров. Если при этом я не осветил ваш конкретный вопрос, что ж, значит, вам придется проявить творческий подход и попытаться решить свою задачу самостоятельно.

В следующей главе вы узнаете, как можно динамически адаптировать элементы интерфейса под данные, предоставленные пользователем. Мы начнем с простой техники, которую легко можно применить в большинстве приложений, и постепенно перейдем к интерфейсу пользователя, полностью сгенерированному динамически при помощи кода.

Глава 10

Динамический интерфейс пользователя

До сих пор вы наблюдали в приложениях Shiny четкое разграничение между пользовательским интерфейсом и серверной функцией: элементы интерфейса определялись статически в момент запуска приложения и никак не могли реагировать на происходящее вокруг. В данной главе вы научитесь создавать *интерфейс пользователя динамически* (dynamic user interface) и изменять его непосредственно в коде функции `server`.

Существуют три ключевые техники для создания динамического интерфейса пользователя:

- с помощью функций семейства `update`, позволяющих изменять параметры элементов ввода;
- посредством функции `tabsetPanel()`, умеющей выборочно показывать и скрывать различные части интерфейса;
- с использованием функций `uiOutput()` и `renderUI()`, позволяющих генерировать элементы пользовательского интерфейса непосредственно в коде.

В совокупности эти три инструмента дают разработчику возможность в полной мере реагировать на действия пользователя путем изменения элементов ввода и вывода. Я покажу некоторые наиболее эффективные способы их использования, но вы можете применять их по своему усмотрению и ограничены в этом лишь вашей фантазией. В то же время не стоит забывать, что использование этих инструментов может существенно усложнить понимание ваших приложений, так что применяйте их с умом и всегда выбирайте наиболее простой способ решения вашей задачи из всех возможных. Итак, приступим:

```
library(shiny)
library(dplyr, warn.conflicts = FALSE)
```

ОБНОВЛЕНИЕ ЭЛЕМЕНТОВ ВВОДА

Начнем с простой техники, позволяющей модифицировать элементы ввода после их создания. Для этого предназначены *функции семейства `update`* (up-

date family). Каждый элемент ввода, например `textInput()`, может быть связан со своей функцией обновления `updateTextInput()`, позволяющей изменять его уже после создания.

Взгляните на код приложения, внешний вид которого представлен на рис. 10.1. В приложении имеется два числовых элемента ввода для контроля минимального и максимального значений третьего элемента, представляющего собой ползунок. Ключевая идея состоит в использовании функции `observeEvent()`¹ для вызова, в свою очередь, функции `updateSliderInput()`, позволяющей изменять характеристики ползунка при изменении значений в полях ввода:

```
ui <- fluidPage(
  numericInput("min", "Minimum", 0),
  numericInput("max", "Maximum", 3),
  sliderInput("n", "n", min = 0, max = 3, value = 1)
)

server <- function(input, output, session) {
  observeEvent(input$min, {
    updateSliderInput(inputId = "n", min = input$min)
  })
  observeEvent(input$max, {
    updateSliderInput(inputId = "n", max = input$max)
  })
}
```

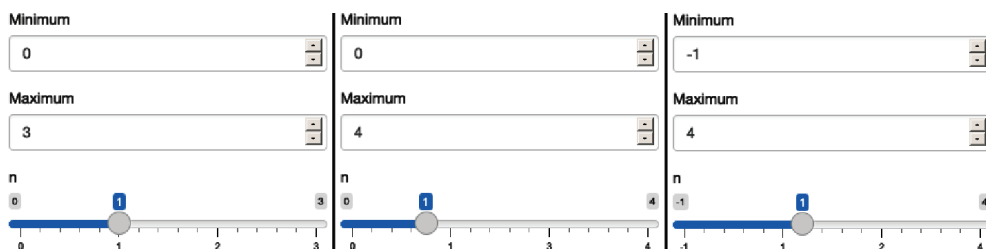


Рис. 10.1 ❖ Внешний вид приложения после запуска (слева), после увеличения максимального значения (в центре) и уменьшения минимального (справа).

Посмотреть приложение онлайн можно по адресу
<https://hadley.shinyapps.io/ms-update-basics/>

Функции семейства `update` немного отличаются от других функций Shiny – все они принимают имя элемента ввода в качестве аргумента `inputId`². Остальные аргументы согласуются с аргументами конструктора исходного элемента ввода, которые могут быть изменены.

Чтобы помочь вам лучше разобраться с функциями семейства `update`, я покажу вам несколько простых примеров, после чего мы рассмотрим

¹ Мы уже обсуждали функцию `observeEvent()` в главе 3 и будем говорить о ней подробнее в главе 15.

² Первый аргумент – `session` – присутствует для обратной совместимости, но используется крайне редко.

более сложный случай с иерархическими выпадающими списками, а затем поговорим о проблеме возникновения *циклических ссылок* (circular reference).

Простое использование

Простейшее предназначение функций семейства `update` заключается в облегчении работы пользователя. К примеру, вам может понадобиться, чтобы пользователь мог быстро сбрасывать значения элементов в исходное состояние. В следующем фрагменте кода показано, как можно сочетать функции `actionButton()`, `observeEvent()` и `updateSliderInput()` для реализации этой задачи. А внешний вид приложения продемонстрирован на рис. 10.2:

```
ui <- fluidPage(
  sliderInput("x1", "x1", 0, min = -10, max = 10),
  sliderInput("x2", "x2", 0, min = -10, max = 10),
  sliderInput("x3", "x3", 0, min = -10, max = 10),
  actionButton("reset", "Reset")
)

server <- function(input, output, session) {
  observeEvent(input$reset, {
    updateSliderInput(inputId = "x1", value = 0)
    updateSliderInput(inputId = "x2", value = 0)
    updateSliderInput(inputId = "x3", value = 0)
  })
}
```

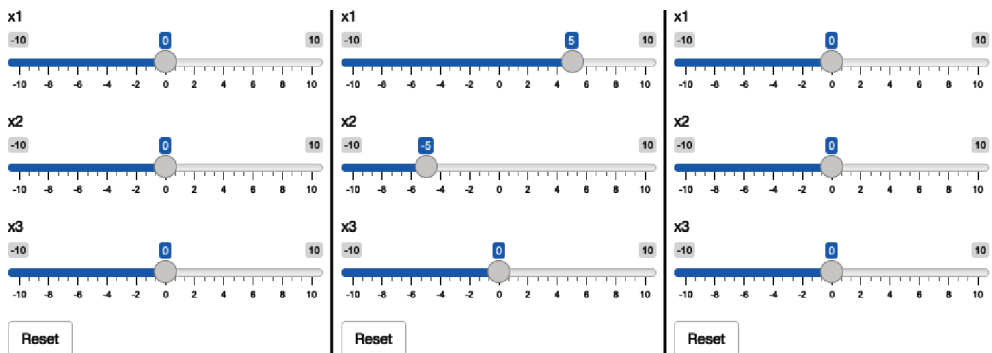


Рис. 10.2 ❖ Внешний вид приложения после запуска (слева), после перемещения ползунков (в центре) и сброса значений в исходное состояние (справа).

Посмотреть приложение можно по адресу
<https://hadley.shinyapps.io/ms-update-reset/>

Еще один простой пример приложения показывает, как можно динамически менять текст на кнопке, чтобы пользователь понимал, какое действие будет выполнено. На рис. 10.3 показан внешний вид этого приложения.

```

ui <- fluidPage(
  numericInput("n", "Simulations", 10),
  actionButton("simulate", "Simulate")
)

server <- function(input, output, session) {
  observeEvent(input$n, {
    label <- paste0("Simulate ", input$n, " times")
    updateActionButton(inputId = "simulate", label = label)
  })
}

```



Рис. 10.3 ❖ Приложение после его запуска (слева), после установки количества симуляций в 1 (в центре) и в 100 (справа).
Посмотреть приложение можно по адресу
<https://hadley.shinyapps.io/ms-update-button>

Существует масса способов похожего использования функций семейства `update`. Ваша цель в этом случае – дать пользователю больше полезной информации о работе приложения. Также при помощи этих функций можно динамически ограничивать выбор в выпадающих списках посредством пошаговой фильтрации, как будет показано дальше.

Иерархические выпадающие списки

Еще одним простым, но очень полезным применением функций семейства `update` является реализация интерактивной детализации по нескольким категориям при помощи выпадающих списков. Я продемонстрирую этот прием на примере вымышленных данных о продажах из набора данных Kaggle (<https://www.kaggle.com/kyanyoga/sample-sales-data>):

```

sales <- vroom::vroom(
  "sales-dashboard/sales_data_sample.csv",
  col_types = list(),
  na = ""
)

sales %>%
  select(TERRITORY, CUSTOMERNAME, ORDERNUMBER, everything()) %>%
  arrange(ORDERNUMBER)

#> # A tibble: 2,823 x 25
#>   TERRITORY CUSTOMERNAME ORDERNUMBER QUANTITYORDERED PRICEEACH ORDERLINENUMBER
#>   <chr>      <chr>          <dbl>          <dbl>      <dbl>          <dbl>

```

```
#> 1 NA      Online Diecas...    10100      30    100      3
#> 2 NA      Online Diecas...    10100      50    67.8     2
#> 3 NA      Online Diecas...    10100      22    86.5     4
#> 4 NA      Online Diecas...    10100      49    34.5     1
#> 5 EMEA     Blauer See Au...    10101      25    100      4
#> 6 EMEA     Blauer See Au...    10101      26    100      1
#> 7 EMEA     Blauer See Au...    10101      45    31.2     3
#> 8 EMEA     Blauer See Au...    10101      46    53.8     2
#> 9 NA      Vitachrome In...    10102      39    100      2
#> 10 NA     Vitachrome In...    10102      41    50.1     1
#> # ... with 2,813 more rows, and 19 more variables: SALES <dbl>, ORDERDATE <chr>,
#> #   STATUS <chr>, QTR_ID <dbl>, MONTH_ID <dbl>, YEAR_ID <dbl>,
#> #   PRODUCTLINE <chr>, MSRP <dbl>, PRODUCTCODE <chr>, PHONE <chr>,
#> #   ADDRESSLINE1 <chr>, ADDRESSLINE2 <chr>, CITY <chr>, STATE <chr>,
#> #   POSTALCODE <chr>, COUNTRY <chr>, CONTACTLASTNAME <chr>,
#> #   CONTACTFIRSTNAME <chr>, DEALSIZE <chr>
```

В этом примере мы будем опираться на естественную иерархию данных:

- *покупатели* (customer) принадлежат *территориям* (territory);
- у покупателей может быть по несколько *заказов* (order);
- каждый заказ содержит строки.

Наша цель – разработать интерфейс пользователя, в котором можно будет:

- выбрать территорию для просмотра всех принадлежащих ей покупателей;
- выбрать покупателя для просмотра всех принадлежащих ему заказов;
- выбрать заказ для просмотра всех строк в нем.

Базовая часть пользовательского интерфейса будет состоять из трех выпадающих списков и одной таблицы для вывода. При этом наполнение списков покупателей (customername) и заказов (ordernumber) будет осуществляться динамически, так что установим параметр choices равным NULL:

```
ui <- fluidPage(
  selectInput("territory", "Territory", choices = unique(sales$TERRITORY)),
  selectInput("customername", "Customer", choices = NULL),
  selectInput("ordernumber", "Order number", choices = NULL),
  tableOutput("data")
)
```

Что же будет происходить в серверной функции?

1. Создадим реактивное выражение `territory()` со списком строк из таблицы `sales`, соответствующих выбранной территории.
2. При изменении `territory()` будем обновлять содержимое выпадающего списка `input$customername` путем изменения его параметра `choices`.
3. Создадим еще одно реактивное выражение `customer()` со списком строк из `territory()`, соответствующих выбранному покупателю.
4. При изменении `customer()` будем обновлять список заказов в выпадающем списке `input$ordernumber`, также изменяя параметр `choices`.
5. Выводим список выбранных заказов в таблице `output$data`.

Код функции `server` приведен ниже, а результат работы приложения показан на рис. 10.4:

```
server <- function(input, output, session) {
  territory <- reactive({
    filter(sales, TERRITORY == input$territory)
  })
  observeEvent(territory(), {
    choices <- unique(territory())$CUSTOMERNAME
    updateSelectInput(inputId = "customername", choices = choices)
  })

  customer <- reactive({
    req(input$customername)
    filter(territory(), CUSTOMERNAME == input$customername)
  })
  observeEvent(customer(), {
    choices <- unique(customer())$ORDERNUMBER
    updateSelectInput(inputId = "ordernumber", choices = choices)
  })

  output$data <- renderTable({
    req(input$ordernumber)
    customer() %>%
      filter(ORDERNUMBER == input$ordernumber) %>%
      select(QUANTITYORDERED, PRICEEACH, PRODUCTCODE)
  })
}
```

QUANTITYORDERED	PRICEEACH	PRODUCTCODE
30.00	95.70	S10_1678
39.00	99.91	S10_2016
27.00	100.00	S10_4698
21.00	100.00	S12_2823
29.00	70.87	S18_2625
25.00	100.00	S24_1578
38.00	83.03	S24_2000
20.00	92.90	S32_1374

QUANTITYORDERED	PRICEEACH	PRODUCTCODE
41.00	94.74	S10_1678
27.00	100.00	S10_2016
31.00	100.00	S10_4698
20.00	100.00	S12_2823
30.00	61.78	S18_2625
35.00	93.54	S24_1578
43.00	83.03	S24_2000

Рис. 10.4 ❖ Выбираем территорию *EMEA* (слева), затем покупателя *Lyon Souvenirs* (по центру) и ищем нужный нам заказ (справа).
Посмотреть приложение онлайн можно по адресу <https://hadley.shinyapps.io/ms-update-nested>

Исходный код полноценного приложения можно скачать по адресу <https://github.com/hadley/mastering-shiny/tree/master/sales-dashboard>.

Заморозка реактивного ввода

Иногда наличие в приложении иерархических выпадающих списков, содержимое которых обновляется динамически, приводит к кратковременному мерцанию элементов вывода из-за неправильной комбинации значений элементов ввода. Взгляните на следующее простое приложение, в котором пользователь может выбрать набор данных и переменную для просмотра сводной информации:

```
ui <- fluidPage(
  selectInput("dataset", "Choose a dataset", c("pressure", "cars")),
  selectInput("column", "Choose column", character(0)),
  verbatimTextOutput("summary")
)

server <- function(input, output, session) {
  dataset <- reactive(get(input$dataset, "package:datasets"))

  observeEvent(input$dataset, {
    updateSelectInput(inputId = "column", choices = names(dataset()))
  })

  output$summary <- renderPrint({
    summary(dataset()[[input$column]])
  })
}
```

Опробовав в действии это приложение по адресу <https://hadley.shinyapps.io/ms-freeze/>, вы заметите, что при переключении наборов данных таблица вывода моргает. Причина этого состоит в том, что функция `updateSelectInput()` фиксирует изменения только после того, как отработают все элементы вывода и наблюдатели, так что на какое-то мгновение возникает ситуация, когда в первом списке выбран набор данных B, а во втором – переменная из набора A. Это приводит к выполнению инструкции `summary(NULL)`.

Во избежание данного эффекта принято «замораживать» элемент ввода при помощи функции `freezeReactiveValue()`. В результате реактивные выражения и элементы вывода, использующие этот элемент ввода, не будут обновляться вплоть до следующего полного раунда перевода в статус недействительности¹:

```
server <- function(input, output, session) {
  dataset <- reactive(get(input$dataset, "package:datasets"))

  observeEvent(input$dataset, {
    freezeReactiveValue(input, "column")
    updateSelectInput(inputId = "column", choices = names(dataset()))
  })

  output$summary <- renderPrint({
```

¹ Если говорить точнее, любая попытка прочитать замороженный элемент ввода будет приводить к инструкции `req(FALSE)`.

```
summary(dataset()[[input$column]])
  })
}
```

Обратите внимание, что после заморозки нет необходимости «отогревать» элементы ввода, – это произойдет автоматически после того, как Shiny определит, что сессия и сервер вновь вошли в состояние синхронизации.

Наверняка вас интересует вопрос: когда все же нужно использовать функцию `freezeReactiveValue()`. На самом деле ее лучше вызывать *всегда*, когда происходит динамическое изменение значений элементов ввода. Всякое изменение требует определенного времени для отправки значений в браузер, а затем обратно в Shiny, и в этом промежутке любые попытки прочесть значения в лучшем случае будут лишними, а в худшем приведут к появлению ошибок. Используйте функцию `freezeReactiveValue()`, чтобы сообщить всем последующим вычислениям о том, что значение элемента ввода утратило актуальность и следует повременить с обновлением до получения им нового значения.

Циклические ссылки

Есть еще один важный момент, который просто необходимо обсудить, если вы собираетесь использовать функции семейства `update` для изменения текущих значений элементов ввода¹. С точки зрения Shiny изменение вами значений при помощи функций `update` ничем не отличается от модификации значений пользователем при помощи мыши и клавиатуры. Это означает, что функции семейства `update` могут инициировать обновление реактивных выражений точно так же, как это делает пользователь. А значит, фактически вы выходите за рамки классического реактивного программирования, и вам стоит позаботиться о возникновении *циклических ссылок* (circular reference) и бесконечных циклов.

Рассмотрим следующее простое приложение. В нем содержится единственный элемент ввода и наблюдатель, увеличивающий его значение на единицу. При каждом срабатывании функции `updateNumericInput()` происходит изменение значения в поле `input$n`, что, в свою очередь, снова вызывает функцию `updateNumericInput()`. В результате приложение входит в бесконечный цикл, постоянно увеличивая значение элемента `input$n`:

```
ui <- fluidPage(
  numericInput("n", "n", 0)
)

server <- function(input, output, session) {
```

¹ В основном это относится к изменению вами значений напрямую, но не стоит забывать, что значения могут быть изменены и неявно. К примеру, если вы меняете свойство `choices` у элемента ввода `selectInput()` или `min/max` у `sliderInput()`, текущее значение `value` может быть автоматически изменено, если более не попадает в установленный диапазон значений.

```

    observeEvent(input$n,
      updateNumericInput(inputId = "n", value = input$n + 1)
    )
  }

```

Конечно, вы вряд ли допустите в своем приложении такую очевидную ошибку, но подобное поведение может проявиться в случае, если вы изменяете значения нескольких элементов ввода, зависящих друг от друга, как будет показано в следующем примере.

Взаимосвязанные элементы ввода

Проблема возникновения циклических ссылок зачастую проявляется в приложениях с несколькими «источниками истины». Представьте, что вы разрабатываете приложение для преобразования значений температуры, в котором пользователь может вводить исходные данные как в градусах Цельсия, так и в градусах Фаренгейта:

```

ui <- fluidPage(
  numericInput("temp_c", "Celsius", NA, step = 1),
  numericInput("temp_f", "Fahrenheit", NA, step = 1)
)

server <- function(input, output, session) {
  observeEvent(input$temp_f, {
    c <- round((input$temp_f - 32) * 5 / 9)
    updateNumericInput(inputId = "temp_c", value = c)
  })
  observeEvent(input$temp_c, {
    f <- round((input$temp_c * 9 / 5) + 32)
    updateNumericInput(inputId = "temp_f", value = f)
  })
}

```

Если вы попытаете поработать с этим приложением онлайн по адресу <https://hadley.shinyapps.io/ms-temperature>, то очень быстро обнаружите, что в большинстве случаев оно будет работать нормально, но время от времени будут возникать двойные пересчеты. Проверьте сами:

- установите температуру, равную 120 градусам по Фаренгейту, и нажмите на кнопку вниз;
- значение в поле изменится на 119, а температура по Цельсию будет выставлена в значение 48;
- 48 градусов по Цельсию будут преобразованы в 118 градусов по Фаренгейту, что вновь приведет к изменению значения в соответствующем поле;
- к счастью, 118 градусов по Фаренгейту соответствуют 48 градусам Цельсия, в связи с чем цикл будет остановлен.

Обойти эту проблему невозможно, поскольку в данном случае у нас есть один показатель (температура) и два способа его выражения (в градусах

Цельсия и Фаренгейта). Нам повезло, что цикл довольно быстро завершился, когда были найдены значения, отвечающие обоим требованиям. В общем, желательно избегать подобных ситуаций в своих приложениях, если вы не хотите долго и упорно анализировать свойства сходимости созданной вами динамической системы.

Упражнения

Упражнение 1

Напишите серверную функцию для представленного ниже интерфейса пользователя. Значение поля `input$date` должно обновляться автоматически, тогда как пользователь должен иметь возможность вводить год только в поле `input$year`:

```
ui <- fluidPage(
  numericInput("year", "year", value = 2020),
  dateInput("date", "date")
)
```

Упражнение 2

Напишите серверную функцию для представленного ниже интерфейса пользователя. Выпадающий список `input$county` должен автоматически заполняться значениями на основании выбора штата в списке `input$state`. В качестве дополнительного задания предлагаем вам менять подпись *County* на *Parish* (приход) для Луизианы (Louisiana) и на *Borough* (боро) для Аляски (Alaska):

```
library(openintro, warn.conflicts = FALSE)
states <- unique(county$state)

ui <- fluidPage(
  selectInput("state", "State", choices = states),
  selectInput("county", "County", choices = NULL)
)
```

Упражнение 3

Напишите серверную функцию для представленного ниже интерфейса пользователя. Выпадающий список `input$country` должен автоматически заполняться значениями на основании выбора континента в списке `input$continent`. Используйте элемент вывода `output$data` для отображения полученных данных:

```
library(gapminder)
continents <- unique(gapminder$continent)

ui <- fluidPage(
  selectInput("continent", "Continent", choices = continents),
  selectInput("country", "Country", choices = NULL),
  tableOutput("data")
)
```

Упражнение 4

Расширьте приложение из предыдущего упражнения таким образом, чтобы пользователь мог выбрать все континенты и, следовательно, увидеть все страны. Для этого необходимо будет в выпадающий список добавить значение "(All)" и учитывать его при фильтрации.

Упражнение 5

В чем заключается суть проблемы, описанной в посте по адресу <https://community.rstudio.com/t/mutually-dependent-numericinput-in-shiny/29307>?

ДИНАМИЧЕСКАЯ ВИДИМОСТЬ

Следующим по уровню сложности аспектом динамического управления пользовательским интерфейсом является выборочный показ и скрытие областей приложения. Если вы знаете JavaScript и CSS, то сможете применить более изощренные подходы в отношении видимости элементов, но эти знания вам не понадобятся, чтобы реализовать приложение с использованием *набора вкладок* (tabset), о котором мы уже говорили в главе 6. С помощью этой техники вы можете показывать и скрывать целые области приложения без необходимости создавать элементы заново, как вы увидите в следующем разделе:

```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      selectInput("controller", "Show", choices = paste0("panel", 1:3))
    ),
    mainPanel(
      tabsetPanel(
        id = "switcher",
        type = "hidden",
        tabPanelBody("panel1", "Panel 1 content"),
        tabPanelBody("panel2", "Panel 2 content"),
        tabPanelBody("panel3", "Panel 3 content")
      )
    )
  )
)

server <- function(input, output, session) {
  observeEvent(input$controller, {
    updateTabsetPanel(inputId = "switcher", selected = input$controller)
  })
}
```

Результат запуска приложения показан на рис. 10.5.



Рис. 10.5 ❖ Выбор *panel1* (слева), *panel2* (по центру) и *panel3* (справа).

Посмотреть приложение онлайн можно по адресу
<https://hadley.shinyapps.io/ms-dynamic-panels>

В этом подходе заложены две основные идеи:

- используется `tabsetPanel` со скрытыми вкладками;
- переключение между вкладками осуществляется при помощи функции `updateTabsetPanel()`.

Мы продемонстрировали простейший способ реализации этой техники, но достаточно проявить немного креатива, чтобы создать довольно сложный макет приложения. В следующих двух разделах мы покажем, как можно использовать этот способ организации элементов в макете приложения на практике.

Условный интерфейс пользователя

Представьте, что вам нужно разработать приложение, в котором пользователь для тех или иных целей будет выбирать между нормальным, равномерным и экспоненциальным распределениями. Каждое распределение характеризуется своим набором параметров, так что нам необходимо отображать разные элементы управления в зависимости от выбора пользователя. Давайте для каждого набора параметров создадим свою вкладку при помощи функции `tabPanel()` и объединим их в единый набор вкладок, воспользовавшись функцией `tabsetPanel()`, как показано ниже:

```
parameter_tabs <- tabsetPanel(
  id = "params",
  type = "hidden",
  tabPanel("normal",
    numericInput("mean", "mean", value = 1),
    numericInput("sd", "standard deviation", min = 0, value = 1)
  ),
  tabPanel("uniform",
    numericInput("min", "min", value = 0),
    numericInput("max", "max", value = 1)
  ),
  tabPanel("exponential",
```

```

    numericInput("rate", "rate", value = 1, min = 0),
  )
)

```

После этого встроим полученную в результате панель вкладок в наш интерфейс, в котором пользователь может выбрать тип распределения, указать количество выборок и построить гистограмму:

```

ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      selectInput("dist", "Distribution",
        choices = c("normal", "uniform", "exponential")
      ),
      numericInput("n", "Number of samples", value = 100),
      parameter_tabs,
    ),
    mainPanel(
      plotOutput("hist")
    )
  )
)

```

Заметьте, что названия типов распределения в параметре `choices` элемента ввода `input$dist` в точности совпадают с названиями вкладок. Это сделано для облегчения написания функции `observeEvent()`, которая автоматически переключает вкладки при изменении выбора пользователя. В остальном в приложении применяются уже известные вам техники:

```

server <- function(input, output, session) {
  observeEvent(input$dist, {
    updateTabsetPanel(inputId = "params", selected = input$dist)
  })

  sample <- reactive({
    switch(input$dist,
      normal = rnorm(input$n, input$mean, input$sd),
      uniform = runif(input$n, input$min, input$max),
      exponential = rexp(input$n, input$rate)
    )
  })

  output$hist <- renderPlot(hist(sample()), res = 96)
}

```

На рис. 10.6 показан внешний вид приложения. Заметьте, что значения элементов ввода, например `input$mean`, не зависят от того, видна соответствующая секция пользователю или нет. Код разметки HTML не изменится, вы просто не будете ее видеть.

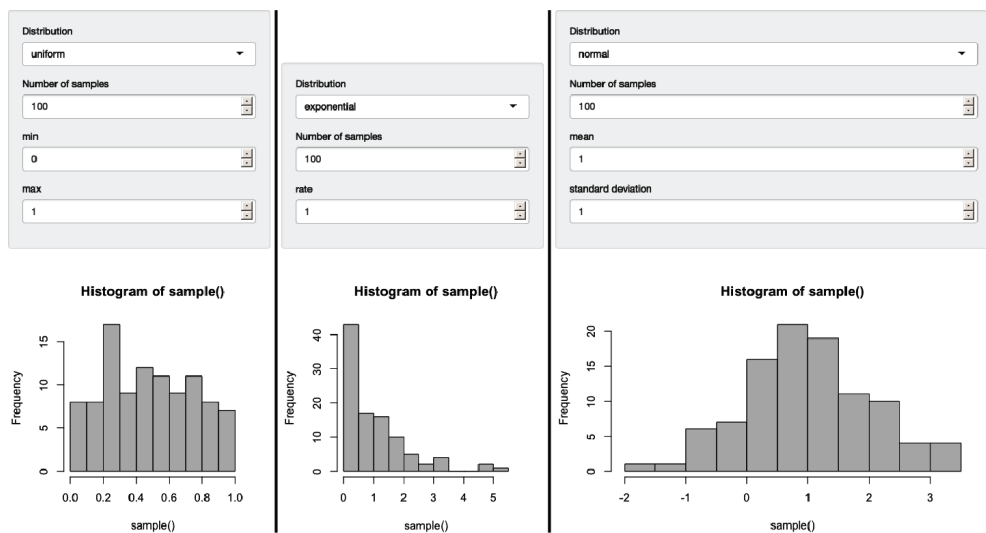


Рис. 10.6 ❖ Выбор равномерного (слева), экспоненциального (в центре) и нормального (справа) распределения.

Приложение онлайн можно посмотреть по адресу <https://hadley.shinyapps.io/ms-dynamic-conditional>

Интерфейс мастера

Описанные здесь идеи можно эффективно использовать при создании так называемого *мастера* (wizard) – типа приложения, в котором данные собираются пошагово, при переходе со страницы на страницу. Давайте расставим кнопки на страницы и позволим пользователю перемещаться с их помощью вперед и назад:

```
ui <- fluidPage(
  tabsetPanel(
    id = "wizard",
    type = "hidden",
    tabPanel("page_1",
      "Welcome!",
      actionButton("page_12", "next")
    ),
    tabPanel("page_2",
      "Only one page to go",
      actionButton("page_21", "prev"),
      actionButton("page_23", "next")
    ),
    tabPanel("page_3",
      "You're done!",
      actionButton("page_32", "prev")
    )
  )
)
```

```
server <- function(input, output, session) {
  switch_page <- function(i) {
    updateTabsetPanel(inputId = "wizard", selected = paste0("page_", i))
  }

  observeEvent(input$page_12, switch_page(2))
  observeEvent(input$page_21, switch_page(1))
  observeEvent(input$page_23, switch_page(3))
  observeEvent(input$page_32, switch_page(2))
}
```

Пошаговое выполнение приложения показано на рис. 10.7.



Рис. 10.7 ❖ Простой многостраничный мастер.

С приложением можно ознакомиться по адресу <https://hadley.shinyapps.io/ms-wizard/>

Обратите внимание, как мы использовали функцию `switch_page()` для уменьшения количества дублирующегося кода на серверной стороне. Мы вернемся к этому в главе 18 и создадим модуль для автоматизации создания мастера.

Упражнения

Упражнение 1

Используйте в приложении скрытую вкладку с элементами управления, которая будет появляться только при установке флажка *Дополнительно* (Advanced).

Упражнение 2

Разработайте приложение, использующее инструкцию `ggplot(diamonds, aes(carat))` для построения графика, но при этом пользователь должен иметь возможность выбора геометрии между `geom_histogram()`, `geom_freqpoly()` и `geom_density()`. Используйте скрытую вкладку, чтобы позволить пользователю установить значения характерных для выбранной геометрии аргументов: для `geom_histogram()` и `geom_freqpoly()` это `binwidth`, а для `geom_density()` – `bw`.

Упражнение 3

Измените созданное в предыдущем упражнении приложение таким образом, чтобы пользователь мог выбрать, показывать или нет конкретную геометрию. То есть вместо того чтобы видеть всегда лишь одну геометрию, пользователь должен иметь возможность выбрать, сколько геометрий показывать: ни одной, одну, две или три. Обеспечьте отдельную настройку аргумента `binwidth` для геометрий `geom_histogram()` и `geom_freqpoly()`.

СОЗДАНИЕ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ ПРИ ПОМОЩИ КОДА

Бывают ситуации, когда ни одна из рассмотренных нами техник не даст вам требуемого уровня динамики управления содержимым приложения. Функции семейства `update` позволяют вам менять характеристики только существующих элементов управления, а вкладки могут применяться лишь при наличии фиксированного и известного набора возможных комбинаций. Иногда же необходимо создавать элементы ввода или вывода динамически на основании других элементов в приложении. И техника, которую мы обсудим в этом разделе, позволит это сделать.

Здесь стоит отметить, что вы и до этого все элементы управления в приложении создавали при помощи кода – разница лишь в том, что делали вы это до запуска приложения. Рассмотренный в этом разделе подход позволит вам создавать элементы прямо во время работы приложения. Для его реализации необходимо и достаточно выполнения двух условий:

- в интерфейсе пользователя должна присутствовать заглушка в виде `uiOutput()`. Именно это пространство впоследствии будет заполнено новыми элементами;
- внутри функции `server()` должна вызываться функция `renderUI()` для динамического заполнения пространства созданными элементами.

Сначала на простом примере посмотрим, как это работает, после чего опишем более реалистичный случай.

Введение

Давайте разработаем простое приложение с динамически создаваемым элементом ввода, тип и название которого будут определяться двумя другими элементами:

```
ui <- fluidPage(
  textInput("label", "label"),
  selectInput("type", "type", c("slider", "numeric")),
  uiOutput("numeric")
)

server <- function(input, output, session) {
  output$numeric <- renderUI({
    if (input$type == "slider") {
      sliderInput("dynamic", input$label, value = 0, min = 0, max = 10)
    } else {
      numericInput("dynamic", input$label, value = 0, min = 0, max = 10)
    }
  })
}
```

Результат запуска приложения показан на рис. 10.8.

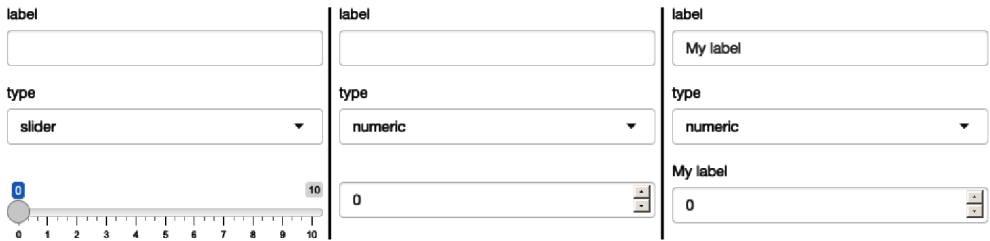


Рис. 10.8 ❖ Приложение после запуска (слева), после изменения типа элемента на числовой (в центре) и после задания метки (справа).
Посмотреть приложение онлайн можно по адресу
<https://hadley.shinyapps.io/ms-render-simple>

При запуске приложения вы заметите, что ему понадобится какое-то заметное время для отрисовки элементов. Это происходит из-за реактивного характера приложения: после запуска оно инициирует реактивное событие, после чего вызывается серверная функция, возвращающая код разметки HTML, предназначенный для вставки на страницу. В этом заключается основной недостаток функции `renderUI()` – если слишком уж на нее полагаться, интерфейс приложения может получиться несколько заторможенным. Для обеспечения максимальной производительности старайтесь как можно больше элементов размещать в интерфейсе статически с использованием техник, описанных в предыдущих разделах.

Есть у этого подхода и еще одно неудобство – при изменении элементов вы теряете текущее выбранное значение. Поддержка текущего состояния – одна из важнейших задач в области динамического создания интерфейса пользователя. Именно поэтому более приемлемым может быть вариант с выборочным отображением и скрытием областей приложения – в этом случае нет необходимости заново создавать элементы управления, а значит, не нужно беспокоиться о сохранении их текущих значений. Но зачастую эта проблема решается путем восстановления значения параметра `value` для созданного элемента, как показано ниже:

```
server <- function(input, output, session) {
  output$numeric <- renderUI({
    value <- isolate(input$dynamic)
    if (input$type == "slider") {
      sliderInput("dynamic", input$label, value = value, min = 0, max = 10)
    } else {
      numericInput("dynamic", input$label, value = value, min = 0, max = 10)
    }
  })
}
```

Использование функции `isolate()` очень важно. Мы рассмотрим ее более подробно в главе 15, а сейчас скажем, что в данном случае ее вызов по-

звояет избежать создания реактивной зависимости, которая привела бы к повторному запуску кода каждый раз, когда меняется значение элемента `input$dynamic` (а это будет происходить при любом изменении значения пользователем). Мы хотим, чтобы изменения происходили только в случае выбора пользователем нового значения поля `input$type` или `input$label`.

Множественные элементы управления

Динамический интерфейс пользователя бывает особенно полезен при необходимости создавать произвольное количество элементов разных типов. Для подобных задач я бы рекомендовал использовать принципы *функционального программирования* (functional programming). В данном случае мы воспользуемся функциями `purrr::map()` и `purrr::reduce()`, но могли бы добиться таких же результатов и с использованием стандартных функций `lapply()` и `Reduce()`. Загрузим пакет:

```
library(purrr)
```

Если вы незнакомы с принципами работы функций `map()` и `reduce()`, я бы советовал вам для начала прочитать о парадигме функционального программирования по адресу <https://adv-r.hadley.nz/functionals.html>, после чего возвращаться к чтению книги. Мы также подробнее будем говорить об этом в главе 18. Это не самый простой материал, так что не расстраивайтесь, если не усвоите его с первого раза.

Представьте, что вы хотите дать пользователю возможность составить собственную цветовую палитру. Для начала он должен указать, какое количество цветов в ней будет, а затем вписать значения всех цветов. Базовый интерфейс пользователя будет довольно простым: один элемент `numericInput()` для ввода количества цветов в палитре, `uiOutput()` для будущих полей и `textOutput()` для вывода перечня выбранных цветов:

```
ui <- fluidPage(
  numericInput("n", "Number of colours", value = 5, min = 1),
  uiOutput("col"),
  textOutput("palette")
)
```

Серверная функция в нашем случае будет короткой, но довольно емкой:

```
server <- function(input, output, session) {
  col_names <- reactive(paste0("col", seq_len(input$n)))

  output$col <- renderUI({
    map(col_names(), ~ textInput(.x, NULL))
  })

  output$palette <- renderText({
    map_chr(col_names(), ~ input[[.x]] %||% "")
  })
}
```

- Я использовал реактивное выражение `col_names()` для хранения имен будущих полей ввода цветов.
- После этого я применил функцию `map()` для создания списка элементов `textInput()` – по одному для каждого имени в списке `col_names()`. Затем функция `genderUI()` принимает этот список компонентов HTML и добавляет их в интерфейс пользователя.
- Мне пришлось воспользоваться специальным трюком для доступа к значениям элементов ввода. До этого мы пользовались только записью со знаком доллара (`$`). Но в данном случае имена элементов ввода находятся в символьном векторе вроде `var <- "col1"`. Символ `$` здесь не помог бы, так что нам пришлось переключиться на двойные квадратные скобки (`[[]]`), т. е. на запись `input[[var]]`.
- Функцией `map_chr()` я воспользовался для сбора всех значений в символьный вектор и его отображения в элементе вывода `output$palette`. К сожалению, непосредственно перед отображением новых элементов в браузере возникает кратковременный период, когда их значения равны `NULL`. Это приводит к возникновению ошибки при вызове функции `map_chr()`, чего нам удалось избежать при помощи удобного оператора `%/%`, возвращающего правую часть выражения, когда левая равна `NULL`.

Внешний вид приложения показан на рис. 10.9.

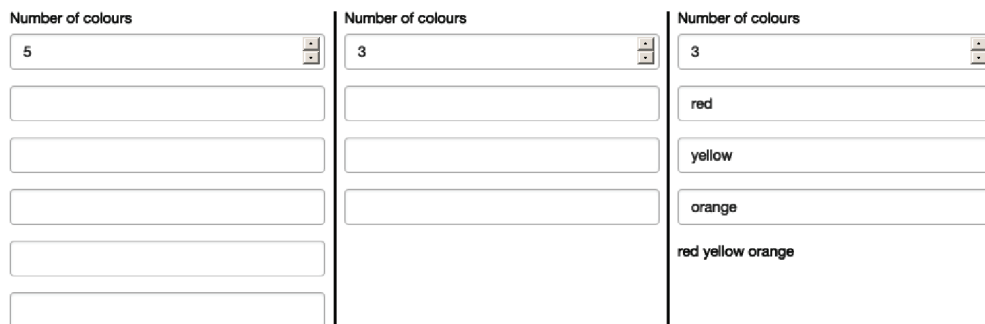


Рис. 10.9 ❖ Приложение после запуска (слева), после ввода количества цветов (в центре) и после ввода их значений (справа).
Посмотреть приложение онлайн можно по адресу <https://hadley.shinyapps.io/ms-render-palette>

Если вы запустите это приложение, то столкнетесь с его крайне раздражающим поведением: при каждом изменении количества цветов все введенные ранее значения будут теряться. Мы можем исправить эту проблему так же, как и в предыдущем примере, – при помощи функции `isolated()`. Попутно приведем внешний вид приложения в порядок и выведем выбранные цвета на графике. Итоговое приложение показано на рис. 10.10:

```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
```

```

        numericInput("n", "Number of colours", value = 5, min = 1),
        uiOutput("col"),
    ),
    mainPanel(
        plotOutput("plot")
    )
)
)
)

server <- function(input, output, session) {
  col_names <- reactive(paste0("col", seq_len(input$n)))

  output$col <- renderUI({
    map(col_names(), ~ textInput(., NULL, value = isolate(input[[.x]])))
  })

  output$plot <- renderPlot({
    cols <- map_chr(col_names(), ~ input[[.x]] %||% "")
    # делаем пустые значения прозрачными
    cols[cols == ""] <- NA

    barplot(
      rep(1, length(cols)),
      col = cols,
      space = 0,
      axes = FALSE
    )
  }, res = 96)
}

```

Динамическая фильтрация

В заключение этой главы мы разработаем приложение для *динамической фильтрации* (dynamic filtering) произвольного датафрейма. Каждая числовая переменная (столбец) получит свой ползунок с диапазоном, а каждая факторная – элемент управления множественным выбором. Таким образом, если в датафрейме присутствуют три числовые переменные и две факторные, в приложении будет три ползунка и два поля с множественным выбором.

Начнем с функции, создающей интерфейс пользователя для единственной переменной. В случае с числовым форматом поля будем возвращать ползунок нужного диапазона, для фактора вернем соответствующее поле с мультिवыбором, а для остальных типов – NULL:

```

make_ui <- function(x, var) {
  if (is.numeric(x)) {
    rng <- range(x, na.rm = TRUE)
    sliderInput(var, var, min = rng[1], max = rng[2], value = rng)
  } else if (is.factor(x)) {
    levs <- levels(x)
    selectInput(var, var, choices = levs, selected = levs, multiple = TRUE)
  } else {
    # Не поддерживается
  }
}

```

```

    NULL
  }
}

```



Рис. 10.10 ❖ Приложение с пятью цветами радуги (вверху) и тремя (внизу). Заметьте, что прежний выбор цветов сохранился. Посмотреть приложение онлайн можно по адресу <https://hadley.shinyapps.io/ms-render-palette-full>

Теперь напомним серверный аналог этой функции. На вход будет подаваться переменная и значение из элемента управления, а на выходе получим логический вектор, указывающий, включать или нет то или иное наблюдение. Использование логического вектора значительно облегчает задачу комбинирования результатов из нескольких столбцов:

```

filter_var <- function(x, val) {
  if (is.numeric(x)) {
    !is.na(x) & x >= val[1] & x <= val[2]
  } else if (is.factor(x)) {
    x %in% val
  } else {
    # Не фильтруем
    TRUE
  }
}

```

В результате мы можем использовать эти функции для создания простого интерфейса пользователя с фильтрацией набора данных `iris` следующим образом:

```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      make_ui(iris$Sepal.Length, "Sepal.Length"),
      make_ui(iris$Sepal.Width, "Sepal.Width"),
      make_ui(iris$Species, "Species")
    ),
    mainPanel(
      tableOutput("data")
    )
  )
)

server <- function(input, output, session) {
  selected <- reactive({
    filter_var(iris$Sepal.Length, input$Sepal.Length) &
    filter_var(iris$Sepal.Width, input$Sepal.Width) &
    filter_var(iris$Species, input$Species)
  })

  output$data <- renderTable(head(iris[selected(), ], 12))
}
```

На рис. 10.11 показан внешний вид нашего приложения.

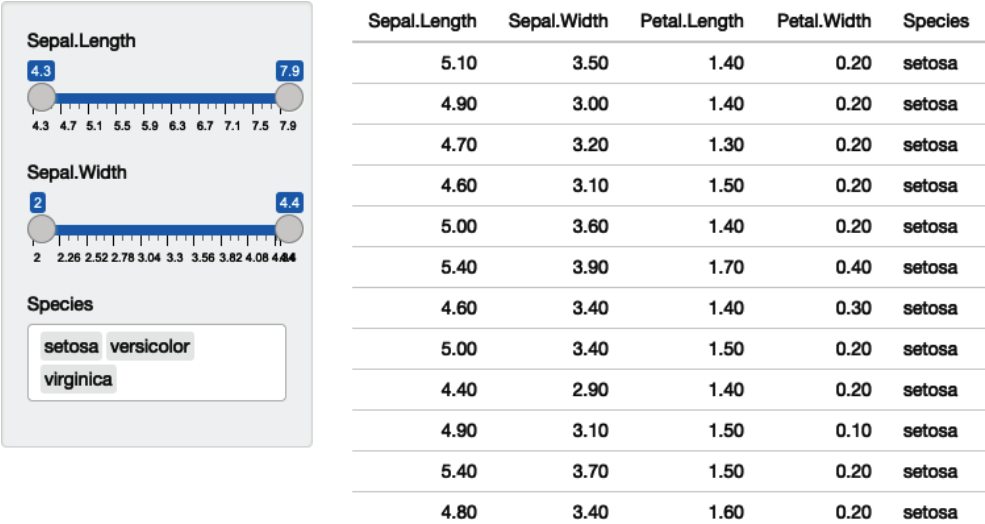


Рис. 10.11 ❖ Простой интерфейс с динамической фильтрацией набора данных `iris`

Вы, наверное, заметили, что при написании этого приложения нам пришлось неоднократно пользоваться техникой копирования и вставки, и в ре-

зультате оно работает только с тремя указанными нами столбцами. Но можно сделать так, чтобы приложение работало со всеми столбцами в наборе данных, – для этого необходимо применить принципы функционального программирования следующим образом:

- в интерфейсе мы воспользуемся функцией `map()` для создания элемента управления для каждой переменной в наборе данных;
- в функции `server()` применим функцию `map()` для создания вектора выбора для каждой переменной. После этого используем функцию `reduce()` для объединения логических векторов в единый логический вектор с применением оператора `&`.

Не волнуйтесь, если вы не до конца понимаете написанный код. Я лишь хотел показать, что знание принципов функционального программирования позволяет писать предельно лаконичный код при создании сложных динамических приложений:

```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      map(names(iris), ~ make_ui(iris[.[x]], .x))
    ),
    mainPanel(
      tableOutput("data")
    )
  )
)

server <- function(input, output, session) {
  selected <- reactive({
    each_var <- map(names(iris), ~ filter_var(iris[.[x]], input[.[x]]))
    reduce(each_var, ~ .x & .y)
  })

  output$data <- renderTable(head(iris[selected()], ), 12))
}
```

На рис. 10.12 показан внешний вид приложения.

Осталось вынести за скобки анализируемый набор данных, чтобы пользователь мог сам его выбирать в приложении. В приведенном ниже примере предусмотрен выбор набора данных из пакета `datasets`, но его вполне можно расширить для анализа датафреймов, загруженных пользователем:

```
dfs <- keep(ls("package:datasets"), ~ is.data.frame(get(.x, "package:datasets")))

ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      selectInput("dataset", label = "Dataset", choices = dfs),
      uiOutput("filter")
    ),
    mainPanel(
      tableOutput("data")
    )
  )
)
```

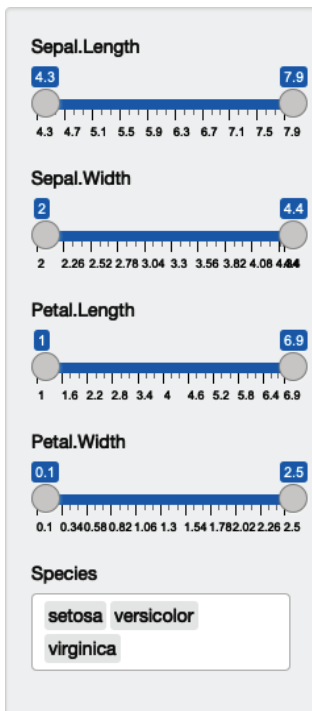
```
)
)
)

server <- function(input, output, session) {
  data <- reactive({
    get(input$dataset, "package:datasets")
  })
  vars <- reactive(names(data()))

  output$filter <- renderUI(
    map(vars(), ~ make_ui(data()[[.x]], .x))
  )

  selected <- reactive({
    each_var <- map(vars(), ~ filter_var(data()[[.x]], input[[.x]]))
    reduce(each_var, `&`)
  })

  output$data <- renderTable(head(data()[selected(), ], 12))
}
```



Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.10	3.50	1.40	0.20	setosa
4.90	3.00	1.40	0.20	setosa
4.70	3.20	1.30	0.20	setosa
4.60	3.10	1.50	0.20	setosa
5.00	3.60	1.40	0.20	setosa
5.40	3.90	1.70	0.40	setosa
4.60	3.40	1.40	0.30	setosa
5.00	3.40	1.50	0.20	setosa
4.40	2.90	1.40	0.20	setosa
4.90	3.10	1.50	0.10	setosa
5.40	3.70	1.50	0.20	setosa
4.80	3.40	1.60	0.20	setosa

Рис. 10.12 ❖ Приложение с динамической фильтрацией набора данных iris, созданное при помощи функционального программирования

Итоговый вид приложения показан на рис. 10.13.

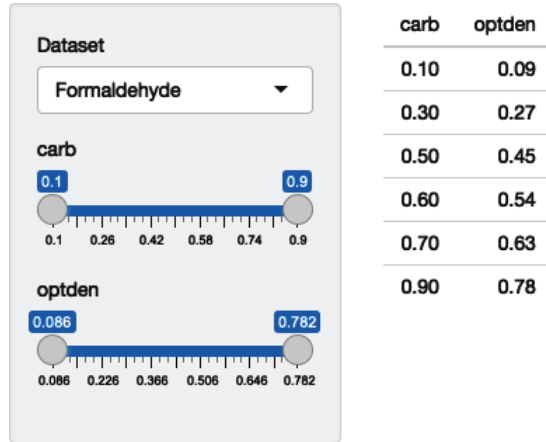


Рис. 10.13 ❖ Интерфейс пользователя, созданный динамически на основании полей из выбранного набора данных. Опробовать приложение онлайн можно по адресу <https://hadley.shinyapps.io/ms-filtering-final>

Диалоговые окна

В заключение я хотел бы упомянуть о технике создания динамического содержания применительно к *диалоговым окнам* (dialog box). Мы уже упоминали их в главе 8, но тогда строки для оповещения создавались статически. Однако, по причине того что функция `modalDialog()` вызывается внутри серверной функции, мы вполне можем динамически генерировать контент для нее по примеру функции `renderUI()`. Это бывает очень полезно, когда нужно дать пользователю сделать тот или иной выбор для продолжения работы с приложением.

Упражнения

Упражнение 1

Возьмите за основу в этом упражнении следующее простое приложение:

```
ui <- fluidPage(
  selectInput("type", "type", c("slider", "numeric")),
  uiOutput("numeric")
)

server <- function(input, output, session) {
  output$numeric <- renderUI({
    if (input$type == "slider") {
      sliderInput("n", "n", value = 0, min = 0, max = 100)
    } else {
      numericInput("n", "n", value = 0, min = 0, max = 100)
    }
  })
}
```



```
  })  
}
```

Как можно реализовать это приложение при помощи динамического управления видимостью? И как в этом случае вы будете сохранять текущие значения при смене элементов управления?

Упражнение 2

Опишите работу приведенного ниже приложения. Почему пароль исчезает после второго нажатия на кнопку **Enter password**?

```
ui <- fluidPage(  
  actionButton("go", "Enter password"),  
  textOutput("text")  
)  
  
server <- function(input, output, session) {  
  observeEvent(input$go, {  
    showModal(modalDialog(  
      passwordInput("password", NULL),  
      title = "Please enter your password"  
    ))  
  })  
  
  output$text <- renderText({  
    if (!isTruthy(input$password)) {  
      "No password"  
    } else {  
      "Password entered"  
    }  
  })  
}
```

Упражнение 3

Что произойдет в приложении из раздела «Введение» данной главы, если убрать функцию `isolate()` из выражения `value <- isolate(input$dynamic)`?

Упражнение 4

Добавьте поддержку столбцов типа *date* и *date-time* в функции `make_ui()` и `filter_var()`.

Упражнение 5

(Повышенной сложности) Если вы знакомы с системой объектно-ориентированного программирования (ООП) *S3*, подумайте, как можно заменить блоки `if` в функциях `make_ui()` и `filter_var()` с использованием *обобщенных функций* (generic function).

ЗАКЛЮЧЕНИЕ

Перед чтением данной главы вы были ограничены лишь средствами статического создания пользовательских интерфейсов до вызова функции `serv-`

ег()). Теперь вы знаете, как можно модифицировать интерфейс и создать его полностью в ответ на действия пользователя. При этом динамический пользовательский интерфейс способен значительно повысить сложность вашего приложения, так что не удивляйтесь, если начнете испытывать сложности с его отладкой. Всегда используйте самую простую технику, способную решить стоящую перед вами задачу, и следуйте советам из раздела «Отладка» главы 5.

В следующей главе мы будем говорить о закладках как способе сохранения текущего состояния приложения.

Глава 11

Закладки

Изначально у приложения Shiny есть один существенный недостаток по сравнению с другими страницами в интернете – вы не можете сохранить его в закладки, чтобы затем вернуться к прежнему состоянию или поделиться с кем-то ссылкой на сохраненное приложение. Причина в том, что по умолчанию Shiny не включает характеристики текущего состояния приложения в ссылку на него. К счастью, вы можете изменить такое поведение, приложив немного усилий, и в данной главе мы расскажем, как это сделать. Как обычно, начнем с загрузки пакета Shiny:

```
library(shiny)
```

ОСНОВНАЯ ИДЕЯ

Давайте рассмотрим пример простого приложения, которое вы хотите иметь возможность сохранять в закладках. Это приложение отображает на графике *фигуры Лиссажу* (Lissajous figure), повторяющие движения маятника. В результате работы приложения могут образовываться очень причудливые узоры, которыми было бы неплохо иметь возможность делиться с друзьями:

```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      sliderInput("omega", "omega", value = 1, min = -2, max = 2, step = 0.01),
      sliderInput("delta", "delta", value = 1, min = 0, max = 2, step = 0.01),
      sliderInput("damping", "damping", value = 1, min = 0.9, max = 1, step = 0.001),
      numericInput("length", "length", value = 100)
    ),
    mainPanel(
      plotOutput("fig")
    )
  )
)

server <- function(input, output, session) {
  t <- reactive(seq(0, input$length, length.out = input$length * 100))
```

```

x <- reactive(sin(input$omega * t() + input$delta) * input$damping ^ t())
y <- reactive(sin(t()) * input$damping ^ t())

output$fig <- renderPlot({
  plot(x(), y(), axes = FALSE, xlab = "", ylab = "", type = "l", lwd = 2)
}, res = 96)
}

```

На рис. 11.1 показан результат работы приложения.

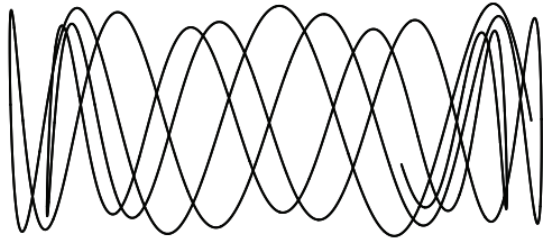
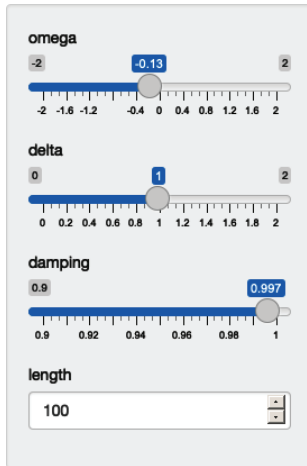


Рис. 11.1 ❖ Приложение позволяет генерировать любопытные фигуры с использованием модели маятника. Хотели бы поделиться ими с друзьями?

Чтобы приложение можно было сохранять в закладки, вам необходимо сделать три вещи.

1. Добавить в интерфейс специальную функцию `bookmarkButton()`. В результате будет создана кнопка, при нажатии на которую будет генерироваться ссылка для сохранения в закладках.
2. Превратить `ui` в полноценную функцию. Это нужно сделать, поскольку сохраненное в закладках приложение должно восстановить прежние значения: по сути, Shiny меняет аргумент `value` для каждого элемента ввода. Таким образом, у нас будет не один статический интерфейс пользователя, а несколько возможных интерфейсов, зависящих от параметров в ссылке. А значит, интерфейс должен быть преобразован в функцию.
3. Добавить аргумент `enableBookmarking = "url"` при вызове функции `shinyApp()`.

С учетом этих изменений мы получим следующий код приложения:

```

ui <- function(request) {
  fluidPage(
    sidebarLayout(
      sidebarPanel(

```

```

    sliderInput("omega", "omega", value = 1, min = -2, max = 2, step = 0.01),
    sliderInput("delta", "delta", value = 1, min = 0, max = 2, step = 0.01),
    sliderInput("damping", "damping", value = 1, min = 0.9, max = 1, step = 0.001),
    numericInput("length", "length", value = 100),
    bookmarkButton()
  ),
  mainPanel(
    plotOutput("fig")
  )
)
}

```

```
shinyApp(ui, server, enableBookmarking = "url")
```

Если вы откроете это приложение онлайн по адресу <https://hadley.shinyapps.io/ms-bookmark-url> и сделаете пару закладок, то обнаружите, что ссылки примут примерно следующий вид:

```
https://hadley.shinyapps.io/ms-bookmark-url/
?_inputs_&damping=1&delta=1&length=100&omega=1
```

```
https://hadley.shinyapps.io/ms-bookmark-url/
?_inputs_&damping=0.966&delta=1.25&length=100&omega=-0.54
```

```
https://hadley.shinyapps.io/ms-bookmark-url/
?_inputs_&damping=0.997&delta=1.37&length=500&omega=-0.9
```

Чтобы понять, что произошло, давайте возьмем первую ссылку и разобьем ее на части:

- `http://` – это протокол для доступа к приложению. Здесь всегда будет `http` или `https`;
- `hadley.shinyapps.io/ms-bookmark-url` указывает на расположение приложения;
- все, что идет после символа `?`, относится к параметрам. Друг от друга параметры отделяются знаком `&`, и если присмотреться к параметрам, можно легко определить значение каждого из них:
 - `damping=1`;
 - `delta=1`;
 - `length=100`;
 - `omega=1`.

Таким образом, создание закладки сводится к сохранению текущих значений элементов ввода в параметры ссылки. Если вы запускаете приложение локально, ссылки могут принимать примерно следующий вид:

```
http://127.0.0.1:4087/?_inputs_&damping=1&delta=1&length=100&omega=1
```

```
http://127.0.0.1:4087/?_inputs_&damping=0.966&delta=1.25&length=100&omega=-0.54
```

```
http://127.0.0.1:4087/?_inputs_&damping=0.997&delta=1.37&length=500&omega=-0.9
```

Большая часть ссылок совпадает, за исключением местоположения приложения – вместо `hadley.shinyapps.io/ms-bookmark-url` вы увидите что-то вроде `127.0.0.1:4087`. `127.0.0.1` представляет собой специальный IP-адрес, указывающий на локальный компьютер, а `4087` – случайным образом присвоенный номер порта. Обычно у разных приложений будут разные пути или IP-адреса, но при запуске приложений на локальном компьютере это будет не так.

Обновление ссылки

Вместо предоставления пользователю кнопки для создания закладки вы можете автоматически обновлять ссылку в адресной строке браузера. Это позволит пользователям использовать для создания закладок встроенные средства браузера и копировать ссылку при необходимости прямо из строки с адресом.

Автоматическое обновление ссылки в браузере потребует написания определенного шаблона в серверной функции:

```
# Автоматическое создание закладки при каждом изменении элемента ввода
observe({
  reactiveValuesToList(input)
  session$doBookmark()
})
# Обновление строки запроса
onBookmarked(updateQueryString)
```

Это приведет к следующим изменениям в функции `server()`:

```
server <- function(input, output, session) {
  t <- reactive(seq(0, input$length, length = input$length * 100))
  x <- reactive(sin(input$omega * t() + input$delta) * input$damping ^ t())
  y <- reactive(sin(t()) * input$damping ^ t())

  output$fig <- renderPlot({
    plot(x(), y(), axes = FALSE, xlab = "", ylab = "", type = "l", lwd = 2)
  }, res = 96)

  observe({
    reactiveValuesToList(input)
    session$doBookmark()
  })
  onBookmarked(updateQueryString)
}

shinyApp(ui, server, enableBookmarking = "url")
```

Посмотреть работу обновленного приложения можно по адресу <https://hadley.shinyapps.io/ms-bookmark-auto>. Поскольку ссылка теперь будет обновляться автоматически, кнопку создания закладки можно убрать из интерфейса пользователя.

Сохранение состояния в файл

До этого момента мы использовали аргумент `enableBookmarking = "url"`, позволяющий сохранять состояние приложения непосредственно в ссылке. Это очень простой способ создания закладок, и он будет работать везде, где бы вы ни развернули приложение Shiny, так что с него вполне можно начать. Однако, как вы сами понимаете, при наличии большого количества элементов ввода ссылка на приложение будет только увеличиваться в размерах, к тому же этот способ не дает возможности сохранить информацию о загруженном файле.

В качестве альтернативы вы можете использовать значение аргумента `enableBookmarking = "server"`, что позволит сохранить состояние приложения в файл `.rds` на сервере. В результате ссылка на приложение останется короткой и элегантной, но потребуется дополнительное место на сервере:

```
shinyApp(ui, server, enableBookmarking = "server")
```

В настоящее время сайт <https://www.shinyapps.io> не поддерживает создание закладок на стороне сервера, так что вы можете опробовать этот способ локально. В результате вы получите ссылки следующего вида:

```
http://127.0.0.1:4087/?_state_id_=0d645f1b28f05c97
```

```
http://127.0.0.1:4087/?_state_id_=87b56383d8a1062c
```

```
http://127.0.0.1:4087/?_state_id_=c8b0291ba622b69c
```

Одновременно с созданием закладок в вашей рабочей директории будут созданы следующие папки:

```
shiny_bookmarks/0d645f1b28f05c97
```

```
shiny_bookmarks/87b56383d8a1062c
```

```
shiny_bookmarks/c8b0291ba622b69c
```

Главные недостатки этого метода состоят в необходимости хранить дополнительные файлы на сервере и неочевидности того, как долго нужно их держать в доступе. Если ваше приложение содержит довольно много информации и вы никогда не будете чистить папки с закладками на сервере, со временем оно может занять все свободное место на диске. А если удалить файлы, какие-нибудь старые закладки перестанут работать.

Сложности при сохранении закладок

Метод автоматического сохранения закладок полагается на реактивный график. Элементы ввода заполняются сохраненными значениями, после чего запускаются все реактивные выражения и элементы вывода, в результате чего вы получаете исходный вид приложения, если реактивный график до-

статочно прямолинеен. Ниже описаны случаи, в которых нужно проявить большую осторожность:

- если в вашем приложении используются случайные числа, результаты при восстановлении могут отличаться от исходного вида приложения. Если вам критически важно, чтобы восстанавливались те же числа, вам стоит задуматься о том, как сделать процесс генерирования случайных величин воспроизводимым. Простейший способ это сделать – воспользоваться функцией `repeatable()`, подробности работы которой можно узнать в документации;
- если в вашем приложении присутствуют вкладки и вы хотите сохранять и восстанавливать активную вкладку, убедитесь, что вы передаете аргумент `id` при вызове функции `tabsetPanel()`;
- если в вашем приложении есть элементы ввода, не предназначенные для сохранения и восстановления в закладках (к примеру, они могут содержать конфиденциальную информацию, требующую ручного ввода), включите в серверную функцию вызов функции `setBookmarkExclude()`. Допустим, инструкция `setBookmarkExclude(c("secret1", "secret2"))` позволит исключить из списка сохраняемых полей элементы `secret1` и `secret2`;
- если вы вручную управляете реактивным состоянием при помощи объекта `reactiveValues()`, который мы будем подробнее обсуждать в главе 16, вам придется воспользоваться функциями обратного вызова `onBookmark()` и `onRestore()` для ручного сохранения и восстановления дополнительного состояния. За деталями можете обратиться к статье *Advanced Bookmarking* по адресу <https://shiny.rstudio.com/articles/advanced-bookmarking.html>.

УПРАЖНЕНИЯ

Упражнение 1

Разработайте приложение для визуализации результатов работы функции `ambient::noise_simplex()` (https://ambient.data-imaginist.com/reference/noise_simplex.html). Пользователь должен иметь возможность контролировать частоту (`frequency`), тип фрактала (`fractal`), множитель частоты между последовательными слоями шума (`lacunarity`) и усиление (`gain`), а также сохранять состояние приложения в закладках. Как вы добьетесь того, чтобы изображение полностью восстанавливалось при открытии приложения из закладок? Подумайте о том, что может означать аргумент `seed`.

Упражнение 2

Создайте простое приложение, в котором пользователь сможет загружать файл CSV и сохранять состояние в закладках. Загрузите несколько файлов и загляните в папку `shiny_bookmarks`. Как файлы соотносятся с закладками? Подсказка: вы можете использовать функцию `readRDS()` для просмотра содержимого создаваемых Shiny файлов.

ЗАКЛЮЧЕНИЕ

В данной главе мы рассмотрели различные способы сохранения состояния приложений в закладках. Это очень полезная возможность, позволяющая пользователям делиться с коллегами своими наработками. В следующей главе мы поговорим об использовании методов под общим названием *tidy evaluation* (сокращенно *tidy eval*) применительно к фреймворку Shiny. Принципы *tidy eval* представлены во многих функциях пакета *tidyverse*, и вам необходимо будет их освоить, если вы хотите позволить пользователям ваших приложений менять переменные, допустим, в конвейерах *dplyr* или графиках *ggplot2*.

Глава 12

Tidy eval

Используя фреймворк Shiny совместно с пакетом *tidyverse*, вы почти наверняка столкнетесь со сложностями в программировании с использованием методов *tidy eval*. Принципы *tidy eval* используются в пакете *tidyverse* повсеместно для облегчения исследования интерактивных данных, но за все приходится платить: ссылаться на переменные неявным образом становится труднее, и вместе с тем повышается общая сложность программирования.

В данной главе вы узнаете, как применять функции *ggplot2* и *dplyr* при написании приложений Shiny (если вы не используете пакет *tidyverse*, можете пропустить эту главу). Методы оборачивания функций *ggplot2* и *dplyr* в другие функции или пакеты немного отличаются и описаны, например, по следующим ссылкам: <https://ggplot2.tidyverse.org/dev/articles/ggplot2-in-packages.html>, <https://dplyr.tidyverse.org/articles/programming.html>. Итак, приступим:

```
library(shiny)
library(dplyr, warn.conflicts = FALSE)
library(ggplot2)
```

Предпосылки

Представьте, что вам необходимо разработать приложение, в котором будет возможность фильтровать числовую переменную для выбора строк, в которых значение этой переменной превышает определенный порог. Вы могли бы написать такой код:

```
num_vars <- c("carat", "depth", "table", "price", "x", "y", "z")

ui <- fluidPage(
  selectInput("var", "Variable", choices = num_vars),
  numericInput("min", "Minimum", value = 1),
  tableOutput("output")
)

server <- function(input, output, session) {
  data <- reactive(diamonds %>% filter(input$var > input$min))
  output$output <- renderTable(head(data()))
}
```

Variable

carat

Minimum

1

carat	cut	color	clarity	depth	table	price	x	y	z
0.23	Ideal	E	SI2	61.50	55.00	326	3.95	3.98	2.43
0.21	Premium	E	SI1	59.80	61.00	326	3.89	3.84	2.31
0.23	Good	E	VS1	56.90	65.00	327	4.05	4.07	2.31
0.29	Premium	I	VS2	62.40	58.00	334	4.20	4.23	2.63
0.31	Good	J	SI2	63.30	58.00	335	4.34	4.35	2.75
0.24	Very Good	J	VS2	62.80	57.00	336	3.94	3.96	2.48

Рис. 12.1 ❖ Приложение безуспешно пытается оставить в таблице только строки со значениями переменной, превышающими заданный пользователем минимум

Как видно по рис. 12.1, приложение запустилось без ошибок, но своего предназначения оно не выполняет – во всех выведенных строках значение переменной `carat` превышает единицу. Цель этой главы – помочь вам понять, почему это приложение не работает так, как ожидалось, и что заставляет *dplyr* воспринимать написанное нами выражение как `filter(diamonds, "carat" > 1)`.

Проблема состоит в *косвенной адресации* (indirection): обычно при использовании функций *tidyverse* вы указываете название переменной в явном виде, вызывая функцию. Но здесь нам необходимо применить неявную адресацию, поскольку одна переменная (`carat`) фактически заключена в другую (`input$var`).

Для вас такое выражение может быть интуитивно понятным, но формально оно может приводить в замешательство, ведь я пытаюсь использовать переменные для обозначения немного разных вещей. Согласитесь, эта двусмысленность может исчезнуть, если мы введем два новых понятия:

- *переменные окружения* (env-variable) – под переменными окружения мы будем понимать программные переменные, которые вы обычно создаете при помощи оператора присваивания `<-`. К примеру, `input$var` является переменной окружения;
- *переменные данных* (data-variable) – это «статистические» переменные, живущие внутри датафреймов. В нашем примере `carat` – это переменная данных.

С учетом этих новых понятий мы можем сформулировать нашу задачу более четко. Итак, у нас есть переменная данных `carat`, хранящаяся внутри переменной окружения `input$var`, и нам необходимо как-то об этом сообщить пакету *dplyr*. Существует два способа сделать это в зависимости от того,

с какой функцией вы имеете дело – с функцией, реализующей *маскирование данных*, или с функцией, следующей принципам *tidy-selection*.

МАСКИРОВАНИЕ ДАННЫХ

Функции, реализующие *маскирование данных* (data-masking), позволяют использовать переменные в «текущем» датафрейме без применения дополнительного синтаксиса. Эти принципы используются во многих функциях из пакета *dplyr*, включая `arrange()`, `filter()`, `group_by()`, `mutate()` и `summarise()`, а также в функции `aes()` из пакета *ggplot2*. Маскирование данных бывает очень полезно, поскольку позволяет использовать переменные данных без применения вспомогательного синтаксиса.

Введение

Давайте начнем с такого вызова функции `filter()`, в котором одновременно используется переменная данных (`carat`) и переменная окружения (`min`):

```
min <- 1
diamonds %>% filter(carat > min)
#> # A tibble: 17,502 x 10
#>   carat cut      color clarity depth table price     x     y     z
#>   <dbl> <ord>    <ord> <ord>    <dbl> <dbl> <int> <dbl> <dbl> <dbl>
#> 1  1.17 Very Good J      I1      60.2   61  2774  6.83  6.9  4.13
#> 2  1.01 Premium F      I1      61.8   60  2781  6.39  6.36  3.94
#> 3  1.01 Fair   E      I1      64.5   58  2788  6.29  6.21  4.03
#> 4  1.01 Premium H      SI2     62.7   59  2788  6.31  6.22  3.93
#> 5  1.05 Very Good J      SI2     63.2   56  2789  6.49  6.45  4.09
#> 6  1.05 Fair   J      SI2     65.8   59  2789  6.41  6.27  4.18
#> # ... with 17,496 more rows
```

Сравните эту запись с базовым синтаксисом языка R:

```
diamonds[diamonds$carat > min, ]
```

В большинстве (но не во всех¹) базовых функций R вы будете обращаться к переменным данных при помощи символа `$`. Это означает, что зачастую вам придется много раз повторять имя датафрейма в одном выражении, зато не будет проблем с определением того, где речь идет о переменных данных, а где о переменных окружения. Кроме того, в этом случае косвенная адресация применяется довольно естественным образом², поскольку вы можете

¹ Функция `dplyr::filter()` основана на базовой функции `base::subset()`. Функция `subset()` использует маскирование данных, но не посредством *tidy eval*, так что, к сожалению, техники, описываемые в данной главе, не могут быть к ней применены.

² В приложениях Shiny самой распространенной формой косвенной адресации является хранение имен переменных данных в реактивных значениях. Существует

хранить имя переменной данных внутри переменной окружения и переключаться с \$ на [[]:

```
var <- "carat"
diamonds[diamonds[[var]] > min, ]
```

Как можно добиться такого результата с *tidy eval*? Придется как-то возвращать в игру символ \$. К счастью, в функциях, реализующих маскирование данных, можно использовать префиксы `.data` и `.env` для адресации переменных данных и переменных окружения соответственно:

```
diamonds %>% filter(.data$carat > .env$min)
```

Теперь можно переключиться с \$ на [[]:

```
diamonds %>% filter(.data[[var]] > .env$min)
```

С учетом всех полученных знаний пришло время обновить функцию `server()` следующим образом:

```
num_vars <- c("carat", "depth", "table", "price", "x", "y", "z")

ui <- fluidPage(
  selectInput("var", "Variable", choices = num_vars),
  numericInput("min", "Minimum", value = 1),
  tableOutput("output")
)

server <- function(input, output, session) {
  data <- reactive(diamonds %>% filter(.data[[input$var]] > .env$input$min))
  output$output <- renderTable(head(data()))
}
```

На рис. 12.2 видно, что мы добились того, чего хотели, – в таблице остались только строки со значениями поля `carat`, превышающими единицу.

Теперь, когда вы усвоили основы, давайте разработаем пару более реалистичных, но по-прежнему достаточно простых приложений Shiny.

Пример: ggplot2

Применим озвученные выше идеи к построению динамической диаграммы рассеяния, на которой пользователь сможет сам выбирать, какие переменные выносить на оси `x` и `y`:

```
ui <- fluidPage(
  selectInput("x", "X variable", choices = names(iris)),
  selectInput("y", "Y variable", choices = names(iris)),
```

еще один способ применения косвенной адресации, когда при написании функций используется нотация с двойными фигурными скобками `{{ x }}`, о чем вы можете подробнее почитать по адресу <https://dplyr.tidyverse.org/articles/programming.html>.

```

    plotOutput("plot")
)

server <- function(input, output, session) {
  output$plot <- renderPlot({
    ggplot(iris, aes(.data[[input$x]], .data[[input$y]])) +
      geom_point(position = ggforce::position_auto())
  }, res = 96)
}

```

Variable

carat ▼

Minimum

1

carat	cut	color	clarity	depth	table	price	x	y	z
1.17	Very Good	J	I1	60.20	61.00	2774	6.83	6.90	4.13
1.01	Premium	F	I1	61.80	60.00	2781	6.39	6.36	3.94
1.01	Fair	E	I1	64.50	58.00	2788	6.29	6.21	4.03
1.01	Premium	H	SI2	62.70	59.00	2788	6.31	6.22	3.93
1.05	Very Good	J	SI2	63.20	56.00	2789	6.49	6.45	4.09
1.05	Fair	J	SI2	65.80	59.00	2789	6.41	6.27	4.18

Рис. 12.2 ❖ После применения нотации с использованием префиксов `.data` и `.env`, а также после замены `$` на `[[` наше приложение заработало так, как и ожидалось. Посмотреть приложение онлайн можно по адресу <https://hadley.shinyapps.io/ms-tidied-up>

Результат работы приложения показан на рис. 12.3.

Здесь мы применили функцию `ggforce::position_auto()`, чтобы наша геометрия работала одинаково хорошо при выборе непрерывных и дискретных значений на осях `x` и `y`. Также мы можем позволить пользователю выбрать тип геометрии. В следующем приложении мы использовали функцию `switch()` для создания реактивной геометрии, которая позже выводится на экран:

```

ui <- fluidPage(
  selectInput("x", "X variable", choices = names(iris)),
  selectInput("y", "Y variable", choices = names(iris)),
  selectInput("geom", "geom", c("point", "smooth", "jitter")),
  plotOutput("plot")
)

server <- function(input, output, session) {
  plot_geom <- reactive({

```

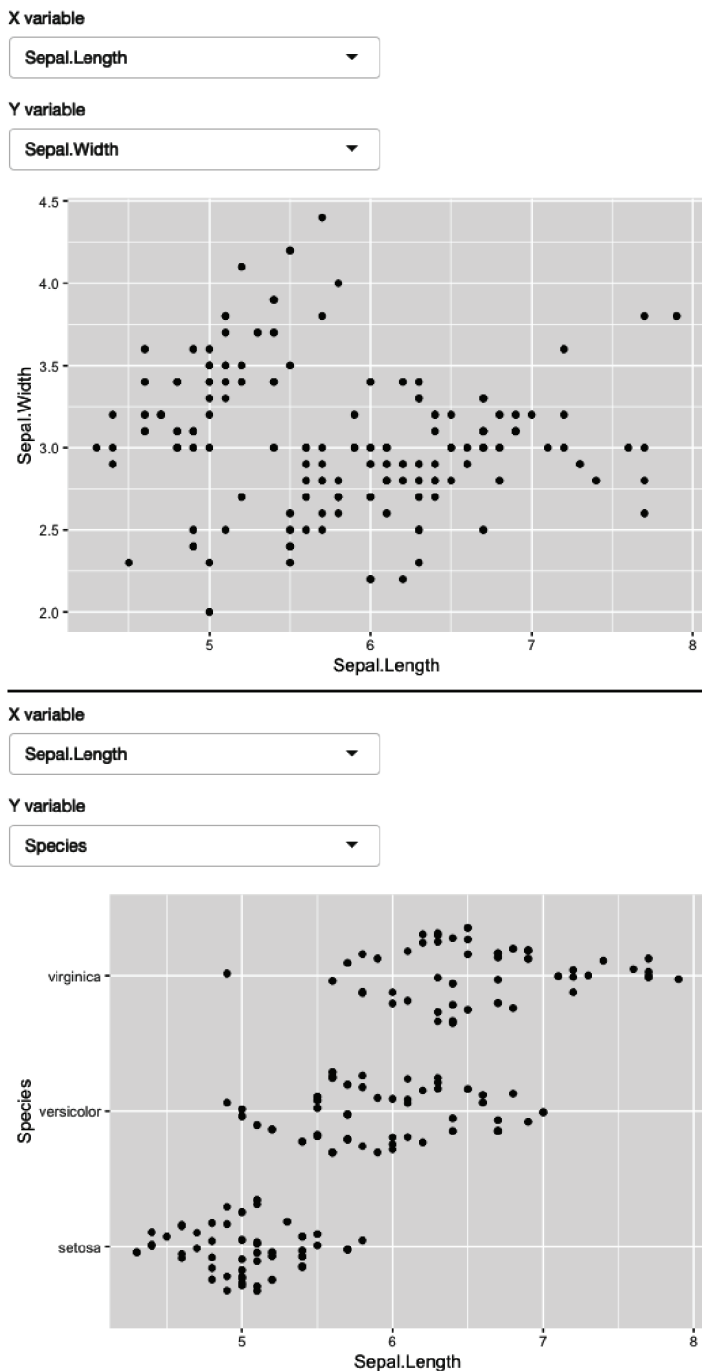


Рис. 12.3 ❖ Простое приложение с динамическим графиком и возможностью выбора пользователем переменных для осей x и y. Посмотреть приложение онлайн можно по адресу <https://hadley.shinyapps.io/ms-ggplot2/>

```

        switch(input$geom,
          point = geom_point(),
          smooth = geom_smooth(se = FALSE),
          jitter = geom_jitter()
        )
      })

  output$plot <- renderPlot({
    ggplot(iris, aes(.data[[input$x]], .data[[input$y]])) +
      plot_geom()
  }, res = 96)
}

```

В этом и состоит основная сложность программирования с возможностью выбора пользователем переменных – код становится все более сложным для обработки всех значений, которые могут быть выбраны.

Пример: dplyr

Эта же техника будет прекрасно работать и с пакетом *dplyr*. В следующем примере мы расширили наше предыдущее приложение, позволив пользователю выбрать переменную для фильтрации с минимальным значением для нее, а также переменную для сортировки:

```

ui <- fluidPage(
  selectInput("var", "Select variable", choices = names(mtcars)),
  sliderInput("min", "Minimum value", 0, min = 0, max = 100),
  selectInput("sort", "Sort by", choices = names(mtcars)),
  tableOutput("data")
)

server <- function(input, output, session) {
  observeEvent(input$var, {
    rng <- range(mtcars[[input$var]])
    updateSliderInput(
      session, "min",
      value = rng[[1]],
      min = rng[[1]],
      max = rng[[2]]
    )
  })

  output$data <- renderTable({
    mtcars %>%
      filter(.data[[input$var]] > input$min) %>%
      arrange(.data[[input$sort]])
  })
}

```

На рис. 12.4 показан внешний вид обновленного приложения.

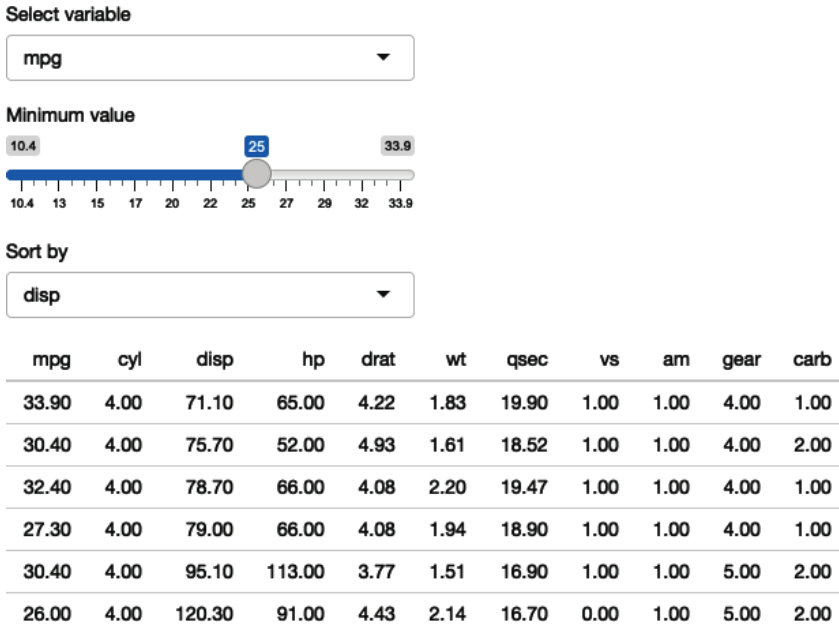


Рис. 12.4 ❖ Простое приложение с возможностью выбора полей для фильтрации и сортировки. Посмотреть приложение онлайн можно по адресу <https://hadley.shinyapps.io/ms-dplyr>

Ну, а дальше все зависит от вашего умения обращаться с префиксом `.data` и полета мысли. Например, мы можем предоставить пользователю возможность выбрать направление сортировки следующим образом:

```
ui <- fluidPage(
  selectInput("var", "Sort by", choices = names(mtcars)),
  checkboxInput("desc", "Descending order?"),
  tableOutput("data")
)

server <- function(input, output, session) {
  sorted <- reactive({
    if (input$desc) {
      arrange(mtcars, desc(.data[[input$var]]))
    } else {
      arrange(mtcars, .data[[input$var]])
    }
  })

  output$data <- renderTable(sorted())
}
```

С добавлением все новых и новых элементов управления ваш код будет становиться только сложнее, а интерфейс пользователя все труднее будет сделать одновременно удовлетворяющим всем вашим требованиям и удоб-

ным для работы. Именно поэтому я всегда был так сосредоточен на разработке программных инструментов для анализа данных: создавать хорошие пользовательские интерфейсы действительно очень сложно!

Пользовательские данные

Прежде чем перейти к принципам *tidy-selection*, немного поговорим о пользовательских данных. Взгляните на представленное ниже приложение, результат работы которого показан на рис. 12.5. Здесь пользователь может загрузить свой файл TSV, после чего выбрать переменную и выполнить по ней фильтрацию. Этот пример будет работать с подавляющим большинством входных данных:

```
ui <- fluidPage(
  fileInput("data", "dataset", accept = ".tsv"),
  selectInput("var", "var", character()),
  numericInput("min", "min", 1, min = 0, step = 1),
  tableOutput("output")
)

server <- function(input, output, session) {
  data <- reactive({
    req(input$data)
    vroom::vroom(input$data$datapath)
  })
  observeEvent(data(), {
    updateSelectInput(session, "var", choices = names(data()))
  })
  observeEvent(input$var, {
    val <- data()[[input$var]]
    updateNumericInput(session, "min", value = min(val))
  })

  output$output <- renderTable({
    req(input$var)

    data() %>%
      filter(.data[[input$var]] > input$min) %>%
      arrange(.data[[input$var]]) %>%
      head(10)
  })
}
```

Здесь есть одна небольшая проблема с использованием функции `filter()`. Давайте изолируем фрагмент кода с этой функцией, чтобы поработать с ней отдельно от приложения:

```
df <- data.frame(x = 1, y = 2)
input <- list(var = "x", min = 0)

df %>% filter(.data[[input$var]] > input$min)
#>   x y
#> 1 1 2
```

dataset

No file selected

var

min

Рис. 12.5 ❖ Удивительно отказоустойчивое приложение для фильтрации загруженных пользователем данных. Посмотреть приложение онлайн можно по адресу <https://hadley.shinyapps.io/ms-user-supplied>

Если вы поэкспериментируете с этим кодом, то поймете, что он прекрасно работает с большинством датафреймов. Но что будет, если мы добавим в датафрейм переменную с именем `input`?

```
df <- data.frame(x = 1, y = 2, input = 3)
df %>% filter(.data[[input$var]] > input$min)
#> Error: Problem with `filter()` input `..1`.
#> x $ operator is invalid for atomic vectors
#> i Input `..1` is `.data[["x"]] > input$min`.
```

Мы получили ошибку, поскольку функция `filter()` попыталась вычислить выражение `df$input$min`:

```
df$input$min
#> Error in df$input$min: $ operator is invalid for atomic vectors
```

Проблема, как вы понимаете, кроется в неопределенности доступа к переменным данных и окружения, что обусловлено тем, что маскирование данных при наличии обеих переменных предпочитает извлекать именно переменную данных. Эту проблему можно решить, используя префикс `.env`¹ для явного указания функции `filter()` на то, что переменную `min` следует искать исключительно в переменных окружения:

```
df %>% filter(.data[[input$var]] > .env$input$min)
#>   x y input
#> 1 1 2     3
```

¹ Вы можете предположить, что подобная проблема может возникнуть, если назвать переменные `.data` и `.env`. При столь маловероятном событии, когда у переменных будут такие необычные имена, вы сможете обращаться к ним явно при помощи синтаксиса `.data$.data` и `.data$.env`.

Об этой проблеме стоит беспокоиться только при работе с данными, предоставленными пользователем. Работая с собственными наборами данных, вы можете сами озаботиться тем, чтобы имена переменных данных и переменных окружения не пересекались. А если они пересекутся, вы очень быстро это обнаружите.

Почему бы не использовать базовый синтаксис R?

На данном этапе вы вполне могли задуматься, а почему бы не отказаться от этой функции `filter()` и не использовать эквивалентный код базового синтаксиса R:

```
df[df[[input$var]] > input$min, ]
#>   x y input
#> 1 1 2     3
```

Это вполне логичная мысль, и вы можете воспользоваться базовым R, но тогда вам самостоятельно придется реализовывать работу, которую за вас выполняет функция `filter()`, а именно:

- указать аргумент `drop = FALSE`, если `df` содержит единственный столбец. В противном случае вы получите на выходе вектор вместо датафрейма;
- использовать функцию `which()` или что-то подобное для исключения отсутствующих значений.

Из минусов стоит отметить, что вы не сможете выполнять групповую фильтрацию по примеру следующего синтаксиса: `df %>% group_by(g) %>% filter(n() == 1)`.

В самых простых ситуациях вполне можно использовать базовый синтаксис R без маскирования данных вместо богатого функционала *dplyr*, но, по моему мнению, одним из главных преимуществ *tidyverse* является стабильность работы в пограничных ситуациях. Я не хочу сгущать краски, но не стоит забывать о некоторых причудах, свойственных функциям из базового пакета R, из-за которых может случиться так, что в 95 % случаев ваш код будет работать, как и ожидалось, а в оставшихся 5 % даст сбой.

TIDY-SELECTION

Наряду с маскированием данных набор техник *tidy eval* включает в себя еще одну концепцию, именуемую *tidy-selection*. *Tidy-selection* представляет удобный способ для доступа к столбцам по их позициям, именам или типам. Эти принципы реализуются в таких функциях, как `dplyr::select()` и `dplyr::across()`, а также во многих функциях *tidyr*, например `pivot_longer()`, `pivot_wider()`, `separate()`, `extract()` и `unite()`.

Косвенная адресация

Для неявного обращения к переменным можно воспользоваться функциями *any_of()* или *all_of()*¹: обе принимают на вход символьный вектор в виде переменной окружения, содержащий имена переменных данных. Единственная разница между ними состоит в обработке имен переменных, не присутствующих на входе: функция *all_of()* выдаст ошибку, тогда как *any_of()* просто проигнорирует эту ситуацию.

В следующем простом приложении, в котором пользователь может выбрать любое количество переменных с использованием списка с мультивыбором, применяется функция *all_of()*:

```
ui <- fluidPage(
  selectInput("vars", "Variables", names(mtcars), multiple = TRUE),
  tableOutput("data")
)

server <- function(input, output, session) {
  output$data <- renderTable({
    req(input$vars)
    mtcars %>% select(all_of(input$vars))
  })
}
```

Tidy-Selection и маскирование данных

Работа с несколькими переменными значительно облегчается при использовании функций, поддерживающих *tidy-selection*: достаточно просто передать символьный вектор из имен переменных в функции *any_of()* или *all_of()*. Но было бы здорово, если бы мы могли делать то же самое в функциях с маскированием данных. В этом и состоит идея функции *across()*, появившейся в пакете *dplyr* версии 1.0.0. Она позволяет использовать принципы *tidy-selection* в функциях с маскированием данных.

Функция *across()* традиционно используется с одним или двумя аргументами. Первый аргумент служит для выбора переменных и бывает полезен в функциях вроде *group_by()* или *distinct()*. К примеру, в приложении, показанном на рис. 12.6, пользователь может выбрать любое количество переменных и подсчитать количество уникальных комбинаций их значений:

```
ui <- fluidPage(
  selectInput("vars", "Variables", names(mtcars), multiple = TRUE),
  tableOutput("count")
)

server <- function(input, output, session) {
  output$count <- renderTable({
    req(input$vars)
```

¹ В более старых версиях *tidyselect* и *dplyr* вы можете использовать функцию *one_of()*, обладающую той же семантикой, что и *any_of()*.

```
mtcars %>%
  group_by(across(all_of(input$vars))) %>%
  summarise(n = n(), .groups = "drop")
})
}
```

Variables

vs am

vs	am	n
0.00	0.00	12
0.00	1.00	6
1.00	0.00	7
1.00	1.00	7

Рис. 12.6 ❖ Выбор нескольких групп
с подсчетом уникальных комбинаций их значений.
Посмотреть приложение онлайн можно по адресу
<https://hadley.shinyapps.io/ms-across>

Второй аргумент представляет собой функцию или список функций, применяющихся к выбранным столбцам. Это бывает удобно при использовании функций `mutate()` и `summarise()`, в которых происходит определенное преобразование столбцов. В следующем приложении мы позволим пользователю выбрать переменные для группировки и переменные для суммирования с использованием средних значений:

```
ui <- fluidPage(
  selectInput("vars_g", "Group by", names(mtcars), multiple = TRUE),
  selectInput("vars_s", "Summarise", names(mtcars), multiple = TRUE),
  tableOutput("data")
)

server <- function(input, output, session) {
  output$data <- renderTable({
    mtcars %>%
      group_by(across(all_of(input$vars_g))) %>%
      summarise(across(all_of(input$vars_s), mean), n = n())
  })
}
```

ФУНКЦИИ `PARSE()` И `EVAL()`

В заключение я бы хотел ненадолго остановиться на комбинации функций `paste()` + `parse()` + `eval()`. Если вы не имеете понятия об этой последовательности, можете спокойно пропустить данный раздел. Для тех же, кто сталкивался с ней, я дам небольшой комментарий.

Использовать этот подход – большое искушение, поскольку для этого не нужно осваивать новые идеи. Но есть здесь и серьезные недостатки, кроющиеся в том, что при объединении строк очень легко получить неработающий код или код, выполняющий не то, что задумывалось. Это не так важно, если приложением будете пользоваться только вы, но в целом такую привычку хорошей не назовешь из-за возможных проблем с безопасностью. В главе 22 мы вернемся к этой теме и обсудим ее более подробно.

Конечно, вам может быть неприятно, если этот подход – единственный, с помощью которого вы способны решить конкретную задачу. Но я бы посоветовал вам потратить какое-то время и изучить другие варианты – без применения манипуляций со строками. Это поможет вам развить свои навыки и выйти на новый уровень программирования на языке R.

ЗАКЛЮЧЕНИЕ

В данной главе вы научились писать приложения, позволяющие пользователям выбирать переменные для дальнейшей обработки в функциях *tidyverse*, таких как `dplyr::filter()` и `ggplot2::aes()`. Это потребовало от вас понимания различий между переменными данных и переменными окружения. Вам наверняка потребуется какое-то время, чтобы освоиться с новой для вас концепцией, но в конечном счете это поможет вам открыть небывалую мощь в области анализа данных вместе с пакетом *tidyverse*.

Это была заключительная глава второй части книги под названием «Shiny в действии». Теперь, когда вы располагаете всем арсеналом инструментов для построения приложений Shiny любого уровня сложности, мы обратимся к теории, лежащей в основе фреймворка Shiny.

ОСВАИВАЕМ РЕАКТИВНОСТЬ

Прочитав две первые части этой книги, вы усвоили все базовые техники для написания богатых по функционалу приложений. Теперь пришло время поближе познакомиться с теорией и принципами реактивного программирования, лежащего в основе фреймворка Shiny:

- в главе 13 мы узнаем, зачем вообще понадобилась реактивная модель программирования, а также обратимся к истокам этой технологии за пределами языка R;
- глава 14 целиком и полностью будет посвящена реактивным графикам, служащим для определения моментов обновления реактивных компонентов приложения;
- в главе 15 мы поговорим о строительных блоках реактивной технологии, а именно о наблюдателях и отложенных во времени событиях;
- глава 16 научит вас обходить ограничения реактивных графиков с помощью функций `reactiveVal()` и `observe()`.

Разумеется, вам нет необходимости знать досконально все эти подробности при разработке обычных ежедневных приложений Shiny. Но понимание деталей поможет вам сразу писать корректный код, а в случае возникновения ошибок вам будет легче локализовать и устранить их.

Глава 13

Зачем нужна реактивность?

ВВЕДЕНИЕ

Первое впечатление от фреймворка Shiny – это какая-то магия! И это действительно так, ведь с его помощью вы можете создавать простые приложения быстро и легко – словно по мановению волшебной палочки. Но магия в области программирования часто оборачивается крушением иллюзий – без абсолютно четкого понимания глубинных процессов бывает не так просто понять, как поведет себя приложение во внешней среде, за границами простых примеров и демонстраций. А когда все идет не так, как вы ожидаете, ни о какой отладке не может быть и речи.

К счастью, Shiny – это белая магия. Как сказал Том Дейл (Tom Dale) о своем фреймворке *Ember.js*: «Мы творим чудеса. Но мы добрые волшебники, и поэтому наши чудеса разлагаются на разумные примитивы»¹. Именно к этой философии стремились разработчики фреймворка Shiny, особенно в отношении реактивного программирования. Разбирая реактивное программирование по косточкам, вы не увидите груды эвристических алгоритмов, особых подходов и «костылей». Вместо этого перед вами предстанет очень интеллектуальный, но в то же время достаточно прямолинейный механизм. А полное понимание модели реактивных принципов позволит увидеть, что рукава Shiny пусты, а вся магия основывается на простейших концепциях в комбинации с единообразными и согласованными подходами.

В данной главе мы объясним пользу реактивных принципов программирования, сперва попытавшись обойтись без них, а затем расскажем об истории развития этой технологии применительно к фреймворку Shiny.

¹ Steve Sanderson, «Rich JavaScript Applications – The Seven Frameworks (Throne of JS, 2012)» Steve Sanderson's Blog, August 1, 2012.

ЗАЧЕМ НУЖНО РЕАКТИВНОЕ ПРОГРАММИРОВАНИЕ?

Реактивное программирование (reactive programming) представляет собой стиль написания программного кода, основанный на изменяющихся с течением времени значениях и действиях, зависящих от этих изменений. Этот стиль программирования пришелся впору приложениям Shiny по причине их интерактивности: пользователи постоянно меняют значения элементов ввода (перетаскивают ползунки, вводят текст в поля, устанавливают переключатели и т. д.), тем самым иницилируя запуск процессов на сервере (чтение файлов CSV, создание подмножеств наборов данных, подгонка моделей), что ведет к обновлению элементов вывода (перерисовке графиков, заполнению таблиц и т. п.). Эта концепция в корне отличается от традиционных принципов программирования на R, по большей части основывающихся на статических данных.

Чтобы приложения Shiny приносили максимальную пользу, они просто обязаны использовать реактивные выражения в связке с элементами вывода, которые должны обновляться тогда и только тогда, когда изменяется значение соответствующих элементов ввода. Элементы ввода и вывода должны быть абсолютно четко синхронизированы, и просто недопустимо, чтобы приложение выполняло больше работы, чем необходимо. Чтобы понять, почему это все нужно реализовывать при помощи реактивных принципов, попробуем для начала обойтись вовсе без них – а вдруг получится?

Почему нельзя использовать переменные?

В каком-то смысле вы прекрасно умеете контролировать значения, меняющиеся с течением времени, при помощи старых добрых *переменных* (variable)! Переменные в R представлены значениями, которые могут меняться со временем, но в момент изменения ничего происходить не будет. Давайте рассмотрим простой пример перевода температуры из градусов Цельсия в градусы по Фаренгейту:

```
temp_c <- 10
temp_f <- (temp_c * 9 / 5) + 32
temp_f
#> [1] 50
```

Пока все в порядке, переменная `temp_c` равна 10, а `temp_f` – 50. Теперь изменим значение переменной `temp_c`:

```
temp_c <- 30
```

Но изменение переменной `temp_c` никак не повлияло на значение переменной `temp_f`:

```
temp_f
#> [1] 50
```

Итак, переменные способны менять свои значения с течением времени, но это никогда не происходит автоматически.

А как насчет функций?

Может, попробовать решить эту задачу при помощи функции?

```
temp_c <- 10
temp_f <- function() {
  message("Converting")
  (temp_c * 9 / 5) + 32
}
temp_f()
#> Converting
#> [1] 50
```

Получилась весьма странная функция, не имеющая аргументов и обращающаяся к переменной `temp_c` из внешнего окружения¹, но в целом это приемлемый код.

С помощью этого фрагмента кода мы решили первую задачу реактивного программирования – при каждом обращении к функции `temp_f()` мы будем получать актуальные данные:

```
temp_c <- -3
temp_f()
#> Converting
#> [1] 26.6
```

Но нам не удалось свести к минимуму количество вычислений. Дело в том, что при каждом обращении к функции `temp_f()` будет производиться пересчет, даже если значение переменной `temp_c` между вызовами не менялось:

```
temp_f()
#> Converting
#> [1] 26.6
```

В данном примере вычисление в функции оказалось очень простым, так что ничего страшного не случится, если оно будет производиться каждый раз. Но в общем случае в этом нет никакой необходимости: если входные данные не менялись, зачем пересчитывать выходные?

Событийно-ориентированное программирование

Раз ни переменные, ни функции нам не помогли, придется придумать что-то новое. Пару десятилетий назад мы бы, не думая, применили принципы *событийно-ориентированного программирования* (event-driven programming). Этот

¹ В языке R при определении значений переменных используется лексическая область видимости (lexical scoping).

метод написания программного кода следует простой и привлекательной парадигме: вы регистрируете *функции обратного вызова* (callback function), которые срабатывают в ответ на то или иное событие.

Можно реализовать простейший инструментарий событийно-ориентированного программирования при помощи пакета R6, как показано ниже. Здесь мы объявили класс DynamicValue с тремя важными методами: get() и set() для доступа и изменения значения и onUpdate() для объявления кода, который будет запускаться при изменении значения. Если вы незнакомы с пакетом R6, не беспокойтесь, а лучше сосредоточьтесь на приведенных ниже примерах:

```
DynamicValue <- R6::R6Class("DynamicValue", list(
  value = NULL,
  on_update = NULL,

  get = function() self$value,

  set = function(value) {
    self$value <- value
    if (!is.null(self$on_update))
      self$on_update(value)
    invisible(self)
  },

  onUpdate = function(on_update) {
    self$on_update <- on_update
    invisible(self)
  }
))
```

Если бы фреймворк Shiny был написан лет на пять раньше, фрагмент кода приложения мог бы выглядеть как-то так, а с переменной temp_c мы могли бы использовать оператор <- для обновления значения переменной temp_f при необходимости¹:

```
temp_c <- DynamicValue$new()
temp_c$onUpdate(function(value) {
  message("Converting")
  temp_f <- (value * 9 / 5) + 32
})

temp_c$set(10)
#> Converting
temp_f
#> [1] 50

temp_c$set(-3)
#> Converting
```

¹ <- называется в R глобальным оператором присваивания. В данном случае он устанавливает значение переменной temp_f в глобальном окружении вместо создания копии переменной temp_f внутри функции, как в случае с обычным оператором присваивания <=.

```
temp_f
#> [1] 26.6
```

Таким образом, применение принципов событийно-ориентированного программирования помогло нам решить вторую задачу, связанную с избыточными вычислениями. Но при этом возникла новая проблема, состоящая в излишней сложности отслеживания того, какие входные параметры какие вычисления запускают. В результате вы неизбежно встанете перед выбором между *правильностью* (correctness) данных, когда при любом изменении обновляются все без исключения элементы, и *производительностью* (performance), когда обновляются только нужные поля с риском забыть о каких-то важных элементах интерфейса, поскольку воплощать оба подхода одновременно крайне сложно.

Реактивное программирование

Реактивное программирование помогает легко и элегантно решить обе задачи путем объединения приведенных выше подходов. Давайте посмотрим на программный код с установкой специального режима Shiny `reactiveConsole(TRUE)`, позволяющего экспериментировать с реактивностью прямо в консоли:

```
library(shiny)
reactiveConsole(TRUE)
```

Как и в случае с событийно-ориентированным программированием, нам нужно как-то показать, что у нас есть переменная особого типа. В Shiny мы в этом случае создаем *реактивное значение* (reactive value) при помощи функции `reactiveVal()`. У реактивного значения есть специальный синтаксис для получения текущего значения (для этого необходимо вызвать его как функцию без аргументов) и установки нового (для этого значение передается функции в виде единственного аргумента):

```
temp_c <- reactiveVal(10) # создание
temp_c()                  # получение значения
#> [1] 10
temp_c(20)                # установка значения
temp_c()                  # получение значения
#> [1] 20
```

Теперь можно создать реактивное выражение, зависящее от нашего значения:

```
temp_f <- reactive({
  message("Converting")
  (temp_c() * 9 / 5) + 32
})
temp_f()
#> Converting
#> [1] 68
```

Как вы уже знаете, реактивное выражение будет отслеживать все изменения элементов, от которых зависит, так что при любом изменении значения переменной `temp_c` будет автоматически пересчитано значение `temp_f`:

```
temp_c(-3)
temp_c(-10)
temp_f()
#> Converting
#> [1] 14
```

Если же изменений в `temp_c()` не было, то и `temp_f()` не будет пересчитываться, а ее значение будет извлечено из кеша:

```
temp_f()
#> [1] 14
```

Реактивные выражения в Shiny обладают двумя важными характеристиками:

- они *ленивые* (*lazy*), а это означает, что никакие вычисления производиться не будут, пока выражение не будет вызвано;
- они *кешируемые* (*cached*), а значит, во второй и последующие разы вычисления производиться также не будут, а данные будут извлечены из кеша.

Об этих важных характеристиках реактивных выражений мы подробнее поговорим в главе 14.

КРАТКАЯ ИСТОРИЯ РЕАКТИВНОГО ПРОГРАММИРОВАНИЯ

Если вам интересно узнать больше о реактивном программировании применительно к другим языкам, немного истории не помешает. Истоки реактивного программирования прослеживаются в первой электронной таблице, созданной более 40 лет назад и названной VisiCalc (<https://en.wikipedia.org/wiki/VisiCalc>):

– Я представил себе волшебную классную доску, на которой, если стереть одно число и написать новое, все остальные числа автоматически изменятся – как в случае с обработкой текста, но с числами.

Дэн Бриклин (Dan Bricklin)

Электронные таблицы имеют много общего с реактивным программированием: вы определяете связи между ячейками посредством формул, и когда одна ячейка изменяется, все зависимые от нее ячейки также пересчитываются. Так что вы уже наверняка не раз сталкивались с принципами реактивного программирования, даже не зная об этом!

И хотя идея реактивного программирования витала в воздухе достаточно давно, вплоть до конца 1990-х эта тема не рассматривалась всерьез в обуче-

нии информатике. Исследования в области реактивного программирования начались с появлением FRAN (Functional Reactive ANimation – функционально-реактивные анимации) – новаторской системы внесения изменений с течением времени с функциональным языком программирования. Все это породило массу литературы, но на реальную практику программирования не оказало существенного влияния.

Повсеместное распространение реактивное программирование приобрело лишь в 2010-х годах благодаря фреймворкам пользовательских интерфейсов JavaScript. Первые фреймворки, в числе которых были *Knockout* (<https://knockoutjs.com>), *Ember* (<https://emberjs.com>) и *Meteor* (<https://www.meteor.com>), вдохновивший Джо Ченга на создание Shiny, показали, насколько реактивное программирование способно облегчить проектирование и разработку интерфейса пользователя. Всего за несколько следующих лет реактивное программирование вышло на ведущие роли в веб-разработке с такими популярными фреймворками, как *React* (<https://reactjs.org>), *Vue.js* (<https://vuejs.org>) и *Angular* (<https://angularjs.org>), – реактивными по своей сути, либо взаимодействующими с серверной частью, реализующей реактивные принципы.

Стоит упомянуть, что сам термин *реактивное программирование* является формальным. И хотя все пакеты, фреймворки и языки, реализующие парадигму реактивного программирования, нацелены на создание приложений, реагирующих на изменение значений, между собой они могут очень существенно отличаться в отношении терминологии и принципов разработки и реализации. Поскольку данная книга целиком и полностью посвящена Shiny, мы, говоря о реактивных принципах программирования, имеем в виду исключительно реализацию, принятую в этом фреймворке. Таким образом, если вы прочитаете литературу, посвященную реактивному программированию применительно к другим фреймворкам и технологиям, вряд ли вы сможете в полной мере воспользоваться прочитанным при разработке приложений Shiny. Для читателей, знакомых с реактивными принципами по другим реализациям, скажем, что подход, принятый в Shiny, наиболее близок к реализации фреймворков *Meteor* (<https://www.meteor.com>) и *MobX* (<https://mobx.js.org>) и значительно отличается от семейства *ReactiveX* (<http://reactivex.io>) и любых других реализаций, в описании которых встречается термин *функциональное реактивное программирование* (Functional Reactive Programming).

ЗАКЛЮЧЕНИЕ

Теперь, когда вы познакомились с историей реактивного способа программирования и осознали всю его важность, можно приступить к освоению теории, лежащей в его основе. В следующей главе вы сможете упрочить свое понимание принципов действия реактивных графиков, служащих для связывания реактивных значений и выражений с наблюдателями и контроля над тем, что и когда должно запускаться.

Глава 14

Реактивный график

ВВЕДЕНИЕ

Для понимания сути реактивных вычислений вам для начала необходимо разобраться в назначении *реактивного графика* (reactive graph). В данной главе мы погрузимся в детали этого графика, уделив особое внимание последовательности, в которой выполняются действия. Кроме того, вы познакомитесь с таким важным понятием, как инвалидация, описывающим процесс, позволяющий Shiny выполнять минимум работы. Также вы узнаете, что из себя представляет пакет *reactlog*, с помощью которого можно рисовать реактивные графики для реальных приложений.

Если с момента чтения главы 3 у вас прошло уже довольно много времени, я настоятельно советую перечитать ее, прежде чем двигаться дальше. В ней мы заложили основы того, о чем будем более подробно говорить в данной главе.

ПОШАГОВОЕ РЕАКТИВНОЕ ВЫПОЛНЕНИЕ

Для описания процесса реактивного выполнения мы будем использовать график, изображенный на рис. 14.1. Он состоит из трех элементов ввода, трех реактивных выражений и трех элементов вывода¹. Помните, что реактивные элементы ввода и выражения в совокупности именуются *реактивными поставщиками* (reactive producer), а реактивные выражения и элементы вывода – *реактивными потребителями* (reactive consumer).

Связи между компонентами на графике являются направленными, а стрелка указывает на распространение реактивности. Направление реактивных связей может вас удивить – легко подумать, что потребители одновременно зависят от нескольких поставщиков. Вскоре вы увидите, что процессы реактивных связей наиболее точно моделируются в обратном направлении.

¹ Везде, где вы видите упоминание об элементах вывода, вы можете также подразумевать наблюдателей. Главным отличием между ними является то, что некоторые невидимые элементы вывода никогда не будут пересчитываться. Подробно об этом мы поговорим в главе 15.

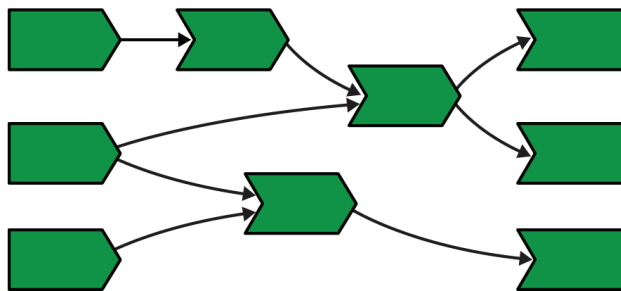


Рис. 14.1 ❖ Полный реактивный график вымышленного приложения с тремя элементами ввода, реактивными выражениями и элементами вывода

Приложение, лежащее в основе реактивного графика, не так важно само по себе, но если вам это поможет, можете представить, что за основу данного графика было взято следующее не самое полезное приложение:

```

ui <- fluidPage(
  numericInput("a", "a", value = 10),
  numericInput("b", "b", value = 1),
  numericInput("c", "c", value = 1),
  plotOutput("x"),
  tableOutput("y"),
  textOutput("z")
)

server <- function(input, output, session) {
  rng <- reactive(input$a * 2)
  smp <- reactive(sample(rng(), input$b, replace = TRUE))
  bc <- reactive(input$b * input$c)
  output$x <- renderPlot(hist(smp()))
  output$y <- renderTable(max(smp()))
  output$z <- renderText(bc())
}

```

Что ж, приступим!

Начало сессии

На рис. 14.2 показано состояние реактивного графика сразу после запуска приложения и первого вызова серверной функции.

Важные нюансы исходного реактивного графика:

- между элементами на графике нет связей, поскольку у Shiny нет информации о будущих взаимодействиях между реактивами;
- все реактивные выражения и элементы вывода находятся в своем исходном состоянии – они неактивны или *недействительны* (invalidated) и закрашены серым цветом, а это означает, что они еще ни разу не были запущены;

- реактивные элементы ввода находятся в состоянии готовности (закрашены зеленым цветом) – это значит, что их значения доступны для произведения расчетов.

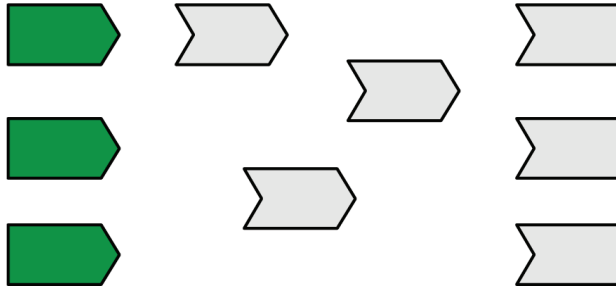


Рис. 14.2 ❖ Исходное состояние реактивного графика после запуска приложения.

Связей между объектами пока нет, и все реактивные выражения неактивны (закрашены серым). Всего на графике шесть поставщиков и шесть потребителей

Начало выполнения первого элемента вывода

Посмотрим на фазу начала выполнения приложения, показанную на рис. 14.3.

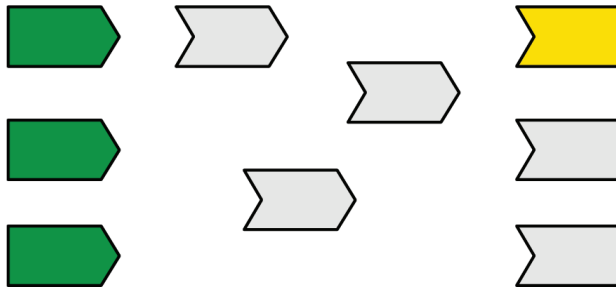


Рис. 14.3 ❖ Shiny начинает запуск с выполнения произвольного наблюдателя / элемента вывода, закрашенного желтым

На этой стадии Shiny выбирает неактивный элемент вывода и начинает его выполнение (показано желтым). Вам, должно быть, интересно, как именно Shiny делает выбор в пользу того или иного неактивного элемента для выполнения. Если говорить коротко, вы должны считать процесс выбора случайным: ваши наблюдатели и элементы вывода должны быть спроектированы для независимого запуска, и им должно быть без разницы, в какой последовательности будет осуществляться запуск¹.

¹ Если у вас есть наблюдатели, для которых критически важен порядок запуска, лучше попробовать перепроектировать приложение. В случае, если это невозможно, можно задать приоритет запуска при помощи аргумента `priority` функции `observe()`.

Чтение реактивного выражения

Выполнение элемента вывода может потребовать наличия значения у реактивного выражения, как показано на рис. 14.4.

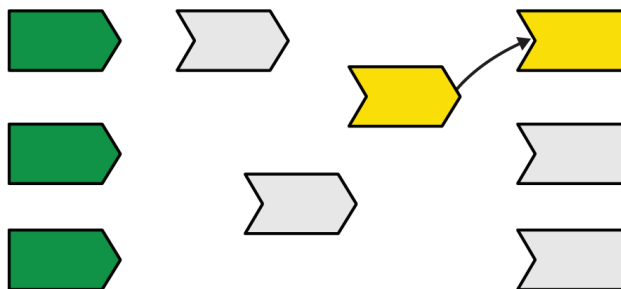


Рис. 14.4 ❖ Элементу вывода понадобилось значение реактивного выражения, так что Shiny приступает к его вычислению

Чтение реактивного выражения приводит к следующим изменениям на графике:

- реактивное выражение также нуждается в пересчете своего значения, так что оно запускается на выполнение (помечено желтым). Обратите внимание, что выполнение элемента вывода в этот момент продолжается: он ожидает рассчитанного значения от реактивного выражения, так что его выполнение не останавливается, как при обычном вызове функции в R;
- Shiny регистрирует связь между элементом вывода и реактивным выражением (на рисунке отмечена стрелкой). Направление стрелки очень важно: выражение фиксирует тот факт, что его результат используется элементом вывода, а элемент вывода, в свою очередь, не фиксирует, что использует расчеты, выполненные в выражении. Это очень тонкое различие, важность которого станет понятна позднее, когда мы будем говорить об инвалидации.

Чтение элемента ввода

Реактивное выражение, которое мы рассматриваем, читает значение реактивного элемента ввода. При этом между ними также устанавливается зависимость/связь, показанная стрелкой на рис. 14.5.

В отличие от реактивных выражений и элементов вывода, элементам ввода не требуется вычисление, так что значение из них может быть возвращено незамедлительно.

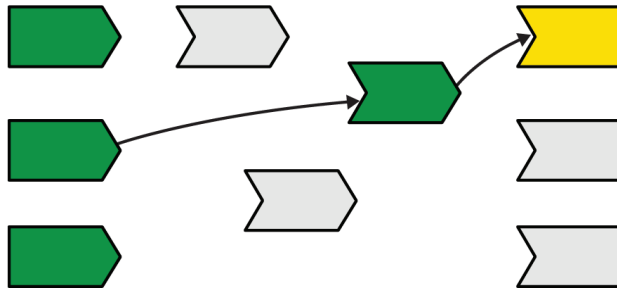


Рис. 14.5 ❖ Реактивное выражение также выполняет чтение реактивного значения, о чем свидетельствует новая стрелка

Окончание выполнения реактивного выражения

В нашем примере рассматриваемому реактивному выражению для выполнения расчета требуется результат другого реактивного выражения, которое, в свою очередь, выполняет чтение иного элемента ввода. Мы не будем досконально описывать этот процесс, поскольку рассказали о нем ранее. Результат вы можете посмотреть на рис. 14.6.

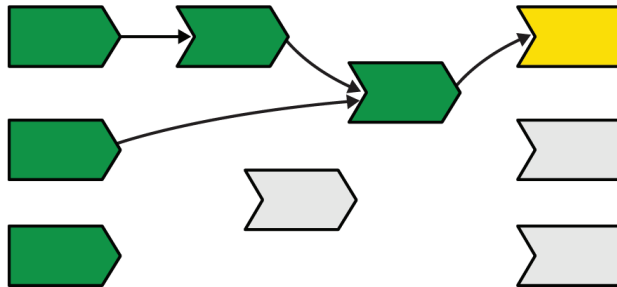


Рис. 14.6 ❖ Реактивное выражение завершило выполнение и теперь помечено зеленым цветом

Теперь, когда наше реактивное выражение закончило вычисление, оно окрашивается на графике зеленым цветом, что говорит о его готовности. В этот момент производится кеширование вычисленного результата, чтобы он мог возвращаться из кеша без дополнительных расчетов в случае, если элемент ввода не менялся.

Окончание выполнения элемента вывода

Когда реактивное выражение завершило вычисление и вернуло результат, он может быть использован для окончания выполнения соответствующего элемента ввода, с которого мы начали процесс пересчета приложения. На рис. 14.7 видно, что этот элемент был окрашен в зеленый цвет.

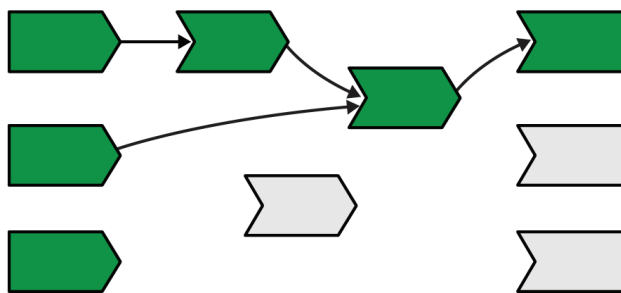


Рис. 14.7 ❖ Элемент вывода завершил выполнение и окрасился в зеленый цвет

Начало выполнения второго элемента вывода

После окончания выполнения первого элемента вывода Shiny приступает к выполнению следующего. На рис. 14.8 показано, что этот элемент окрасился в желтый цвет, что символизирует его выполнение. Начинается процесс чтения значения из реактивного поставщика.

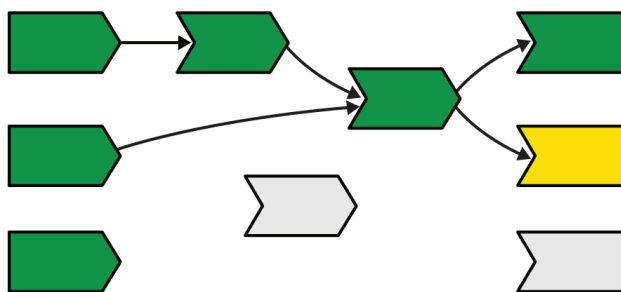


Рис. 14.8 ❖ Начинает выполняться следующий элемент вывода (окрашен желтым)

Реактивные выражения, завершившие свое выполнение, могут возвращать значение из кеша мгновенно, а неактивные реактивы будут запускать собственный реактивный график выполнения. Этот цикл будет выполняться до тех пор, пока все элементы вывода не окрасятся в зеленый цвет.

Завершение процесса выполнения, вывод расчетов

Итак, все элементы вывода завершили выполнение и перешли в состояние покоя, что видно по рис. 14.9.

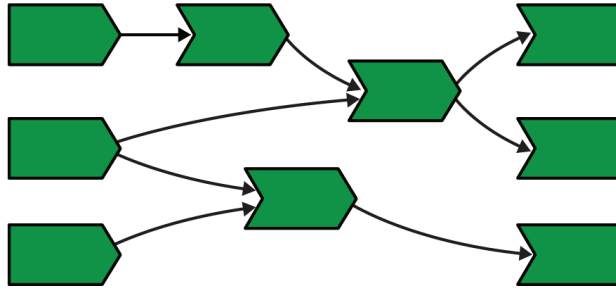


Рис. 14.9 ❖ Все элементы вывода и реактивные выражения завершили выполнение и окрасились в зеленый цвет

На этом этапе процесс реактивного выполнения завершается, и новые действия не потребуются вплоть до внешнего вмешательства в работу приложения. Таким вмешательством может быть, к примеру, перемещение ползунка в интерфейсе пользователем. Выражаясь терминами реактивного программирования, в данный момент сессия перешла в *состояние покоя* (at rest).

Давайте на мгновение остановимся и подумаем, что произошло. Мы прочитали какие-то элементы ввода, вычислили несколько значений и сгенерировали результаты для элементов вывода. Но более важно то, что мы обнаружили *связи* (relationship) между реактивными объектами. Теперь при изменении значения элемента ввода мы точно знаем, какие именно реактивы нам следует обновить.

ИЗМЕНЕНИЕ ЭЛЕМЕНТА ВВОДА

По окончании выполнения предыдущего шага наше приложение Shiny перешло в состояние покоя. Теперь допустим, что пользователь переместил ползунок в другое положение. В результате этого действия браузер посылает сообщение серверной функции, указывая на необходимость обновить соответствующие реактивные элементы ввода. В результате происходит переключение в *фазу инвалидации* (invalidation phase), состоящую из трех частей: инвалидация ввода, оповещение зависимостей и удаление текущих связей.

Инвалидация ввода

Фаза инвалидации начинается в момент изменения ввода/значения, после чего соответствующий элемент на графике окрашивается в серый цвет – принятый нами цвет для неактивных, инвалидированных элементов, что видно по рис. 14.10.

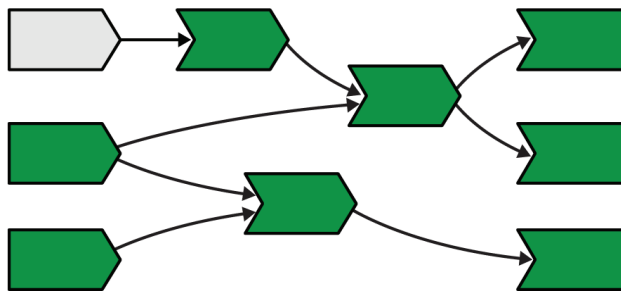


Рис. 14.10 ❖ Пользователь взаимодействует с приложением, переводя элемент ввода в разряд недействительных

Оповещение зависимостей

Теперь мы проследуем по стрелкам, которые были нарисованы ранее, по пути окрашивая каждый узел в серый цвет, а стрелки, по которым движемся, – в светло-серый. Результат показан на рис. 14.11.

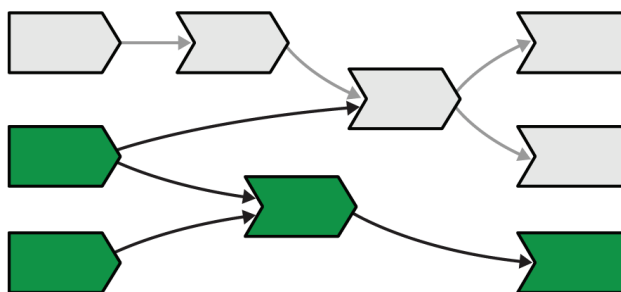


Рис. 14.11 ❖ Процесс инвалидации начинается с элемента ввода и проходит по всем стрелкам слева направо. Все стрелки по пути окрашиваются в светло-серый цвет

Удаление текущих связей

После этого все реактивные выражения и элементы вывода, ставшие недействительными, стирают на графике все входящие и исходящие из них стрелки. На этом завершается фаза инвалидации, а реактивный график приобретает вид, показанный на рис. 14.12.

Стрелки, исходящие из узла, представляют собой одноразовые уведомления, которые срабатывают в следующий раз после изменения значения. После срабатывания их функция считается выполненной, в связи с чем их можно удалить.

Менее очевидна причина, по которой мы удаляем стрелки, входящие в инвалидированные узлы, даже если узлы, из которых они выходят, не утратили актуальности. Хотя эти стрелки представляют оповещения, которые еще

не сработали, потерявшему действительность узлу они больше не нужны: реактивный потребитель нуждается только в тех связях, которые могут его инвалидировать, а это уже произошло.

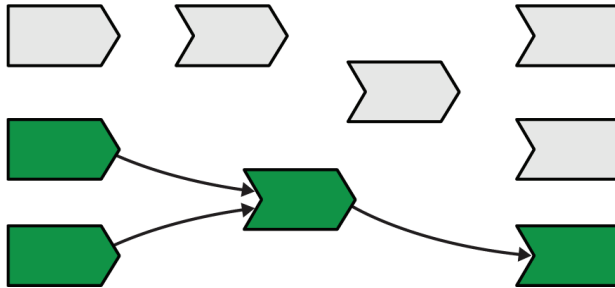


Рис. 14.12 ❖ Инвалидированные узлы «забывают» все свои прежние зависимости, чтобы они могли быть обнаружены заново

Кажется очень странным, что мы так бережно прорисовывали все эти стрелки, а потом просто взяли и выбросили их в мусорное ведро! Но в этом и состоит суть модели реактивного программирования Shiny: хотя эти стрелки и *были* для нас так важны, *теперь* они утратили свою актуальность. Единственный способ обеспечить нашему реактивному графику правильность и точность – стирать устаревшие стрелки и позволить Shiny восстанавливать их вокруг узлов при повторном выполнении. Мы вернемся к этой важной теме далее в данной главе в разделе, посвященном динамизму.

Повторное выполнение

Теперь мы оказались в очень похожей ситуации на ту, когда выполняли второй элемент вывода, но со смесью из действительных и недействительных реактивов. Значит, пришло время сделать то же самое, что и в тот раз: выполнить инвалидированные элементы вывода по одному за раз. Отправная точка для этого процесса показана на рис. 14.13.

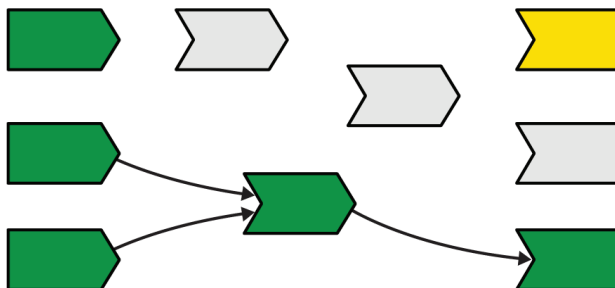


Рис. 14.13 ❖ Повторное выполнение происходит так же, как и первоначальное, но работы на этот раз будет поменьше

Я не буду в подробностях описывать весь оставшийся процесс, но поверьте мне на слово, что итогом цикла повторного выполнения будет полностью зеленый реактивный график в состоянии покоя. Главным преимуществом такого подхода является то, что Shiny пришлось проделать минимум работы по восстановлению реактивного графика. А именно были пересчитаны только те элементы вывода, которые подверглись изменениям в связи со сменой входных значений.

Упражнения

Упражнение 1

Нарисуйте реактивный график для следующей серверной функции и объясните, почему реактивы не запускаются:

```
server <- function(input, output, session) {
  sum <- reactive(input$x + input$y + input$z)
  prod <- reactive(input$x * input$y * input$z)
  division <- reactive(prod() / sum())
}
```

Упражнение 2

Следующий реактивный график моделирует операцию с длительным вычислением при помощи функции `Sys.sleep()`:

```
x1 <- reactiveVal(1)
x2 <- reactiveVal(2)
x3 <- reactiveVal(3)

y1 <- reactive({
  Sys.sleep(1)
  x1()
})

y2 <- reactive({
  Sys.sleep(1)
  x2()
})

y3 <- reactive({
  Sys.sleep(1)
  x2() + x3() + y2() + y2()
})

observe({
  print(y1())
  print(y2())
  print(y3())
})
```

Сколько времени будет пересчитываться график при изменении значения `x1`? А как насчет `x2` или `x3`?

Упражнение 3

Что произойдет, если попытаться создать реактивный график с циклом?

```
x <- reactiveVal(1)
y <- reactive(x + y())
y()
```

ДИНАМИЗМ

Из предыдущего раздела вы узнали, что Shiny легко «забывает» связи между реактивными компонентами, которые так усиленно строил. Это добавляет реактивности Shiny определенной динамики, поскольку она может меняться прямо во время работы приложения. Присущий реактивному графику *динамизм* (dynamism) очень важен, и мы разберем его на следующем простом примере:

```
ui <- fluidPage(
  selectInput("choice", "A or B?", c("a", "b")),
  numericInput("a", "a", 0),
  numericInput("b", "b", 10),
  textOutput("out")
)

server <- function(input, output, session) {
  output$out <- renderText({
    if (input$choice == "a") {
      input$a
    } else {
      input$b
    }
  })
}
```

Вы могли бы подумать, что реактивный график представленного выше кода будет выглядеть так, как показано на рис. 14.14.

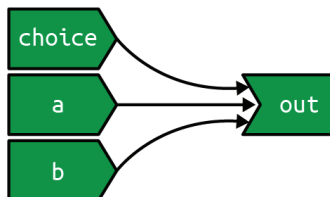


Рис. 14.14 ❖ Если бы Shiny анализировал реактивность статически, на реактивном графике всегда были бы связаны узлы choice, a и b с элементом out

Но, поскольку Shiny динамически перестраивает график каждый раз, когда элемент вывода становится недействительным, он на самом деле будет вы-

глядеть, как показано на рис. 14.15 слева или справа, в зависимости от значения элемента ввода `input$choice`. Это гарантирует Shiny выполнение минимума работы при потере элементом ввода действительности. В этом случае, если в выпадающем списке, представленном переменной `input$choice`, будет выбран вариант `b`, значение элемента ввода `input$a` не будет оказывать влияния на элемент вывода `output$out`, а значит, не будет необходимости в пересчете.

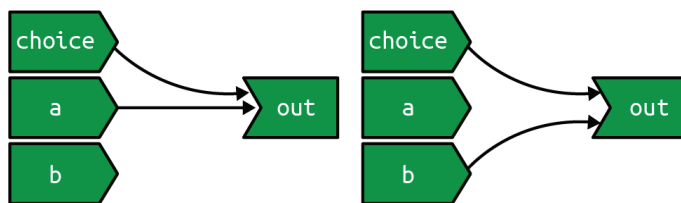


Рис. 14.15 ❖ Реактивный график Shiny отличается динамикой, так что элемент `out` будет связан с узлом `choice`, а также либо с элементом `a` (слева), либо с элементом `b` (справа)

Стоит отметить, как это сделал Цзян Иньдэн (Yindeng Jiang) в своем блоге по адресу <https://shinydata.wordpress.com/2015/02/02/a-few-things-i-learned-about-shiny-and-reactive-programming>, что достаточно немного изменить код, чтобы вывод на графике всегда зависел и от `a`, и от `b`:

```
output$out <- renderText({
  a <- input$a
  b <- input$b

  if (input$choice == "a") {
    a
  } else {
    b
  }
})
```

Это не повлияло бы на вывод в обычном коде R, но в данном случае разница будет, поскольку реактивные зависимости устанавливаются при чтении значения из параметра `input`, а не при использовании этого значения.

ПАКЕТ REACTLOG

Рисовать реактивные графики от руки – это очень полезная практика, помогающая лучше понять принципы работы приложений и парадигму реактивного программирования в целом. Но это бывает очень трудозатратно делать для реальных приложений Shiny со множеством элементов. Было бы неплохо воспользоваться каким-нибудь специальным инструментом, который построил бы график за вас на основании информации, имеющейся

у Shiny. Такой инструмент есть, и называется он *reactlog* (<https://rstudio.github.io/reactlog>). Этот пакет генерирует так называемый *реактивный лог* (*reactlog*), показывающий изменение реактивного графика во времени.

Чтобы воспользоваться пакетом *reactlog*, нужно для начала установить его, активировать при помощи инструкции `reactlog::reactlog_enable()` и запустить приложение. После этого вам будут доступны две опции:

- во время работы приложения нажмите сочетание клавиш **Cmd+F3** (**Ctrl+F3** на Windows), чтобы показать реактивный лог, сгенерированный на данный момент;
- после закрытия приложения выполните инструкцию `shiny::reactlog-Show()` для просмотра полного лога сессии.

В реактивном логе из этого пакета применяются те же условные графические обозначения, которые мы использовали для описания процесса работы реактивного графика в предыдущем разделе, так что, рассматривая его, вы почувствуете себя как дома. Основное различие состоит в том, что в реактивном логе отображаются все зависимости, даже те, которые в данный момент не задействуются, для поддержания созданного макета. Неактивные в текущий момент связи, которые в прошлом или будущем могут быть активированы, показываются в виде тонких пунктирных линий.

На рис. 14.16 показан реактивный график, сгенерированный при помощи пакета *reactlog* для приложения из предыдущего раздела. Вас может удивить присутствие на графике трех дополнительных реактивных элементов ввода (`clientData$output_x_height`, `clientData$output_x_width` и `clientData$pixelratio`), не упоминающихся в исходном коде. Они появились по причине наличия неявной зависимости графика от размера элемента вывода – при его изменении график должен быть перерисован.

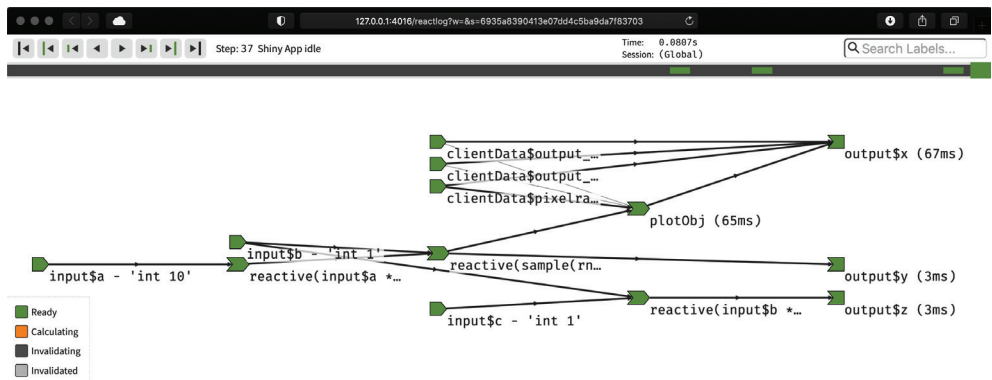


Рис. 14.16 ❖ Реактивный график для приложения, сгенерированный при помощи пакета *reactlog*

Обратите внимание, что у элементов ввода и вывода есть свои имена, тогда как у реактивных выражений и наблюдателей их нет, так что они помечены с помощью их содержимого. Для улучшения внешнего вида графика вы можете использовать аргумент `label` в функциях `reactive()` и `observe()` –

в этом случае для этих элементов будут отображаться заданные вами метки. Вы даже можете использовать значки эмодзи с целью выделения особенно важных элементов.

ЗАКЛЮЧЕНИЕ

В данной главе мы подробно описали принципы работы реактивных графиков. Также вы узнали, что из себя представляет фаза инвалидации и что она не активирует мгновенный пересчет, а просто помечает реактивных потребителей как недействительных, чтобы они могли быть пересчитаны при необходимости. Цикл инвалидации является очень важным процессом функционирования реактивного графика, поскольку он помогает очистить ранее обнаруженные зависимости, чтобы они могли быть выстроены заново, – это позволяет сделать реактивный график полностью динамическим.

Теперь у вас есть полное представление о принципах реактивного программирования, и в следующей главе мы подробнее поговорим о структурах данных, лежащих в основе реактивных значений, выражений и элементов вывода, а также затронем тему инвалидации элементов по времени.

Глава 15

Строительные блоки реактивного программирования

В данный момент вы уже гораздо лучше понимаете принципы и методологию построения реактивных графиков и обладаете в этом небольшим опытом. Пришло время узнать, как парадигма реактивного программирования встроена в язык R. Существует три основных строительных блока реактивного программирования: реактивные значения, реактивные выражения и наблюдатели. О первых двух блоках мы говорили уже достаточно много, так что в этой главе уделим больше внимания именно наблюдателям и элементам вывода, которые, как вы узнаете, являются особым типом наблюдателей. Также вы узнаете еще о двух инструментах для управления ходом выполнения реактивного графика: изоляции и инвалидации элементов по времени.

В данной главе мы снова будем использовать реактивную консоль, чтобы можно было экспериментировать с реактивностью напрямую, без необходимости каждый раз запускать приложение Shiny. Для начала загрузим библиотеку Shiny и активируем консоль для возможности проводить интерактивные эксперименты:

```
library(shiny)
reactiveConsole(TRUE)
```

РЕАКТИВНЫЕ ЗНАЧЕНИЯ

Существует два типа *реактивных значений* (reactive value):

- единичное реактивное значение, объявляемое при помощи функции `reactiveVal()`;
- список реактивных значений, объявляемый как `reactiveValues()`.

Эти типы отличаются интерфейсами чтения и записи значений:

```
x <- reactiveVal(10)
x()      # чтение
```

```
#> [1] 10
x(20)      # запись
x()        # чтение
#> [1] 20

г <- reactiveValues(x = 10)
г$х        # чтение
#> [1] 10
г$х <- 20  # запись
г$х        # чтение
#> [1] 20
```

Очень жаль, что настолько похожие объекты имеют столь разные подходы к записи и чтению данных, но не в наших силах привести их к какому-то единому стандарту. И все же, несмотря на серьезные внешние различия, ведут себя эти объекты одинаково, так что вы можете выбрать, какой из них использовать, основываясь на своих личных предпочтениях в отношении синтаксиса. В этой книге я в основном использую запись `reactiveValues()`, поскольку она обладает более понятным синтаксисом, в то время как в своих собственных проектах отдаю предпочтение инструкции `reactiveVal()`, потому что при ее применении легче понять, что происходит нечто странное.

Важно отметить, что оба типа реактивных выражений характеризуются *ссылочной семантикой* (reference semantics). Большинство объектов в R придерживаются семантики *копирования при изменении* (copy-on-modify), означающей, что если вы присвоите одно и то же значение двум переменным, связь между ними будет разорвана при первом изменении одной из них:

```
a1 <- a2 <- 10
a2 <- 20
a1 # не изменилась
#> [1] 10
```

Но в случае с реактивными значениями картина будет иной – они всегда поддерживают ссылки на одно и то же значение, так что изменение любой копии приведет к изменению всех значений:

```
b1 <- b2 <- reactiveValues(x = 10)
b1$х <- 20
b2$х
#> [1] 20
```

В главе 16 мы поговорим о том, зачем вам может понадобиться создавать собственные реактивные значения. В целом же большинство реактивных значений, с которыми вы будете сталкиваться, будут поступать из аргумента `input` функции `server()`. Они отличаются от значений, которые вы создаете сами при помощи функции `reactiveValues()`, тем, что доступны только для чтения, – вы не можете менять эти значения, поскольку Shiny автоматически обновляет их на основании действий пользователя в браузере.

Упражнения

Упражнение 1

Чем отличаются два приведенных ниже списка реактивных значений? Сравните синтаксис для чтения и записи отдельных реактивных значений:

```
l1 <- reactiveValues(a = 1, b = 2)
l2 <- list(a = reactiveVal(1), b = reactiveVal(2))
```

Упражнение 2

Разработайте и проведите небольшой эксперимент, подтверждающий ссылочную семантику реактивных значений, созданных при помощи функции `reactiveVal()`.

РЕАКТИВНЫЕ ВЫРАЖЕНИЯ

Помните, мы говорили, что реактивы имеют два важных свойства: они ленивые и кешируемые. Это означает, что они выполняются только тогда, когда действительно нужны, а при повторном вызове возвращают заранее сохраненное значение.

Есть еще две важные особенности, о которых мы пока не говорили: как реактивные выражения реагируют на ошибки и как внутри них работает функция `on.exit()`.

Ошибки

Реактивные выражения кешируют ошибки точно так же, как и значения. Рассмотрим следующий пример:

```
r <- reactive(stop("Error occurred at ", Sys.time(), call. = FALSE))
r()
#> Error: Error occurred at 2021-03-05 16:38:19
```

Если подождать секунду или две, то вы увидите, что выйдет такая же ошибка, как и раньше:

```
Sys.sleep(2)
r()
#> Error: Error occurred at 2021-03-05 16:38:19
```

Кроме того, на реактивном графике ошибки тоже ведут себя так же, как значения: они распространяются по графику точно таким же образом, как и обычные значения. Единственное различие состоит в том, что происходит, когда ошибка возникает в элементе вывода или наблюдателе:

- ошибка в элементе вывода будет отображаться в приложении¹;
- ошибка в наблюдателе приведет к завершению текущей сессии. Если вы не хотите, чтобы это происходило, необходимо заключить код в блок `try()` или `tryCatch()`.

Эти же принципы лежат в основе функции `req()`, которую мы обсуждали в главе 8 и которая генерирует особый тип ошибки². Это приводит к остановке процесса, выполняемого наблюдателем или элементом вывода. По умолчанию элементы вывода в этом случае будут сброшены в исходное пустое состояние, а при использовании инструкции `req(..., cancelOutput = TRUE)` сохранят свое текущее состояние.

on.exit()

Вы можете воспринимать выражение `reactive(x())` как укороченную запись инструкции `function() x()` с автоматическим добавлением ленивого вызова и возможности кеширования. Это может быть важно, если вы хотите узнать, как внутренне реализован фреймворк Shiny, но также означает, что вы можете использовать функции, которые работают только внутри других функций. Наиболее важной из таких функций является `on.exit()`, которая позволяет вам запускать код по окончании выполнения реактивного выражения вне зависимости от того, успешно ли выражение вернуло ошибку или завершило свое выполнение в связи с ее возникновением. Именно это позволяет функции `on.exit()` осуществлять оповещения о выполнении процесса, о чем мы говорили в главе 8.

Упражнения

Упражнение 1

Используйте пакет `reactlog` для наблюдения за распространением ошибки в следующем приложении, подтверждая тот факт, что ошибки проходят по реактивному графику так же, как и значения:

```
ui <- fluidPage(
  checkboxInput("error", "error?"),
  textOutput("result")
)

server <- function(input, output, session) {
  a <- reactive({
    if (input$error) {
      stop("Error!")
    }
  })
}
```

¹ По умолчанию текст ошибки будет выводиться целиком. Вы можете сократить его, воспользовавшись приемом очистки (sanitizing), описанным по этому адресу: <https://shiny.rstudio.com/articles/sanitize-errors.html>.

² Чисто технически – *особое условие* (custom condition).

```

    } else {
      1
    }
  })

  b <- reactive(a() + 1)
  c <- reactive(b() + 1)
  output$result <- renderText(c())
}

```

Упражнение 2

Измените предыдущее приложение таким образом, чтобы использовалась функция `req()` вместо `stop()`. Убедитесь, что события распространяются также, как раньше. Что произойдет при применении аргумента `cancelOutput`?

НАБЛЮДАТЕЛИ И ЭЛЕМЕНТЫ ВЫВОДА

Наблюдатели (observer) и элементы вывода (output) являются концевыми узлами (terminal node) на реактивном графике. Они отличаются от реактивных выражений в двух аспектах:

- они *нетерпеливы (eager)* и *забывчивы (forgetful)* – выполняются они сразу, как только получают такую возможность, и не помнят своих предыдущих действий. Кроме того, нетерпеливость наблюдателей и элементов вывода является заразной – если они используют реактивные выражения, они также будут запущены;
- значения, возвращаемые наблюдателями, игнорируются, поскольку они используются с функциями, выполняющими побочные действия, вроде `cat()` или `write.csv()`.

В основе наблюдателей и элементов вывода лежит один и тот же инструмент, а именно функция `observe()`, определяющая блок кода, запускающийся всякий раз при обновлении использующегося реактивного выражения или значения. Обратите внимание, что наблюдатели запускаются незамедлительно при их создании с целью определения своих реактивных зависимостей:

```

y <- reactiveVal(10)
observe({
  message("`y` is ", y())
})
#> Warning: Error in y: could not find function "y"

y(5)
y(4)

```

В данной книге я редко буду использовать функцию `observe()`, поскольку она является низкоуровневым инструментом, лежащим в основе более дружелюбной функции `observeEvent()`. Обычно вы должны использовать имен-

но функцию `observeEvent()`, за исключением случаев, когда с ее помощью невозможно добиться желаемых результатов. В этой книге мы рассмотрим единственный пример, где необходимо будет использовать функцию `observe()`, и будет это в главе 16.

Функция `observe()` также лежит в основе реактивных элементов вывода. Такие элементы представляют собой особый тип наблюдателей, обладающий двумя важными свойствами:

- их определение происходит в момент присваивания их объекту `output` (например, при помощи инструкции `output$text <- ...` создается наблюдатель);
- они обладают ограниченными возможностями для определения того, что в данный момент они невидимы (к примеру, если находятся на неактивной вкладке), чтобы не было необходимости в их пересчете¹.

Важно понимать, что функция `observe()` и реактивные элементы вывода *не делают* что-то, а *создают* что-то, что впоследствии выполняет какие-то действия при необходимости. Это, к примеру, поможет вам осознать, что происходит в приведенном ниже примере:

```
x <- reactiveVal(1)
y <- observe({
  x()
  observe(print(x()))
})
#> Warning: Error in x: could not find function "x"
x(2)
x(3)
```

Каждое изменение переменной `x` будет приводить к запуску наблюдателя. Внутри наблюдателя происходит вызов функции `observe()`, что приводит к созданию *другого* наблюдателя. Так что каждый раз, когда значение `x` меняется, появляется другой наблюдатель и значение переменной выводится еще раз.

Можно принять за правило создавать наблюдателей и элементы вывода исключительно на верхнем уровне серверной функции. Если вы пытаетесь встраивать их в другие функции или создавать наблюдателей внутри элементов вывода, остановитесь и подумайте – возможно, есть и более правильный вариант. В сложных приложениях подобные ошибки бывает очень трудно отследить, и для этого лучше всего воспользоваться пакетом *reactlog*, – если в районе наблюдателей или элементов вывода вы обнаружите неожиданные движения, откатитесь назад и посмотрите, что приводит к их появлению.

¹ В редких случаях вам может быть необходимо обновлять даже те элементы, которые находятся на скрытых вкладках. Вы можете воспользоваться аргументом *suspendWhenHidden* функции *outputOptions()*, чтобы отказаться от приостановки обновления для конкретного элемента вывода.

ИЗОЛИРОВАНИЕ КОДА

В завершение главы давайте рассмотрим еще два важных инструмента для контроля за инвалидацией элементов на реактивном графике. В данном разделе поговорим о функции `isolate()`, лежащей в основе `observeEvent()` и `eventReactive()` и позволяющей вам избегать создания нежелательных реактивных зависимостей. В следующем разделе вы узнаете о функции `invalidateLater()`, с помощью которой можно осуществлять инвалидацию реактивных элементов по расписанию.

`isolate()`

Наблюдатели зачастую объединяются с реактивными значениями для отслеживания их состояния с течением времени. Давайте рассмотрим простой пример, позволяющий узнать, сколько раз изменилось значение `x`:

```
r <- reactiveValues(count = 0, x = 1)
observe({
  r$x
  r$count <- r$count + 1
})
```

Если вы запустите это приложение, то тут же окажетесь в бесконечном цикле, поскольку наблюдатель получит реактивную зависимость как от `x`, так и от `count`, а так как внутри наблюдателя происходит изменение `count`, он будет немедленно перезапущен.

К счастью, в арсенале разработчиков приложений Shiny есть функция `isolate()`, позволяющая получить доступ к текущему реактивному значению или выражению без установления зависимости от него:

```
r <- reactiveValues(count = 0, x = 1)
class(r)
#> [1] "rv_flush_on_write" "reactivevalues"
observe({
  r$x
  r$count <- isolate(r$count) + 1
})
#> Warning: Error in <observer>: object 'r' not found

r$x <- 1
r$x <- 2
r$count
#> [1] 0

r$x <- 3
r$count
#> [1] 0
```

Как и в случае с функцией `observe()`, в большинстве ситуаций вам не придется использовать функцию `isolate()` напрямую, поскольку есть две удобные функции: `observeEvent()` и `eventReactive()`.

observeEvent() и eventReactive()

Рассматривая предыдущий код, вы могли подумать, а почему бы не использовать функцию `observeEvent()`, о которой мы говорили еще в третьей главе книги:

```
observeEvent(x(), {
  count(count() + 1)
})
```

И действительно, это можно было сделать, поскольку выражение `observeEvent(x, y)` полностью эквивалентно записи `observe({x; isolate(y)})`. Такая инструкция позволяет элегантно отвязать то, что вы слушаете, от того, что делаете. Функция `eventReactive()` выполняет аналогичные действия для реактивов: запись `eventReactive(x, y)` эквивалентна выражению `reactive({x; isolate(y)})`.

У функций `observeEvent()` и `eventReactive()` есть дополнительные аргументы, позволяющие изменить их действия по умолчанию:

- изначально обе функции игнорируют события, возвращающие `NULL` (или в случае с кнопками действий – `0`). Используйте аргумент `ignoreNULL = FALSE` для обработки значений `NULL`;
- по умолчанию обе функции будут по разу запущены в момент их создания. Воспользуйтесь аргументом `ignoreInit = TRUE`, чтобы пропустить этот запуск;
- для функции `observeEvent()` вы можете также использовать аргумент `once = TRUE` для однократного запуска обработчика.

Применять эти аргументы вы будете не так часто, но знать о них необходимо, чтобы при случае быстро найти подробности их использования в документации.

Упражнения

Упражнение 1

Напишите для приведенного ниже приложения серверную функцию, обновляющую элемент вывода `out` в соответствии с элементом ввода `x` только по нажатию на кнопку:

```
ui <- fluidPage(
  numericInput("x", "x", value = 50, min = 0, max = 100),
  actionButton("capture", "capture"),
  textOutput("out")
)
```

ИНВАЛИДАЦИЯ ПО ВРЕМЕНИ

Функция `isolate()` позволяет сократить количество инвалидаций реактивного графика. В данном разделе мы рассмотрим функцию `invalidateLater()`, выполняющую ровно обратное действие, а именно устанавливающую инвалидацию графика, когда никакие данные не менялись. В главе 3 мы уже рассматривали пример обновления по расписанию с использованием функции `reactiveTimer()`, а сейчас поговорим о функции `invalidateLater()`, лежащей в ее основе.

Функция `invalidateLater(ms)` позволяет сделать любой реактивный потребитель недействительным по прошествии `ms` миллисекунд. Она бывает полезна при создании анимации и подключении к источникам данных, находящимся за пределами реактивного фреймворка Shiny, которые могут меняться со временем. К примеру, следующий код служит для создания реактива, который будет генерировать десяток чисел с нормальным распределением каждые полсекунды¹:

```
x <- reactive({
  invalidateLater(500)
  rnorm(10)
})
```

А приведенный ниже наблюдатель будет увеличивать нарастающий итог на случайное число:

```
sum <- reactiveVal(0)
observe({
  invalidateLater(300)
  sum(isolate(sum()) + runif(1))
})
```

В следующих разделах вы узнаете, как можно использовать функцию `invalidateLater()` для чтения меняющихся данных с диска, как не позволить ей угодить в бесконечный цикл и еще несколько важных нюансов об инвалидации данных.

Опрос

Примером полезного применения функции `invalidateLater()` является подключение Shiny к данным, изменяющимся за пределами R. Допустим, вы бы могли использовать следующий реактив для чтения файла CSV каждую секунду:

```
data <- reactive({
  on.exit(invalidateLater(1000))
```

¹ При условии, что мы используем его в наблюдателе или элементе вывода. В противном случае он навсегда останется неизменным.

```
    read.csv("data.csv")
  })
```

Таким образом мы подключили изменяющиеся данные к реактивному графику Shiny, но здесь есть один существенный недостаток: при инвалидации реактива вы также делаете недействительными все нижестоящие потребители, так что, даже если данные не изменятся, вся работа, связанная с этими потребителями, должна будет выполняться.

Для решения этого неудобства в Shiny предусмотрена функция *reactivePoll()*, принимающая на вход две функции: первая (более дешевая в плане ресурсов) выполняет быструю проверку на предмет того, что данные изменились, а вторая (более дорогая) предназначена для осуществления основных расчетов. Таким образом, мы можем переписать предыдущий код с использованием функции *reactivePoll()* следующим образом:

```
server <- function(input, output, session) {
  data <- reactivePoll(1000, session,
    function() file.mtime("data.csv"),
    function() read.csv("data.csv")
  )
}
```

Здесь мы воспользовались функцией *file.mtime()*, возвращающей время последнего изменения файла, в качестве быстрой проверки на то, стоит ли нам перезагружать файл.

Чтение файла при его изменении – довольно распространенная задача, так что в Shiny для этой операции предусмотрена отдельная вспомогательная функция, принимающая на вход имя файла и функцию для чтения:

```
server <- function(input, output, session) {
  data <- reactiveFileReader(1000, session, "data.csv", read.csv)
}
```

Если вам необходимо считывать изменяющиеся данные из других источников, таких как база данных, то придется писать собственный код с использованием функции *reactivePoll()*.

Долгоиграющие реактивы

Если вам предстоит выполнять длительные вычисления, необходимо задать себе один важный вопрос: а когда стоит вызывать функцию *invalidateLater()*? Давайте рассмотрим следующий реактив:

```
x <- reactive({
  invalidateLater(500)
  Sys.sleep(1)
  10
})
```

Предположим, Shiny начнет выполнение реактива в нулевой временной отметке и запросит инвалидацию на отметке 500 мс. На выполнение реактиву требуется 1000 мс. На временной отметке 1000 мс реактив становится недействительным и должен быть пересчитан, что провоцирует новую инвалидацию. Как результат мы застряли в бесконечном цикле.

Но если вызвать функцию `invalidateLater()` в конце выполнения реактива, он будет инвалидирован через 500 мс после завершения работы, так что вызываться он будет каждые 1500 мс:

```
x <- reactive({
  on.exit(invalidateLater(500), add = TRUE)
  Sys.sleep(1)
  10
})
```

В этом состоит причина преимущественного использования функции `invalidateLater()` вместо более простой `reactiveTimer()`, которую мы применяли ранее: она дает вам полный контроль над тем, когда точно будет производиться инвалидация.

Точность таймера

Количество миллисекунд, указанное в функции `invalidateLater()`, – это лишь вежливая просьба, но никак не требование. R может заниматься другими делами в тот момент, когда вы попросите об инвалидации, так что вашему запросу, возможно, придется подождать. Фактически это означает, что затребованное время является возможным минимумом, а сама инвалидация может и затянуться. В большинстве случаев это не имеет значения, поскольку столь малые задержки не должны сказаться на быстродействии приложения. Но если присутствует риск накопления мелких ошибок, вы должны постараться точно рассчитать затраченное время и использовать его при корректировке вычислений.

Например, в следующем фрагменте кода мы рассчитаем дистанцию на основании скорости и затраченного времени. Вместо того чтобы предполагать, что `invalidateLater(100)` всегда будет давать точную задержку в 100 мс, я вычислил затраченное время и использовал полученный результат при расчете расстояния:

```
velocity <- 3
r <- reactiveValues(distance = 1)

last <- proc.time()[[3]]
observe({
  cur <- proc.time()[[3]]
  time <- last - cur
  last <- cur
```



```
r$distance <- isolate(r$distance) + velocity * time
invalidateLater(100)
})
```

Если для вас не так важна точная анимация, вы можете проигнорировать свойственные функции `invalidateLater()` колебания. Просто помните о том, что этой функции мы передаем нашу просьбу, а не требование.

Упражнения

Упражнение 1

Почему этот реактив никогда не будет выполнен? В своем объяснении вы должны опираться на реактивный график и инвалидацию:

```
server <- function(input, output, session) {
  x <- reactive({
    invalidateLater(500)
    rnorm(10)
  })
}
```

Упражнение 2

Если вы знакомы с языком запросов SQL, используйте функцию `reactive-Poll()` для чтения из вымышленной таблицы *Results* всякий раз, когда в нее добавляется новая строка. Вы можете предположить, что в таблице *Results* имеется поле `timestamp`, в котором хранится дата и время добавления записи.

Заключение

В данной главе вы поближе познакомились с основными строительными блоками, лежащими в основе фреймворка Shiny: реактивными значениями и выражениями, а также наблюдателями и вычислениями по времени. Теперь обратим свое внимание на особенную комбинацию реактивных значений и наблюдателей, которая поможет нам преодолеть некоторые ограничения реактивного графика.

Глава 16

Отхождение от графика

ВВЕДЕНИЕ

Фреймворк реактивного программирования Shiny призван автоматически определять необходимое и достаточное количество работы, которое нужно выполнить для обновления всех элементов вывода в приложении при изменении элементов ввода. В то же время у этого фреймворка есть свои серьезные ограничения, и иногда очень нужно вырваться из его рамок, чтобы сделать что-то рискованное, но крайне необходимое.

В данной главе вы узнаете, как можно комбинировать функции `reactiveValues()` и `observe()/observeEvent()` для обратной связи правой части реактивного графика с левой. Мы рассмотрим очень мощные техники, позволяющие получить полный контроль за всеми частями реактивного графика. В то же время эти приемы таят в себе некоторую опасность, поскольку позволяют приложению выполнять необязательную или нежелательную работу. Но еще более важно, что с применением этих техник ваше приложение рискует погрязнуть в бесконечных циклах обновлений, конца которым не будет.

Если вам покажутся интересными идеи, которые я озвучу в этой главе, вам, вероятно, захочется ознакомиться с пакетами *shinySignals* (<https://github.com/hadley/shinySignals>) и *rxtools* (<https://github.com/jcheng5/rxtools>). Это экспериментальные пакеты, предназначенные для исследования принципов реактивности высшего порядка, т. е. программного создания реактивных элементов на основе других реактивов. Я бы не советовал вам использовать их в рабочих приложениях, но почитать исходный код этих пакетов может быть поучительно. Как обычно, начнем с загрузки пакета Shiny:

```
library(shiny)
```

ЧТО НЕ ОХВАТЫВАЕТ РЕАКТИВНЫЙ ГРАФИК?

В главе 14 мы много говорили о том, что происходит, когда пользователь делает элемент ввода недействительным. Вы как автор приложения можете провести инвалидацию элементов еще двумя следующими способами:

- вызвав функцию семейства `update` и передав ей аргумент `value`. В результате браузеру будет послано сообщение для изменения значения элемента ввода, о чем будет оповещен и R;
- изменив реактивное значение при помощи функций `reactiveVal()` или `reactiveValues()`.

Важно понимать, что в обоих случаях реактивная зависимость между реактивным значением и наблюдателем создана не будет. И хотя эти действия приводят к инвалидации реактивного графика, они не фиксируются при помощи новых связей¹.

Чтобы лучше усвоить эти идеи, давайте рассмотрим следующий пример приложения, реактивный график которого приведен на рис. 16.1:

```
ui <- fluidPage(
  textInput("nm", "name"),
  actionButton("clr", "Clear"),
  textOutput("hi")
)

server <- function(input, output, session) {
  hi <- reactive(paste0("Hi ", input$nm))
  output$hi <- renderText(hi())
  observeEvent(input$clr, {
    updateTextInput(session, "nm", value = "")
  })
}
```

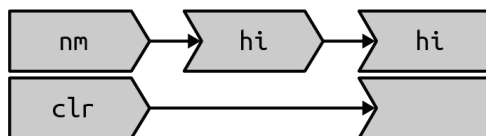


Рис. 16.1 ❖ На реактивном графике не отражена связь между безымянным наблюдателем и элементом ввода `nm`: эта зависимость находится вне пределов его видимости

Что на самом деле происходит, когда вы нажимаете на кнопку очистки?

1. Элемент ввода `input$clr` переходит в недействительное состояние, что приводит к инвалидации наблюдателя.
2. Наблюдатель пересчитывается, воссоздавая зависимость с элементом `input$clr` и сообщая браузеру о необходимости изменения значения элемента ввода.
3. Браузер меняет значение элемента `nm`.
4. Элемент ввода `input$nm` утрачивает действительность, что приводит к инвалидации реактивного выражения `hi()`, а затем и элемента вывода `output$hi`.
5. Элемент `output$hi` пересчитывается, вынуждая `hi()` сделать то же самое.

¹ С целью выполнения отладки пакет *reactlog* позволяет определить и вывести на графике эти связи при изменении реактивных значений из наблюдателей, но в Shiny эта информация не используется.

Ни одно из приведенных действий не оказывает влияния на реактивный график приложения, так что он остается таким же, как было показано на рис. 16.1, и не отображает связь между наблюдателем и элементом ввода `input$nm`.

ПРАКТИЧЕСКИЕ ПРИМЕРЫ

Теперь давайте рассмотрим несколько практических примеров, где можно осуществить сочетание функции `reactiveValues()` с `observeEvent()` или `observe()` для решения задач, которые иначе решить было бы очень сложно, если не невозможно. Кроме того, представленные здесь шаблоны вы можете легко использовать в собственных приложениях.

Один элемент вывода изменяется посредством нескольких элементов ввода

Для начала поставим достаточно нетривиальную задачу: к примеру, нам нужно, чтобы одно и то же текстовое поле менялось под влиянием разных событий¹:

```
ui <- fluidPage(
  actionButton("drink", "drink me"),
  actionButton("eat", "eat me"),
  textOutput("notice")
)

server <- function(input, output, session) {
  r <- reactiveValues(notice = "")
  observeEvent(input$drink, {
    r$notice <- "You are no longer thirsty"
  })
  observeEvent(input$eat, {
    r$notice <- "You are no longer hungry"
  })
  output$notice <- renderText(r$notice)
}
```

В следующем примере мы немного усложним задачу. Теперь речь пойдет о двух кнопках, которые позволяют увеличивать и уменьшать значение поля. Мы использовали функцию `reactiveValues()` для хранения текущего значения поля, а функцию `observeEvent()` применили для увеличения или уменьшения реактивного значения при нажатии на соответствующую кнопку. Дополнительная сложность здесь состоит в том, что новое значение `n()` зависит от предыдущего значения:

¹ Это отчасти похоже на оповещения, о которых мы говорили в главе 8.

```

ui <- fluidPage(
  actionButton("up", "up"),
  actionButton("down", "down"),
  textOutput("n")
)

server <- function(input, output, session) {
  r <- reactiveValues(n = 0)
  observeEvent(input$up, {
    r$n <- r$n + 1
  })
  observeEvent(input$down, {
    r$n <- r$n - 1
  })
  output$n <- renderText(r$n)
}

```

На рис. 16.2 показан реактивный график для этого примера. Обратите внимание, что график не содержит связей от наблюдателей обратно к реактивному значению `n`.

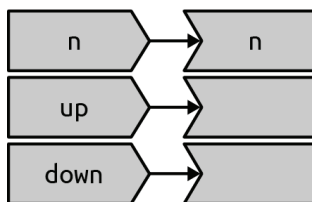


Рис. 16.2 ❖ На реактивном графике отсутствуют обратные связи от наблюдателей ко входящему значению

Накапливание ввода

Похожая реализация может получиться при необходимости накапливать данные для поддержки ввода информации. Главное отличие здесь состоит в том, что мы используем функцию `updateTextInput()` для сброса значения текстового поля после нажатия пользователем на кнопку:

```

ui <- fluidPage(
  textInput("name", "name"),
  actionButton("add", "add"),
  textOutput("names")
)

server <- function(input, output, session) {
  r <- reactiveValues(names = character())
  observeEvent(input$add, {
    r$names <- c(input$name, r$names)
    updateTextInput(session, "name", value = "")
  })
}

```

```

    })

    output$names <- renderText(r$names)
  }

```

Можно расширить этот пример, добавив кнопку удаления и гарантировав отсутствие дубликатов при добавлении данных следующим образом:

```

ui <- fluidPage(
  textInput("name", "name"),
  actionButton("add", "add"),
  actionButton("del", "delete"),
  textOutput("names")
)

server <- function(input, output, session) {
  r <- reactiveValues(names = character())
  observeEvent(input$add, {
    r$names <- union(r$names, input$name)
    updateTextInput(session, "name", value = "")
  })
  observeEvent(input$del, {
    r$names <- setdiff(r$names, input$name)
    updateTextInput(session, "name", value = "")
  })

  output$names <- renderText(r$names)
}

```

Приостановка анимации

Еще одной распространенной практикой является создание пары кнопок для приостановки и продолжения некоего повторяющегося события. В данном примере мы используем реактивное значение `running` для контроля за прерыванием выходного значения, а функция `invalidateLater()` поможет нам инвалидировать наблюдателя каждые 250 мс при запущенном процессе:

```

ui <- fluidPage(
  actionButton("start", "start"),
  actionButton("stop", "stop"),
  textOutput("n")
)

server <- function(input, output, session) {
  r <- reactiveValues(running = FALSE, n = 0)

  observeEvent(input$start, {
    r$running <- TRUE
  })
  observeEvent(input$stop, {
    r$running <- FALSE
  })
}

```

```

  })

  observe({
    if (r$running) {
      r$n <- isolate(r$n) + 1
      invalidateLater(250)
    }
  })
  output$n <- renderText(r$n)
}

```

Заметим, что в данном примере мы не можем просто использовать функцию `observeEvent()`, поскольку должны выполнять разные действия в зависимости от значения `running()`: `TRUE` или `FALSE`. Вместо этого нам пришлось применять функцию `isolate()`. Если бы мы этого не сделали, наблюдатель получил бы реактивную зависимость от значения `n`, которое он обновляет, что привело бы к появлению бесконечного цикла.

Надеюсь, приведенные здесь фрагменты кода создали у вас первое впечатление о программировании с применением подобных функций. Это пример императивного подхода: когда происходит это, делай это, когда происходит то, делай то. В небольших масштабах понять это довольно просто, но все становится сложнее, когда в приложении начинают взаимодействовать большие структуры. Так что старайтесь использовать показанные здесь приемы как можно реже и следите за изоляцией, чтобы как можно меньше наблюдателей могли менять реактивные значения.

Упражнения

Упражнение 1

Напишите серверную функцию, которая будет выводить на график гистограмму по сотне случайных чисел с нормальным распределением при нажатии на кнопку **Normal** и с равномерным распределением – по кнопке **Uniform**:

```

ui <- fluidPage(
  actionButton("rnorm", "Normal"),
  actionButton("runif", "Uniform"),
  plotOutput("plot")
)

```

Упражнение 2

Измените предыдущий код для работы с приведенным ниже интерфейсом пользователя:

```

ui <- fluidPage(
  selectInput("type", "type", c("Normal", "Uniform")),
  actionButton("go", "go"),
  plotOutput("plot")
)

```

Упражнение 3

Перепишите код из предыдущего упражнения таким образом, чтобы в нем не было функций `observe()/observeEvent()`, а использовалась только функция `reactive()`. Почему это можно сделать для второго интерфейса и нельзя для первого?

АНТИШАБЛОНЫ

Привыкнув к такому шаблону, очень легко приобрести дурные привычки:

```
server <- function(input, output, session) {
  r <- reactiveValues(df = cars)
  observe({
    r$df <- head(cars, input$nrows)
  })

  output$plot <- renderPlot(plot(r$df))
  output$table <- renderTable(r$df)
}
```

В представленном простом примере этот код не выполняет много дополнительной работы в сравнении с альтернативным кодом, использующим функцию `reactive()`:

```
server <- function(input, output, session) {
  df <- reactive(head(cars, input$nrows))

  output$plot <- renderPlot(plot(df()))
  output$table <- renderTable(df())
}
```

Но и здесь есть пара недостатков:

- если таблица или график находятся на невидимых в данный момент вкладках, наблюдатель все равно будет их перерисовывать;
- если функция `head()` инициирует ошибку, функция `observe()` просто завершит приложение, тогда как `reactive()` распространит ошибку таким образом, что отображаемый реактивный элемент выдаст ее и не будет передавать дальше.

Все становится еще хуже с ростом сложности приложения. В результате очень легко вернуться к принципам событийно-ориентированного программирования, о которых мы говорили в главе 13. И вот вы уже выполняете кучу лишней работы по отслеживанию возникающих событий, вместо того чтобы позволить Shiny делать это автоматически.

Очень показательно будет сравнить два реактивных графика. На рис. 16.3 показан график из первого примера, и он немного сбивает с толку, поскольку элемент ввода `nrows` здесь никак не связан с `df()`. На рис. 16.4 эта связь уже хорошо заметна. Максимально простой и понятный реактивный график – это

хорошо и для человека, и для Shiny. Человеку его легко понять, а Shiny – оптимизировать.

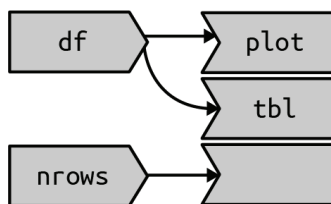


Рис. 16.3 ❖ Использование реактивных значений и наблюдателей делает график разрозненным

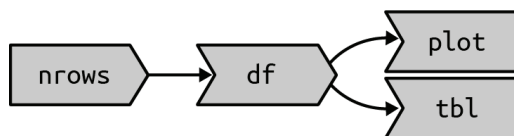


Рис. 16.4 ❖ Реактивный элемент позволил связать все компоненты графика воедино

ЗАКЛЮЧЕНИЕ

В предыдущих четырех главах вы много узнали о модели реактивного программирования, принятой в Shiny. Вы должны были усвоить главный принцип реактивного программирования, заключающийся в выполнении минимально необходимой работы для достижения цели, а также понять, как работает реактивный график. Кроме того, я постарался рассказать о том, как внутренне устроены основные строительные блоки реактивного программирования и как можно использовать их для преодоления присущих реактивному графику ограничений.

В оставшейся части книги мы посмотрим на фреймворк Shiny через призму *инженерии программного обеспечения* (software engineering). В следующих семи главах вы узнаете, как сделать ваши приложения Shiny надежными, эффективными и безопасными вне зависимости от степени их сложности.

Часть IV

ЭФФЕКТИВНЫЕ ПРИЕМЫ

Делая первые шаги в мире Shiny, вы будете довольно много времени тратить на разработку даже самых простых приложений, поскольку вам необходимо будет освоить азы использования этого мощного фреймворка. Со временем вы почувствуете уверенность в работе с интерфейсом приложений и парадигмой реактивного программирования и сможете создавать гораздо более сложные и масштабные продукты. И именно на этапе перехода к написанию сложных приложений вы столкнетесь с новым вызовом: как сохранить надежность, стабильность и организованность растущего объема кода. Проблемы, с которыми вы можете столкнуться на этом пути:

- «в этом огромном файле просто невозможно найти нужный мне фрагмент кода»;
- «я не работал с этим кодом уже полгода – боюсь, я что-нибудь сломаю, если попытаюсь внести какие-то изменения»;
- «мы с коллегой объединили усилия в работе над приложением и постоянно наступаем друг другу на пятки»;
- «приложение прекрасно работает на моем компьютере, но не работает на компьютере коллеги или заказчика».

В этой части книги, названной «Эффективные приемы», вы познакомитесь с ключевыми концепциями и инструментами в области инженерии программного обеспечения, которые помогут вам в преодолении сопутствующих сложностей:

- в главе 17 я вкратце познакомлю вас с идеями, лежащими в основе инженерии программного обеспечения;
- в главе 18 мы познакомимся со способами переноса кода между приложениями и поговорим о том, зачем это может понадобиться;
- в главе 19 мы затронем тему модульной системы Shiny, позволяющей выделять код, включающий интерфейс пользователя и серверную функцию, в изолированные и доступные для повторного использования компоненты;
- глава 20 будет посвящена принципам преобразования приложений в пакеты R и связанным с ними преимуществам применительно к масштабным проектам;

- в главе 21 мы поговорим о том, как автоматизировать ваши существующие тесты приложения, чтобы они могли запускаться всякий раз при изменении проекта;
- из главы 22 вы узнаете о шаблонах, которых следует избегать с целью поддержания безопасности ваших приложений;
- в главе 23 мы рассмотрим способы определения узких мест в приложениях и расскажем, как обеспечить эффективную скорость работы приложений даже при наличии сотен пользователей.

Разумеется, вы не сможете узнать все об инженерии программного обеспечения, прочитав одну часть книги, так что я дам вам дополнительные источники информации для самообразования.

Глава 17

Общие принципы

ВВЕДЕНИЕ

В данной главе мы поговорим о наиболее важных навыках инженерии программного обеспечения, которые вам понадобятся при написании приложений Shiny. Они связаны с организацией кода, тестированием, управлением зависимостями, контролем версий исходного кода, непрерывной интеграцией и анализом кода. Все эти навыки не относятся напрямую к приложениям Shiny, но вам они очень пригодятся при написании сложных приложений, если хотите, чтобы по мере увеличения сложности продукта поддерживать его было не труднее, а легче.

Оттачивание навыков инженерии ПО – это путешествие длиною в жизнь. Поначалу вы столкнетесь с серьезными трудностями на пути освоения этой темы, но помните, что все через это проходили, и если вы проявите достаточное упорство, то сможете одолеть эту науку. Большинство людей, изучающих новую для себя технику, проходят один и тот же путь от «Я ничего в этом не понимаю и вынужден каждый раз лезть за шпаргалкой при необходимости» через «Кое в чем я уже разобрался, но по-прежнему часто открываю документацию» и к «Я хорошо понимаю рабочий процесс и свободно владею всеми инструментами». Но чтобы добраться от начала до конца, потребуется немало времени.

Я бы рекомендовал выделять какое-то время в неделю для оттачивания навыков инженерии ПО. В это время старайтесь не уделять внимания поведению и внешнему виду вашего приложения, а вместо этого сосредоточьтесь на том, чтобы приложение стало легче для понимания и разработки. Это позволит более безболезненно вносить в него изменения в будущем. Кроме того, с развитием навыков инженерии программного обеспечения качество ваших приложений будет неукоснительно расти.

Выражаю огромную благодарность моему коллеге Джеффу Аллену (Jeff Allen) за помощь в написании этой главы.

ОРГАНИЗАЦИЯ КОДА

Любой дурак способен написать код, который поймет компьютер. А код, написанный толковым программистом, поймет даже человек.

*Мартин Фаулер (Martin Fowler),
Refactoring: Improving the Design of Existing Code*

Одним из наиболее очевидных способов повысить качество приложения является улучшение восприятия и читаемости его кода. Даже лучшим в мире программистам будет очень непросто поддерживать код, который они не понимают. Так что это может служить отличной отправной точкой.

Быть хорошим программистом – значит проявлять эмпатию по отношению к людям, которым в будущем придется разбираться в написанном коде (даже если это будете вы сами). Как и другие формы эмпатии, эта требует значительной практики, и чтобы освоить этот навык в совершенстве, может потребоваться немало времени. С его течением вы обнаружите, что те или иные приемы позволяют вам улучшить читаемость вашего кода. Здесь не может быть каких-то универсальных правил, но есть базовые вопросы:

- являются ли имена переменных и функций в вашем коде интуитивно понятными? Если нет, то какие имена способны наиболее полно передать замысел вашего приложения?
- все ли сложные фрагменты кода снабжены комментариями?
- помещается ли текст функции на экран и может ли быть распечатан на одном листе? Если нет, возможно, существуют варианты разбить ее на несколько функций?
- приходилось ли вам многократно копировать и вставлять блоки кода при написании приложения? Если да, скорее всего, можно выделить повторяющиеся фрагменты в переменные или функции, чтобы использовать их повторно;
- сочетаются ли вместе все части вашего приложения или можно управлять различными компонентами отдельно?

Как мы уже говорили, здесь нет одного универсального решения, и часто предлагаемые варианты связаны с субъективным выбором. Но два фундаментальных инструмента у вас в наличии есть всегда:

- функции: в главе 18 мы рассмотрим варианты уменьшения количества дублирования блоков кода в интерфейсе пользователя, что позволит сделать серверную функцию более понятной и легкой для тестирования, а ваш код в целом – более гибким и хорошо организованным;
- модули Shiny: в главе 19 мы посмотрим, как можно писать изолированный код, координирующий работу интерфейсной и серверной частей приложения при помощи модулей Shiny. Модули позволяют полностью разделить задачи, чтобы, например, отдельные страницы вашего приложения могли функционировать независимо, а повторяющиеся компоненты не нуждались в копировании и вставке.

ТЕСТИРОВАНИЕ

Разработка *плана тестирования* (test plan) приложения – очень важный процесс, позволяющий обеспечить надежность и стабильность его работы. Без наличия проверенного плана тестирования любое изменение способно поставить под угрозу все приложение в целом. Если ваше приложение недостаточно большое и вы способны все удерживать в голове, вам может показаться, что наличие дополнительного плана тестирования будет излишним. И действительно, тестирование предельно простых приложений порой хлопот доставляет больше, чем приносит пользы. В то же время отсутствие плана может сыграть с вами злую шутку, когда кто-нибудь захочет внести изменения в приложение или вы сами по прошествии долгого времени задумаете продолжить его разработку.

План тестирования может быть рассчитан полностью на ручной процесс. Можно начать с простого текстового файла со скриптом, который будет запускаться с целью проверки работоспособности приложения. Но этот файл будет непременно увеличиваться в размерах с ростом сложности приложения. И вы неизбежно будете тратить все больше времени на ручное тестирование приложения либо начнете пропускать определенные скрипты.

В таких случаях неплохо бывает задуматься об автоматизации процесса тестирования. Автоматизация требует дополнительного времени для настройки, но в перспективе вы получите выгоду в виде возможности запускать тесты более часто. По этой причине для Shiny было разработано большое количество видов автоматизированных тестов, о чем мы подробнее поговорим в главе 21. Сейчас лишь скажем, что вы можете проводить следующие виды тестов:

- *модульное тестирование* (unit test) – подтверждает корректность работы отдельных функций;
- *интеграционное тестирование* (integration test) – используется для проверки взаимодействий между реактивными элементами;
- *функциональное тестирование* (functional test) – выполняется для проверки полноценной работы приложения в браузере;
- *тестирование под нагрузкой* (load test) – проверка того, что приложение сможет справиться с трафиком в боевом режиме.

Прелесть автоматизированного тестирования состоит в том, что, один раз все написав и настроив, вам не придется вручную проверять работу определенных участков приложения. Кроме того, вы можете проводить *непрерывную интеграцию* (continuous integration), о которой мы поговорим позже, с целью запуска отдельных тестов каждый раз после внесения изменений в приложение перед его публикацией.

УПРАВЛЕНИЕ ЗАВИСИМОСТЯМИ

Если вы когда-либо пытались произвести анализ кода на R, написанного кем-то другим, или хотя бы проанализировать приложение Shiny, которое

сами разработали достаточно давно, вы, вероятно, знаете, что риск запутаться в зависимостях может быть очень велик. К зависимостям в приложении можно отнести все, что не касается непосредственно исходного кода и что необходимо для его корректной работы. Это могут быть и файлы на жестком диске, и внешние базы данных или API, и пакеты R, используемые приложением.

Для любого анализа, который вы планируете воспроизводить в будущем, можно использовать пакет *renv* (<https://rstudio.github.io/renv>), позволяющий создавать *воспроизводимые окружения* (reproducible **e**nvironment). С помощью этого пакета вы можете сохранить необходимые для вашего приложения версии пакетов, чтобы при его запуске на другом компьютере были использованы именно они. Это важно не только для первого запуска приложений, но и для того, чтобы обезопасить их от изменения версий пакетов в будущем.

Еще одним средством управления зависимостями является пакет *config* (<https://github.com/rstudio/config>). Этот пакет не управляет зависимостями как таковыми, а, скорее, обеспечивает удобное место для отслеживания и управления зависимостями помимо пакетов R. Например, вы можете указать путь к файлу CSV, который используется в вашем приложении, или ссылку на необходимый API. Перечисление этих ресурсов в конфигурационном файле позволяет хранить все зависимости в едином месте. Более того, этот пакет дает возможность создавать разные конфигурации для разных окружений. К примеру, если ваше приложение предназначено для работы с объемной базой данных, вы можете настроить окружения следующим образом:

- в рабочем окружении ваше приложение подключается к реальной базе данных;
- в тестовом окружении вы можете использовать тренировочную базу, что позволит вам проверить выполнение подключения и прочие нюансы взаимодействия с источником данных без риска нарушить целостность реальной рабочей базы данных;
- в окружении для разработки вы можете сконфигурировать приложение таким образом, чтобы оно использовало данные из небольшого файла CSV с ограниченной информацией для быстрой работы.

Наконец, остерегайтесь делать какие-либо предположения о локальной файловой системе пользователя. К примеру, если ваше приложение ссылается на данные, расположенные по пути *C:\data\cars.csv* или *~/my-projects/genes.rds*, вы должны понимать, что эти файлы вряд ли найдутся на другом компьютере. Вместо этого лучше указывать пути относительно рабочей директории приложения (допустим, *data/cars.csv* или *genes.rds*) или воспользоваться пакетом *config*, чтобы сделать внешний путь явным и настраиваемым.

КОНТРОЛЬ ВЕРСИЙ ИСХОДНОГО КОДА

Любой, кто программирует более или менее продолжительное время, конечно, сталкивался с ситуацией, когда код оказывается безнадежно сломан

и возникает желание откатиться к его стабильной рабочей версии. Следить за версиями самостоятельно и восстанавливать их вручную – задача не из легких. К счастью, вы можете использовать в работе *систему контроля версий* (version-control system) исходного кода, помогающую отслеживать малейшие изменения в проекте, управлять версиями и координировать совместную работу программистов.

В сообществе R наиболее популярной системой контроля версий считается *Git*. Обычно она используется совместно с GitHub – хранилищем для совместного использования репозиторий. Конечно, вам потребуется какое-то время, чтобы освоить работу в Git и GitHub, но любой профессионал вам подтвердит, что эти усилия окупятся в будущем. Если вы никогда прежде не работали с Git, я настоятельно рекомендую начать ознакомление с этой системой с книги *Happy Git and GitHub for the useR* от Дженни Брайан (Jenny Bryan) по адресу <https://happygitwithr.com>.

НЕПРЕРЫВНАЯ ИНТЕГРАЦИЯ/РАЗВЕРТЫВАНИЕ

Установив систему контроля версий исходного кода и настроив необходимый набор автоматизированных тестов, вы можете в полной мере воспользоваться всеми преимуществами непрерывной интеграции. *Непрерывная интеграция* (continuous integration – CI) позволяет гарантировать, что ни одно изменение не навредит целостности и работоспособности приложения. Вы можете использовать эту систему как в ретроспективном режиме (с сообщениями о том, могут ли навредить приложению сделанные вами изменения), так и в перспективном (с предварительными оповещениями о возможных проблемах, связанных с действиями, которые вы только собираетесь предпринять).

Существует великое множество служб, способных подключаться к репозиторию Git и выполнять определенные тесты, когда вы выполняете или предлагаете к выполнению какое-либо изменение. В зависимости от того, где располагается ваш код, вы можете использовать системы *GitHub actions* (<https://github.com/features/actions>), *Travis CI* (<https://travis-ci.org>), *Azure Pipelines* (<https://azure.microsoft.com/en-us/services/devops/pipelines>), *AppVeyor* (<https://www.appveyor.com>), *Jenkins* (<https://www.jenkins.io>), *GitLab CI/CD* (<https://about.gitlab.com/stages-devops-lifecycle/continuous-integration>) и многие другие.

На рис. 17.1 показано, как может выглядеть процесс подключения системы непрерывной интеграции к GitHub на предмет обработки запросов на включение внесенных изменений в проект. Как видите, все проверки оказались пройдены успешно, что говорит о положительном завершении всех автоматизированных тестов. При возникновении ошибки вы будете уведомлены об этом еще до принятия предложенных вами изменений. Использование системы непрерывной интеграции позволяет не только уберечь опытных разработчиков от случайных ошибок, но и облегчает вход в проект новым участникам.

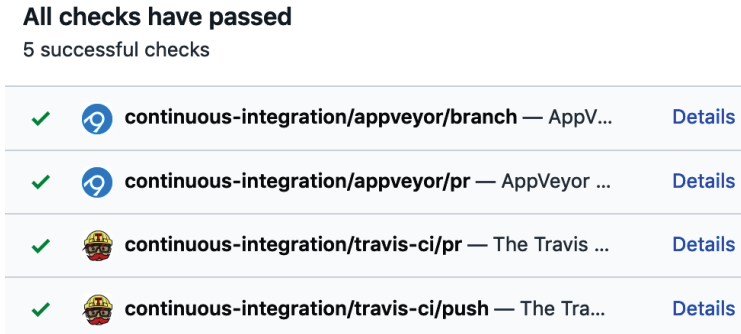


Рис. 17.1 ❖ Пример запущенной системы непрерывной интеграции показывает успешное прохождение нескольких тестов

Анализ кода

Многие компании, занимающиеся производством программного обеспечения, успешно пользуются услугами сторонних разработчиков по анализу кода перед его отправкой в репозиторий. Такой процесс внешнего анализа полезен сразу по нескольким причинам:

- он позволяет отловить ошибки еще до выпуска приложения, что дает возможность исправить их в спокойном режиме;
- он преследует обучающие цели: разработчики обычно узнают много нового, как анализируя чужой код, так и когда кто-то разбирает их фрагменты приложений;
- он способствует процессу взаимного обучения и распространению ценных знаний внутри команды разработчиков, чтобы не возникало ситуаций, когда лишь один программист в полной мере понимает, что делает приложение;
- взаимное общение в процессе выполнения анализа кода чаще всего позволяет улучшить его читаемость.

Обычно анализ кода подразумевает участие сторонних разработчиков, но будет полезно, и если вы все будете делать сами. Опытные разработчики подтвердят, что в процессе анализа собственного кода очень часто можно выявить какие-то мелкие недостатки, особенно если с момента его написания прошло как минимум несколько часов.

Вот лишь некоторые вопросы, которыми стоит задаваться в процессе анализа кода:

- названы ли новые функции лаконично и выразительно?
- присутствуют ли в коде фрагменты, способные запутать разработчика?
- какие участки кода могут измениться в будущем, и будет ли смысл построить для них автоматизированное тестирование?
- соответствует ли стиль фрагмента кода стилю написания всего приложения в целом (а еще лучше – утвержденному стилю написания кода в вашей рабочей группе)?

Если в вашей компании высокоразвита культура инженерии программного обеспечения, даже анализ сложного кода из области науки о данных может показаться вам весьма простой задачей, поскольку в вашем распоряжении будут все необходимые инструменты и накопленный опыт. В противном случае этот процесс может потребовать от вас дополнительной скрупулезной работы.

По теме анализа кода я бы порекомендовал почитать два источника:

- руководство по анализу кода от *thoughtbot* по адресу <https://github.com/thoughtbot/guides/tree/main/code-review>;
- инструкцию для разработчиков по анализу кода по адресу <https://google.github.io/eng-practices/review>.

ЗАКЛЮЧЕНИЕ

Теперь, когда вы немного разобрались в области инженерии программного обеспечения, мы перейдем к следующей главе, в которой подробно поговорим о функциях, а точнее об их написании, тестировании, безопасности и эффективности применительно к приложениям Shiny. Если вы собираетесь читать книгу до конца, то главу 18 точно не следует обходить вниманием...

Глава 18

Функции

С ростом сложности приложения бывает все труднее удержать все составляющие его части в голове, а значит, и труднее понять, как оно работает. Как следствие усложняется процедура добавления новых элементов в код, и все более запутанным становится процесс отладки. Если в такой ситуации не предпринять какие-то намеренные действия, темп разработки приложения рискует значительно снизиться, и работать над ним станет все менее интересно.

В данной главе мы поговорим о том, как в подобных условиях может помочь написание функций. При этом подход к клиентской и серверной частям приложения будет различным:

- в интерфейсе пользователя обычно присутствуют практически идентичные блоки кода, за небольшими исключениями. Выделение повторяющегося кода в функции позволит снизить дублирование кода и вместе с тем облегчит обновление сразу нескольких элементов в одном месте. Кроме того, в этой области можно применить уже знакомые вам принципы функционального программирования для одновременного создания нескольких элементов управления;
- в серверной функции сложнее всего процессу отладки поддаются сложные реактивы, поскольку для этого приложение должно быть запущено. Вынос реактивов в отдельную функцию, даже если она будет вызываться лишь в одном месте, позволит значительно упростить выполнение отладки, ведь в этом случае вы сможете экспериментировать с вычислениями независимо от реактивных элементов.

Функции играют еще одну важную роль в Shiny: они позволяют разбить объемный код приложения на отдельные файлы. Конечно, вы можете все держать в файле *app.R* гигантского размера, но гораздо легче будет управлять кодом, если он будет содержаться в нескольких файлах меньшего объема.

Я полагаю, что вы уже хорошо знакомы с основными принципами работы функций¹. Здесь же мы постараемся улучшить уже имеющиеся у вас навыки работы с функциями и продемонстрируем ситуации, в которых их использование может помочь сделать приложение более понятным и прозрачным. После того как вы освоите все написанное в этой главе, можно будет приступать к изучению приемов написания кода, требующего взаимодействия

¹ Если это не так, вы всегда можете обратиться к соответствующей главе книги *R for Data Science* по адресу <https://r4ds.had.co.nz/functions.html>.

между клиентской и серверной частями приложения. Для этого вам понадобятся модули, о которых мы будем говорить в главе 19. Как обычно, начнем с загрузки нужного нам пакета:

```
library(shiny)
```

ОРГАНИЗАЦИЯ ФАЙЛОВ

Перед тем как приступить к разговору о том, как следует эффективно использовать функции в приложениях, стоит упомянуть, что функции могут находиться за пределами файла *app.R*. А именно вы можете поместить их в два разных места в зависимости от их объема:

- большие функции вместе со вспомогательными функциями, необходимыми для их работы, я рекомендую размещать в отдельных файлах с шаблонными именами *R/{function-name}.R*;
- простые и небольшие по размеру функции можно собрать в одном месте. Я часто использую для них файл *R/utils.R*, но если они преимущественно используются в интерфейсе пользователя, можете поместить их в файл *R/ui.R*.

Если вы ранее уже создавали пакеты R, то наверняка заметили, что в Shiny используется такое же соглашение об именовании файлов, хранящих функции. И если вы разрабатываете большое и сложное приложение, особенно если над ним трудятся несколько человек, есть смысл в создании полноценного пакета. Для получения дополнительных знаний в этой области я бы рекомендовал начать с чтения книги *Engineering Production-Grade Shiny Apps*, расположенной по адресу <https://engineering-shiny.org>, и использования вспомогательного пакета *golem* (<https://thinkr-open.github.io/golem>). Подробнее о пакетах мы поговорим при обсуждении процедуры тестирования.

ФУНКЦИИ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ

Функции являются очень мощным средством для избавления от дублирующихся фрагментов в коде пользовательского интерфейса. Начнем сразу с конкретного примера повторяющегося кода. Представьте, что у вас в интерфейсе есть несколько ползунков с диапазоном от нуля до единицы, начальным значением 0.5 и шагом 0.1. Вы могли бы несколько раз скопировать и вставить текст для создания всех элементов управления следующим образом:

```
ui <- fluidRow(
  sliderInput("alpha", "alpha", min = 0, max = 1, value = 0.5, step = 0.1),
  sliderInput("beta", "beta", min = 0, max = 1, value = 0.5, step = 0.1),
  sliderInput("gamma", "gamma", min = 0, max = 1, value = 0.5, step = 0.1),
  sliderInput("delta", "delta", min = 0, max = 1, value = 0.5, step = 0.1)
)
```

Однако здесь несложно распознать дублирующий шаблон, создать функцию и вызвать ее несколько раз, существенно сократив код в целом:

```
sliderInput01 <- function(id) {
  sliderInput(id, label = id, min = 0, max = 1, value = 0.5, step = 0.1)
}

ui <- fluidRow(
  sliderInput01("alpha"),
  sliderInput01("beta"),
  sliderInput01("gamma"),
  sliderInput01("delta")
)
```

В данном случае функция позволила нам:

- сделать код легко читаемым. Мы дали функции понятное имя, с которым точно не запутаемся, анализируя код через какое-то время;
- облегчить себе жизнь в будущем, если мы захотим изменить поведение элемента, поскольку сделать это можно будет всего в одном месте кода. Допустим, если нам вздумается уменьшить шаг ползунков с 0.1 до 0.01, нам придется только в одном месте вставить инструкцию `step = 0.01`, а не в четырех.

Другое применение

Функции могут быть полезны и в других местах приложений. Вот лишь несколько вариантов их применения:

- если вы используете элементы ввода `dateInput()`, настроенные под вашу страну, соберите их все в одном блоке кода, чтобы можно было передавать нужные аргументы. Например, так вы могли бы создать элемент ввода даты, позволяющий американцам выбирать день недели:

```
usWeekDateInput <- function(inputId, ...) {
  dateInput(inputId, ..., format = "dd M, yy", daysofweekdisabled = c(0, 6))
}
```

Обратите внимание на многоточие в качестве аргумента функции `dateInput()`, позволяющее вам передавать любые другие параметры в эту функцию;

- вы также могли бы создать кнопки-переключатели, облегчающие выбор значков:

```
iconRadioButtons <- function(inputId, label, choices, selected = NULL) {
  names <- lapply(choices, icon)
  values <- if (is.null(names(choices))) names(choices) else choices
  radioButtons(inputId,
    label = label,
    choiceNames = names, choiceValues = values, selected = selected
  )
}
```

- кроме того, вам могут понадобиться специфические выпадающие списки, которые необходимо использовать в разных местах приложения:

```
stateSelectInput <- function(inputId, ...) {
  selectInput(inputId, ..., choices = state.name)
}
```

Если вы разрабатываете разные приложения Shiny в рамках организации, то можете повысить их универсальность, собрав подобные функции в общий пакет.

Функциональное программирование

Возвращаясь к нашему первому примеру, стоит заметить, что вы могли бы еще больше сократить код, воспользовавшись прелестями функционального программирования, как показано ниже:

```
library(purrr)

vars <- c("alpha", "beta", "gamma", "delta")
sliders <- map(vars, sliderInput01)
ui <- fluidRow(sliders)
```

Здесь заложены сразу две важные идеи:

- функция `map()` вызывает функцию `sliderInput01()` по разу для каждой строки, сохраненной в векторе `vars`. В результате мы получаем список ползунков;
- передача списка функции `fluidRow()` (или любому другому контейнеру HTML) приводит к его автоматической распаковке таким образом, что каждый элемент списка становится вложенным элементом контейнера.

Если вы хотите больше узнать о функции `map()` или ее эквиваленте из базового пакета R `lapply()`, то можете обратиться к главе *Functionals* книги *Advanced R* по адресу <https://adv-r.hadley.nz/functionals.html>.

Интерфейс пользователя в виде структуры данных

Можно обобщить эту идею для случая, когда элемент управления включает в себя больше одного элемента ввода с разными характеристиками. Для начала создадим встроенный датафрейм, определяющий параметры каждого элемента при помощи функции `tibble::tribble()`. В результате мы превращаем структуру интерфейса пользователя в явным образом объявленную структуру данных:

```
vars <- tibble::tribble(
  ~ id, ~ min, ~ max,
  "alpha", 0, 1,
  "beta", 0, 10,
```

```

    "gamma",    -1,    1,
    "delta",     0,    1,
  )

```

После этого создадим функцию, в которой имена аргументов совпадают с именами столбцов:

```

mySliderInput <- function(id, label = id, min = 0, max = 1) {
  sliderInput(id, label, min = min, max = max, value = 0.5, step = 0.1)
}

```

И наконец, вызовем функцию `purrr::map()` для применения функции `mySliderInput()` по одному разу для каждой строки датафрейма `vars`:

```
sliders <- map(vars, mySliderInput)
```

Не беспокойтесь, если вы не до конца поняли приведенный выше код – вы можете по старинке использовать технику копирования и вставки. Но в будущем я бы очень рекомендовал вам освоить приемы функционального программирования, которые, как видите, позволяют делать громоздкие и неуклюжие конструкции очень выразительными и лаконичными. Вы можете также обратиться к главе 10, где мы использовали методы функционального программирования для создания динамического интерфейса в ответ на действия пользователя.

СЕРВЕРНЫЕ ФУНКЦИИ

Всегда, когда у вас в приложении появляется длинное реактивное выражение (скажем, объемом больше десяти строк), вы должны задуматься о выделении его в функцию, не использующую реактивность. У этого подхода есть два серьезных преимущества:

- код гораздо легче отлаживать, если все реактивы используются внутри функции `server()`, а сложные вычисления собраны в отдельных функциях;
- при взгляде на реактивное выражение или элемент вывода невозможно быстро определить, от каких значений они зависят, если не вчитываться в код. В то же время в определении функции уже присутствуют все указания на входные параметры.

Основной целью использования функций в интерфейсе пользователя является уменьшение количества дублирующегося кода, тогда как в серверной части применение функций главным образом способствует лучшей изоляции кода и тестированию.

Чтение загруженных данных

Давайте вспомним один пример из главы 9, содержащий довольно объемное реактивное выражение:

```

server <- function(input, output, session) {
  data <- reactive({
    req(input$file)

    ext <- tools::file_ext(input$file$name)
    switch(ext,
      csv = vroom::vroom(input$file$datapath, delim = ","),
      tsv = vroom::vroom(input$file$datapath, delim = "\t"),
      validate("Invalid file; Please upload a .csv or .tsv file")
    )
  })

  output$head <- renderTable({
    head(data(), input$n)
  })
}

```

Если бы речь шла о реальном приложении, я скорее предпочел бы выделить операцию чтения загруженных файлов в отдельную функцию:

```

load_file <- function(name, path) {
  ext <- tools::file_ext(name)
  switch(ext,
    csv = vroom::vroom(path, delim = ","),
    tsv = vroom::vroom(path, delim = "\t"),
    validate("Invalid file; Please upload a .csv or .tsv file")
  )
}

```

При создании таких вспомогательных функций избегайте того, чтобы они принимали на вход реактивы или возвращали элементы вывода. Вместо этого в качестве аргументов функции следует передавать значения и предполагать, что вызывающая функция сама преобразует выходное значение в реактив при необходимости. Нет, это нельзя назвать жестким правилом. Иногда можно допустить, чтобы функции принимали на вход или возвращали реактивы. Однако в общем случае я бы рекомендовал, по возможности, разделять реактивную и нереактивную части приложения. Заметьте, что в данном случае я по-прежнему использовал функцию `validate()`, поскольку за пределами Shiny она работает приблизительно как `stop()`. В то же время функцию `req()` я оставил в функции `server()`, так как код с анализом файла не должен заботиться о том, когда он запускается.

В результате мы получили полностью независимую функцию, которая может жить в своем собственном файле (скажем, с именем *R/load_file.R*), не загромождая функцию `server()`. Как следствие функция `server()` может сосредоточиться на реактивной картине в целом, не вдаваясь в подробности реализации отдельных компонентов:

```

server <- function(input, output, session) {
  data <- reactive({
    req(input$file)
    load_file(input$file$name, input$file$datapath)
  })
}

```



```

output$head <- renderTable({
  head(data(), input$n)
})
}

```

Еще одно преимущество такого подхода состоит в том, что вы получаете возможность запускать функцию `load_file()` прямо из консоли, за пределами Shiny. Это позволит облегчить задачу тестирования, о чем мы будем подробно говорить в главе 21.

Внутренние функции

В большинстве случаев вы должны стремиться делать так, чтобы ваши функции были полностью автономными, работали независимо от функции `server()` и могли быть помещены в отдельный файл. Однако в случаях, когда функции необходимо взаимодействовать с параметрами `input`, `output` или `session`, есть смысл оставить ее внутри функции `server()`:

```

server <- function(input, output, session) {
  switch_page <- function(i) {
    updateTabsetPanel(input = "wizard", selected = paste0("page_", i))
  }

  observeEvent(input$page_12, switch_page(2))
  observeEvent(input$page_21, switch_page(1))
  observeEvent(input$page_23, switch_page(3))
  observeEvent(input$page_32, switch_page(2))
}

```

Это не упростит процесс отладки и тестирования, зато позволит минимизировать количество дублирующегося кода.

Конечно, мы могли бы передать нашей функции переменную `session` в качестве параметра. Но это было бы весьма странно – функция и так привязана к нашему приложению, поскольку оказывает влияние на элемент с именем `wizard` с конкретным набором вкладок.

ЗАКЛЮЧЕНИЕ

С ростом сложности приложения вы можете существенно облегчить себе жизнь, выделяя из общего кода функции, не содержащие реактивов. Именно функции могут помочь вам отделить реактивный код от нереактивного и распределить блоки программы по файлам. Зачастую это позволяет увидеть общую картину приложения. Кроме того, выделяя блоки кода со сложной логикой в отдельные файлы с использованием базового R, вы открываете себе новые возможности для экспериментов, циклических проверок и тестирования в целом. Поначалу процесс разбиения кода на функции может показаться вам сложным и утомительным, но со временем вы будете делать

это все быстрее, и вскоре эта техника станет одним из важнейших инструментов в вашем арсенале.

У функций, которые мы приводили в данной главе, есть один общий недостаток, заключающийся в том, что они способны генерировать либо клиентские, либо серверные компоненты, но никак не одновременно. В следующей главе мы обратимся к теме создания модулей Shiny, позволяющих координировать интерфейсный и серверный коды в единый объект.

Глава 19

Модули Shiny

В предыдущей главе мы использовали функции для декомпозиции приложения или сознательного разделения его на части. Функции прекрасно работают в случае, если код выполняется исключительно на сервере или в интерфейсе. Но если код распространяется на обе стороны приложения, т. е. серверная часть так или иначе зависит от клиентских структур в интерфейсе пользователя, одними функциями не обойтись. Здесь вам пригодится новая техника, именуемая модульной структурой.

В простейшем виде *модуль* (module) представляет собой пару функций: для пользовательского интерфейса и сервера. Магия модулей проявляется в том, что эти функции спроектированы особым образом, с созданием пространства имен. До этого момента все имена (id) наших элементов управления были глобальными, а значит, все части серверной функции могли видеть все элементы интерфейса пользователя. Модули дают возможность создавать элементы управления, видимые только внутри него. Это называется *пространством имен* (namespace), поскольку в этом случае буквально создается *пространство* (space) с *именами* (name), изолированное от остального приложения.

Модули Shiny обладают двумя серьезными преимуществами. Во-первых, пространства имен позволяют лучше понять, как работает ваше приложение, – вы можете писать, анализировать и тестировать индивидуальные компоненты по отдельности. Кроме того, поскольку модули, по сути, представляют собой функции, их использование открывает вам дополнительные возможности для повторного использования кода: иначе говоря, все, что вы можете делать с функциями, вы можете делать и с модулями. Давайте начнем с привычной загрузки пакета:

```
library(shiny)
```

Предпосылки

Перед тем как погрузиться в детали создания модулей, полезно будет узнать, как они способны поменять «форму» вашего приложения. Я займусь пример Эрика Нанца (Eric Nantz), выступавшего на конференции *rstudio::conf(2019)* с докладом о модулях. Сам Эрик был вынужден использовать модули, поскольку разрабатывал очень большое и сложное приложение,

схема которого показана на рис. 19.1. Мы не знаем всей специфики этого приложения, но вполне можем сделать вывод о его сложности по большому количеству связанных компонентов.

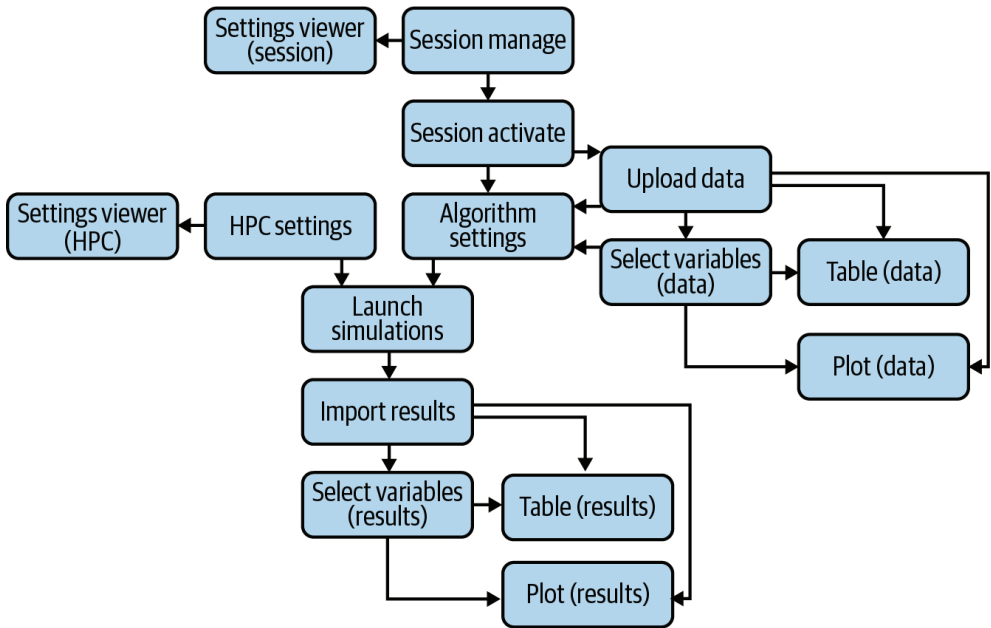


Рис. 19.1 ❖ Грубый набросок схемы работы приложения.

Я постарался максимально просто отразить приложение на диаграмме, но и в таком виде понять его довольно сложно

На рис. 19.2 показана схема работы приложения с использованием модулей. Здесь мы видим следующее:

- приложение разбито на части, каждая из которых обладает своим именем. Именование целых частей приложения упрощает процесс выдачи имен отдельным элементам управления. К примеру, если изначально в приложении были элементы с именами *session manage* и *session activate*, после разбиения его на модули мы можем разместить их в модуле *session* и назвать просто *manage* и *activate*. Так работают пространства имен на практике!
- модуль представляет собой черный ящик с определенными входами и выходами. Другие модули могут коммуницировать с ним только посредством внешнего интерфейса – пробраться внутрь и получить непосредственный доступ к элементам или реактивам модуля у них не получится. Это позволяет существенно упростить структуру приложения в целом;
- модули представляют собой повторно используемые блоки кода, так что мы можем написать отдельные функции для желтых и синих компонентов на рис. 19.2. Это позволяет значительно снизить общий объем кода приложения.

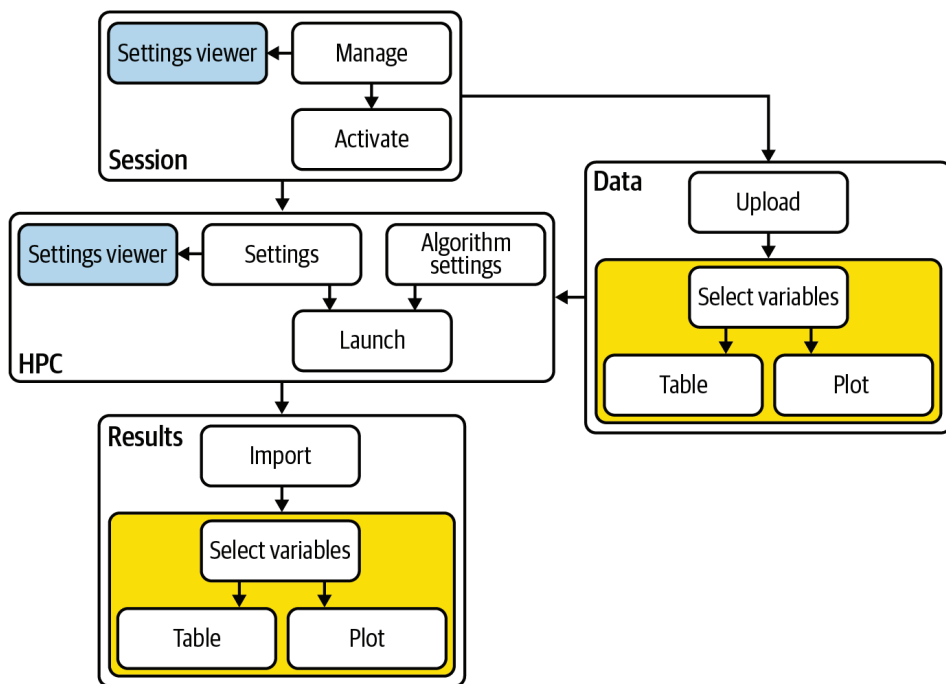


Рис. 19.2 ❖ После перевода приложения на модульную систему его общая структура стала гораздо понятнее, а желтые и синие компоненты могут быть унифицированы при помощи повторного использования кода

ОСНОВЫ МОДУЛЬНОЙ СИСТЕМЫ

Для создания нашего первого модуля мы позаимствуем код простого приложения для построения гистограмм:

```

ui <- fluidPage(
  selectInput("var", "Variable", names(mtcars)),
  numericInput("bins", "bins", 10, min = 1),
  plotOutput("hist")
)

server <- function(input, output, session) {
  data <- reactive(mtcars[[input$var]])
  output$hist <- renderPlot({
    hist(data(), breaks = input$bins, main = input$var)
  }, res = 96)
}

```

Представленное здесь приложение настолько простое, что вовсе не нуждается в применении модульной структуры, но мы используем его для демонстрации базовой механики концепции модулей, после чего обратимся к более сложным примерам.

Модуль внешне очень напоминает приложение. Как и приложение, он состоит из двух частей¹:

- интерфейсной функции, генерирующей определение пользовательского интерфейса модуля;
- серверной функции, запускающей код внутри функции `server`.

Обе функции имеют стандартный вид. Они обе принимают на вход аргумент `id`, используемый для формирования пространства имен модуля. Для образования модуля необходимо перенести код из функций интерфейса и сервера приложения в функции модуля.

Интерфейс модуля

Начнем с *интерфейсной функции модуля* (`module UI`). Здесь вам необходимо выполнить два шага:

- перенести код из вашей функции интерфейса пользователя в функцию с аргументом `id`;
- заключить каждый существующий идентификатор элемента (`id`) в функцию `NS()` таким образом, чтобы, например, `"var"` превратился в `NS(id, "var")`.

В результате мы получим следующую функцию модуля:

```
histogramUI <- function(id) {
  tagList(
    selectInput(NS(id, "var"), "Variable", choices = names(mtcars)),
    numericInput(NS(id, "bins"), "bins", value = 10, min = 1),
    plotOutput(NS(id, "hist"))
  )
}
```

В данном случае мы вернули компоненты интерфейса пользователя в виде `tagList()` – особого типа функции макета, позволяющего объединить вместе все элементы без определения их конкретного расположения. Этим сможет заняться тот, кто будет вызывать функцию `histogramUI()`, – он сам должен будет заключить результат в нужную функцию макета, будь то `column()` или `fluidRow()`.

Серверная логика модуля

Теперь пришло время заняться *серверной функцией модуля* (`module server`). Она будет заключена внутри *другой* функции, которая должна принимать на вход аргумент `id`. Эта внешняя функция будет вызывать функцию `moduleServer()` и передавать ей полученный `id` вместе с функцией, напоминающей обычную серверную функцию Shiny:

¹ В отличие от приложения, интерфейсная и серверная части модуля являются функциями.

```

histogramServer <- function(id) {
  moduleServer(id, function(input, output, session) {
    data <- reactive(mtcars[[input$var]])
    output$hist <- renderPlot({
      hist(data(), breaks = input$bins, main = input$var)
    }, res = 96)
  })
}

```

Здесь необходимо понять важность этой вложенной структуры функций. Подробнее мы будем говорить об этом позже, но если вкратце, то такое деление позволяет различать аргумент модуля от аргументов серверной функции. Не беспокойтесь, если пока вам это кажется слишком сложным, – по сути, это всего лишь шаблон, который вы можете копировать и вставлять для каждого нового модуля.

Обратите внимание, что в функции `moduleServer()` пространство имен учитывается автоматически: внутри `moduleServer(id)` инструкции `input$var` и `input$bins` ссылаются на элементы ввода с именами `NS(id, "var")` и `NS(id, "bins")` соответственно.

Обновленное приложение

Теперь, когда у нас есть интерфейсная и серверная функции, можно объединить их в функцию для создания приложения, которое мы будем использовать для экспериментов и тестов:

```

histogramApp <- function() {
  ui <- fluidPage(
    histogramUI("hist1")
  )
  server <- function(input, output, session) {
    histogramServer("hist1")
  }
  shinyApp(ui, server)
}

```

Заметьте, что в обеих внутренних функциях вы должны использовать одинаковый идентификатор, – в противном случае связь между частями приложения установлена не будет.

Из истории модулей

Модули были представлены в Shiny версии 0.13 (в январе 2016 года) изначально с функцией `callModule()`, а в июне 2020-го (в версии Shiny 1.5.0) появилась функция `moduleServer()`. Если вы изучали модульную архитектуру Shiny в прошлом, вероятно, вы знакомы с функцией `callModule()` и не можете понять, откуда взялась функция `moduleServer()`. Эти две функции полностью идентичны, за исключением того, что первые два аргумента в них поменяны местами. Это небольшое изменение привело к ощутимой смене структуры всего приложения в целом:

```

histogramServerOld <- function(input, output, session) {
  data <- reactive(mtcars[[input$var]])
  output$hist <- renderPlot({
    hist(data(), breaks = input$bins, main = input$var)
  }, res = 96)
}
server <- function(input, output, session) {
  callModule(histogramServerOld, "hist1")
}

```

Для этого простого примера разница кажется несущественной, но в случае с более сложными модулями с аргументами использование функции `moduleServer()` может значительно облегчить понимание приложения.

Пространства имен

Теперь, когда мы написали простое модульное приложение, давайте вернемся на шаг назад и поговорим о пространстве имен. Ключевая идея, лежащая в основе модульной архитектуры приложений, заключается в том, что теперь имя каждого элемента управления, т. е. его `id`, определяется двумя составляющими:

- первую определяет *пользователь* модуля – разработчик, вызывающий функцию `histogramServer()`;
- вторую определяет *автор* модуля – разработчик, написавший функцию `histogramServer()`.

Такая двухуровневая спецификация означает, что автору модуля не нужно беспокоиться о возможных конфликтах с другими элементами управления, созданными пользователем. У вас есть свое собственное пространство имен, которым вы оперируете по своему усмотрению.

Именно пространства имен превращают модули в черные ящики. Снаружи модуля не видны элементы ввода и вывода, а также реактивы внутри него. Давайте для примера рассмотрим следующее приложение. Текстовый элемент вывода `output$out` никогда не обновится, поскольку нет элемента ввода `input$bins` – он виден только внутри модуля `hist1`:

```

ui <- fluidPage(
  histogramUI("hist1"),
  textOutput("out")
)
server <- function(input, output, session) {
  histogramServer("hist1")
  output$out <- renderText(paste0("Bins: ", input$bins))
}

```

Если вы хотите получить доступ к элементу ввода из реактива откуда-то еще, вам необходимо передать соответствующие аргументы в модуль явным образом, о чем мы подробнее будем говорить позже.

Обратите внимание, что в интерфейсной и серверной частях модуля пространства имен выражаются по-разному:

- в интерфейсе пространство имен указывается явно – для создания элемента ввода или вывода вам каждый раз нужно будет вызывать функцию `NS(id, "name")`;
- в серверной функции пространство имен лишь подразумевается. Вам достаточно передать `id` при вызове функции `moduleServer()` в качестве аргумента. Shiny автоматически настроит адресацию элементов ввода и вывода таким образом, чтобы в вашем модуле код `input$name` указывал на элемент ввода с именем `NS(id, "name")`.

Соглашение об именовании

В данном примере я использовал особую схему именования для всех компонентов модуля и очень рекомендую вам придерживаться тех же правил при написании собственных модулей. Наш модуль служит для создания гистограмм, поэтому мы назвали его `histogram`. Далее это имя встречается в разных компонентах модуля:

- файл `R/histogram.R` содержит весь код модуля;
- функция интерфейса модуля названа `histogramUI()`. Если бы она преимущественно использовалась для элементов ввода или вывода, я бы назвал ее `histogramInput()` или `histogramOutput()` соответственно;
- серверная функция модуля названа `histogramServer()`;
- приложение, созданное для проведения экспериментов и тестов, названо `histogramApp()`.

Упражнения

Упражнение 1

Почему хорошей практикой считается хранение модуля в своем файле в директории `R/`? Что нужно сделать, чтобы убедиться, что модуль подгружен в Shiny?

Упражнение 2

В показанном ниже примере код интерфейса модуля содержит критическую ошибку. В чем она заключается и почему приводит к проблемам?

```
histogramUI <- function(id) {
  tagList(
    selectInput("var", "Variable", choices = names(mtcars)),
    numericInput("bins", "bins", value = 10, min = 1),
    plotOutput("hist")
  )
}
```

Упражнение 3

Следующий модуль предназначен для генерирования случайного числа при каждом нажатии на кнопку `go`:

```
randomUI <- function(id) {
  tagList(
    textOutput(NS(id, "val")),
    actionButton(NS(id, "go"), "Go!")
  )
}
randomServer <- function(id) {
  moduleServer(id, function(input, output, session) {
    rand <- eventReactive(input$go, sample(100, 1))
    output$val <- renderText(rand())
  })
}
```

Создайте приложение, выводящее четыре копии этого модуля на одной странице. Убедитесь, что каждый модуль работает независимо. Как бы вы изменили возвращаемое функцией `randomUI()` значение, чтобы сделать приложение внешне более привлекательным?

Упражнение 4

Вам еще не надоело писать один и тот же шаблон модулей? Почитайте о сниппетах в RStudio по адресу <https://support.rstudio.com/hc/en-us/articles/204463668-Code-Snippets> и добавьте в конфигурацию RStudio следующий сниппет, облегчающий создание новых модулей:

```
#{1}UI <- function(id) {
  tagList(
    #{2}
  )
}

#{1}Server <- function(id) {
  moduleServer(id, function(input, output, session) {
    #{3}
  })
}
```

Ввод и вывод

Иногда модули с единственным аргументом `id` в интерфейсе и сервере могут быть полезны тем, что позволяют изолировать сложный блок кода в отдельном файле. Это может быть особенно важно для приложений, состоящих из независимых компонентов, например корпоративных дашбордов, в которых на каждой вкладке располагается отдельный отчет для своего вида деятельности. В таком случае модули помогут вам разрабатывать все отчеты по отдельности и хранить в собственных файлах, не беспокоясь о возможных конфликтах имен с другими компонентами.

Но чаще всего модули предполагают передачу в них дополнительных аргументов. В интерфейсе модуля эти аргументы могут помочь улучшить внешний вид элементов, что позволит использовать один и тот же модуль в раз-

ных частях приложения. Но интерфейс модуля – это всего лишь функция R, так что ничего нового мы здесь не расскажем, о функциях все необходимое было сказано в главе 18.

В следующих разделах мы сосредоточимся на серверной части модуля и поговорим о том, как передавать модулю дополнительный реактивный ввод и возвращать из него один или несколько реактивных выводов. В отличие от традиционного кода Shiny, связывание модулей воедино требует явного указания элементов ввода и вывода. Это может показаться очень скучным и утомительным занятием, и в Shiny действительно придется выполнять чуть больше работы в этом отношении по сравнению с обычным кодом R. Но на то есть свои причины. Несмотря на чуть большее количество действий, которые необходимо выполнить при создании модулей, в дальнейшем они позволяют облегчить чтение кода и создавать довольно сложные приложения.

Вы могли встретить советы использовать переменную `session$userData` и другие техники для облегчения работы с модулями в Shiny. Я советую относиться к ним с большой осторожностью. Все эти советы направлены на обход требований, налагаемых пространствами имен, и следование им может внести лишь дополнительную сложность в ваши приложения и даже свести на нет все преимущества от использования модулей в Shiny.

Приступим: интерфейсный ввод и серверный вывод

Чтобы посмотреть, как работают ввод и вывод, начнем с модуля, позволяющего пользователю выбрать набор данных из вариантов, предлагаемых пакетом `datasets`. Это будет не очень полезное приложение само по себе, но вполне показательное в плане демонстрации основных принципов; кроме того, оно может служить важным строительным блоком для более сложных модулей, что вы уже видели в главе 6.

Начнем с интерфейса модуля и будем использовать один дополнительный аргумент, который позволит пользователю ограничить выбор наборов данных датафреймами (`filter = is.data.frame`) или матрицами (`filter = is.matrix`). Применим этот аргумент в качестве фильтра к наборам данных, после чего создадим уже знакомый нам элемент ввода `selectInput()`:

```
datasetInput <- function(id, filter = NULL) {
  names <- ls("package:datasets")
  if (!is.null(filter)) {
    data <- lapply(names, get, "package:datasets")
    names <- names[vapply(data, filter, logical(1))]
  }

  selectInput(NS(id, "dataset"), "Pick a dataset", choices = names)
}
```

Серверная логика модуля также будет предельно простой: мы используем функцию `get()` для получения набора данных по имени. С этой функцией

будет связана одна новая идея: подобно обычной функции и в отличие от традиционной функции `server()` она будет возвращать значение. И здесь мы воспользуемся правилом о том, что последнее выражение, вычисленное в рамках функции, становится ее возвращаемым значением¹. Это возвращаемое значение всегда должно быть реактивным:

```
datasetServer <- function(id) {
  moduleServer(id, function(input, output, session) {
    reactive(get(input$dataset, "package:datasets"))
  })
}
```

Чтобы серверная функция модуля вернула значение, достаточно захватить его при помощи оператора присваивания `<-`. Это показано в следующем приложении, где мы захватываем выбранный пользователем набор данных и отображаем при помощи табличного элемента вывода `tableOutput`:

```
datasetApp <- function(filter = NULL) {
  ui <- fluidPage(
    datasetInput("dataset", filter = filter),
    tableOutput("data")
  )
  server <- function(input, output, session) {
    data <- datasetServer("dataset")
    output$data <- renderTable(head(data()))
  }
  shinyApp(ui, server)
}
```

Стоит отметить еще два нюанса, характерных для этого приложения:

- функция приложения принимает аргумент `filter`, который впоследствии передается функции интерфейса модуля, что облегчает экспериментирование с приложением;
- я выбрал табличный элемент вывода, чтобы показать все данные. В этом случае не очень важно, что вы увидите, но чем выразительнее будет ваш интерфейс, тем легче будет понять, что модуль выполняет всю возложенную на него работу.

Практический пример: выбор числовой переменной

Теперь давайте создадим элемент управления, позволяющий пользователю сделать выбор переменной определенного типа из заданного реактивного набора данных. Поскольку мы хотим, чтобы набор данных был реактивным,

¹ Руководство по стилю программирования с `tidyverse`, расположенное по адресу <https://style.tidyverse.org/functions.html#return>, советует использовать `return()` только в случае, если вы возвращаете значение из функции раньше ее окончания.

мы не можем заполнить элементы списка при запуске приложения. В результате интерфейс модуля у нас будет весьма простым:

```
selectVarInput <- function(id) {
  selectInput(NS(id, "var"), "Variable", choices = NULL)
}
```

Серверная функция будет принимать на вход два аргумента:

- аргумент `data`, представляющий собой набор данных, из которого пользователь будет выбирать переменную. Мы хотим, чтобы он был реактивным и мог работать с модулем `dataset`, созданным ранее;
- аргумент `filter` определяет, какие переменные включать в список. Он будет устанавливаться тем, кто вызывает модуль, так что быть реактивным ему ни к чему. Для максимального облегчения реализации серверной функции модуля я выделил ключевую идею в отдельную функцию:

```
find_vars <- function(data, filter) {
  names(data)[vapply(data, filter, logical(1))]
}
```

После этого в серверной части модуля используем функцию `observeEvent()` для обновления списка элементов в `inputSelect` при изменении входных данных и возвращаем реактив со значениями выбранной переменной:

```
selectVarServer <- function(id, data, filter = is.numeric) {
  moduleServer(id, function(input, output, session) {
    observeEvent(data(), {
      updateSelectInput(session, "var", choices = find_vars(data(), filter))
    })

    reactive(data()[[input$var]])
  })
}
```

Для создания приложения мы снова захватим результат серверной функции модуля и соединим его с выводом в интерфейсе. Чтобы убедиться, что с нашими реактивами все в порядке, используем модуль `dataset` в качестве источника реактивных датафреймов:

```
selectVarApp <- function(filter = is.numeric) {
  ui <- fluidPage(
    datasetInput("data", is.data.frame),
    selectVarInput("var"),
    verbatimTextOutput("out")
  )
  server <- function(input, output, session) {
    data <- datasetServer("data")
    var <- selectVarServer("var", data, filter = filter)
    output$out <- renderPrint(var())
  }

  shinyApp(ui, server)
}
```

Серверный ввод

При разработке серверной функции модуля необходимо думать о том, кто будет предоставлять значение для каждого аргумента: будет ли это программист R, вызывающий ваш модуль, или пользователь, работающий с приложением. Также этот вопрос можно поставить иначе: может ли значение меняться, т. е. будет ли оно оставаться фиксированным на протяжении всего жизненного цикла приложения или реактивным, то есть будет меняться в зависимости от действий пользователя. Это очень важный вопрос, определяющий, будет ли аргумент реактивным.

После принятия этого решения неплохо будет проверить все входящие в модель значения на предмет их реактивности. Если этого не сделать, пользователю может быть возвращено замысловатое сообщение об ошибке. Можно значительно облегчить жизнь пользователю приложения, если использовать простую и надежную функцию `stopifnot()`. Например, в функции `selectVarServer()` вы можете удостовериться, что аргумент `data` является реактивным, а `filter` – нет, при помощи следующего простого кода:

```
selectVarServer <- function(id, data, filter = is.numeric) {
  stopifnot(is.reactive(data))
  stopifnot(!is.reactive(filter))

  moduleServer(id, function(input, output, session) {
    observeEvent(data(), {
      updateSelectInput(session, "var", choices = find_vars(data(), filter))
    })

    reactive(data()[[input$var]])
  })
}
```

Если предполагается, что ваш модуль будет использоваться многократно разными разработчиками, можно прописать в нем более человеческий вывод ошибок с условными конструкциями `if` и вызовами функции `stop()`.

Проверка входящих в модуль параметров на реактивность позволяет избежать распространенной проблемы при совмещении с другими модулями со своими элементами ввода. `input$var` не является реактивом, так что каждый раз при передаче значения элемента ввода в модуль вам придется оборачивать его в функцию `reactive()` (например, `selectVarServer("var", reactive(input$x))`). Если вы будете проверять ввод так, как я здесь рекомендую, вы увидите понятное сообщение об ошибке, а если нет, то на экран будет выведено что-то вроде *could not find function "data"*.

Примечание. Этот же прием вы можете применить к функции `find_vars()`. В данном случае это не так важно, но с учетом того, что приложения Shiny отлаживать куда сложнее, чем традиционный код на R, думаю, стоит потратить немного времени на проверку ввода, чтобы в случае чего получать человеческие сообщения об ошибках:

```
find_vars <- function(data, filter) {
  stopifnot(is.data.frame(data))
```

```

    stopifnot(is.function(filter))
    names(data)[vapply(data, filter, logical(1))]
  }

```

Кстати, это помогло мне отловить пару ошибок при написании данной главы.

Вложенные модули

Перед тем как продолжить говорить о возвращаемых серверными функциями значениях, я бы хотел отметить, что модули могут быть вложены друг в друга, и такую технику нередко бывает полезно использовать. К примеру, мы могли бы скомбинировать модули `dataset` и `selectVar`, чтобы создать единый модуль, позволяющий пользователю выбирать переменную из встроенного набора данных:

```

selectDataVarUI <- function(id) {
  tagList(
    datasetInput(NS(id, "data"), filter = is.data.frame),
    selectVarInput(NS(id, "var"))
  )
}

selectDataVarServer <- function(id, filter = is.numeric) {
  moduleServer(id, function(input, output, session) {
    data <- datasetServer("data")
    var <- selectVarServer("var", data, filter = filter)
    var
  })
}

selectDataVarApp <- function(filter = is.numeric) {
  ui <- fluidPage(
    sidebarLayout(
      sidebarPanel(selectDataVarUI("var")),
      mainPanel(verbatimTextOutput("out"))
    )
  )
  server <- function(input, output, session) {
    var <- selectDataVarServer("var", filter)
    output$out <- renderPrint(var(), width = 40)
  }
  shinyApp(ui, server)
}

```

Практический пример: гистограмма

Теперь давайте вернемся к нашему первому примеру с гистограммой и улучшим его при помощи вложенности модулей. Главной сложностью при создании модулей является написание функций, которые, с одной стороны, будут достаточно гибкими для использования в разных местах приложения, но в то же время будут сохранять простоту для лучшего их понимания. Осознание

того, как создавать функции, которые могут стать полноценными строительными блоками при проектировании новых приложений, приходит далеко не сразу, и вы можете сразу подготовиться к тому, что допустите немало ошибок, прежде чем у вас начнет что-то получаться. Конечно, мне хотелось бы здесь дать вам более обнадеживающий совет, но как уж есть – этот навык вам действительно придется оттачивать на протяжении довольно долгого времени.

В данном примере мы будем рассматривать наш элемент как элемент вывода – несмотря на то что он будет также отвечать за ввод (пользователь сможет вводить количество столбиков), он больше будет использоваться для отображения информации и не будет возвращаться из модуля:

```
histogramOutput <- function(id) {
  tagList(
    numericInput(NS(id, "bins"), "bins", 10, min = 1, step = 1),
    plotOutput(NS(id, "hist"))
  )
}
```

Я решил дать модулю два входящих значения: `x`, представляющий собой переменную для вывода на диаграмме, и `title` для заголовка графика. Оба параметра будут реактивными, так что они могут менять свои значения с течением времени. Параметр для заголовка диаграммы – это немного странно, но на его примере мы покажем одну очень важную технику. Заметьте, что мы дали параметру `title` значение по умолчанию. Поскольку заголовок у нас будет реактивным, мы заключили константу в функцию `reactive()`:

```
histogramServer <- function(id, x, title = reactive("Histogram")) {
  stopifnot(is.reactive(x))
  stopifnot(is.reactive(title))

  moduleServer(id, function(input, output, session) {
    output$hist <- renderPlot({
      req(is.numeric(x()))
      main <- paste0(title(), " [", input$bins, "]")
      hist(x(), breaks = input$bins, main = main)
    }, res = 96)
  })
}

histogramApp <- function() {
  ui <- fluidPage(
    sidebarLayout(
      sidebarPanel(
        datasetInput("data", is.data.frame),
        selectVarInput("var"),
      ),
      mainPanel(
        histogramOutput("hist")
      )
    )
  )
}
```



```

server <- function(input, output, session) {
  data <- datasetServer("data")
  x <- selectVarServer("var", data)
  histogramServer("hist", x)
}
shinyApp(ui, server)
}
# histogramApp()

```

Примечание. Если вы хотите разместить элементы управления гистограммой и саму диаграмму в разных местах приложения, можно сделать это с применением нескольких функций интерфейса. В данном случае нам это не нужно, но вообще это делается примерно так:

```

histogramOutputBins <- function(id) {
  numericInput(NS(id, "bins"), "bins", 10, min = 1, step = 1)
}
histogramOutputPlot <- function(id) {
  plotOutput(NS(id, "hist"))
}

ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      datasetInput("data", is.data.frame),
      selectVarInput("var"),
      histogramOutputBins("hist")
    ),
    mainPanel(
      histogramOutputPlot("hist")
    )
  )
)

```

Множественный вывод

Было бы здорово включить в название диаграммы имя выбранной пользователем переменной. Но как это сделать? Ведь функция `selectVarServer()` возвращает только значение переменной, а не ее имя. Конечно, можно было бы переписать функцию, чтобы она возвращала имя, но тогда пользователю модуля пришлось бы заниматься выделением подмножества. Лучше вернуть и имя, и значение.

Серверная функция модуля может возвращать несколько элементов так же точно, как и любая другая функция, – посредством списков. Давайте изменим функцию `selectVarServer()` таким образом, чтобы она возвращала и имя переменной, и значение в виде реактивов:

```

selectVarServer <- function(id, data, filter = is.numeric) {
  stopifnot(is.reactive(data))
  stopifnot(!is.reactive(filter))

```

```

moduleServer(id, function(input, output, session) {
  observeEvent(data(), {
    updateSelectInput(session, "var", choices = find_vars(data(), filter))
  })

  list(
    name = reactive(input$var),
    value = reactive(data()[[input$var]])
  )
})
}

```

Теперь можно соответствующим образом переписать нашу функцию `histogramApp()`. Интерфейс останется прежним, но на этот раз мы будем передавать функции `histogramServer()` и имя переменной, и значение:

```

histogramApp <- function() {
  ui <- fluidPage(...)

  server <- function(input, output, session) {
    data <- datasetServer("data")
    x <- selectVarServer("var", data)
    histogramServer("hist", x$value, x$name)
  }
  shinyApp(ui, server)
}

```

Основные сложности с такого рода кодом связаны с запоминанием того, когда вы используете реактивы (вроде `x$value()`), а когда просто значения (`x$value`). Запомните: когда при передаче аргумента в модуль вам нужно, чтобы модуль реагировал на изменение его значения, вы должны передавать реактив, а не значение.

Если вам часто приходится возвращать несколько значений из реактива, вы можете воспользоваться пакетом *zeallot* (<https://github.com/r-lib/zeallot>). Этот пакет предоставляет в ваше распоряжение оператор `%<-%`, позволяющий присваивать значения сразу нескольким переменным (иногда его называют оператором множественного, распаковывающего или деструктурирующего присваивания). Его использование может быть полезным при возврате нескольких значений, поскольку позволяет избежать косвенной адресации:

```

library(zeallot)

histogramApp <- function() {
  ui <- fluidPage(...)

  server <- function(input, output, session) {
    data <- datasetServer("data")
    c(value, name) %<-% selectVarServer("var", data)
    histogramServer("hist", value, name)
  }
  shinyApp(ui, server)
}

```

Упражнения

Упражнение 1

Перепишите функцию `selectVarServer()` таким образом, чтобы оба параметра (`data` и `filter`) стали реактивными. После этого используйте ее в приложении так, чтобы пользователь мог выбрать набор данных при помощи модуля `dataset` и фильтрующей функции с использованием элемента ввода `inputSelect()`. Дайте возможность пользователю фильтровать числовые, символьные и факторные переменные.

Упражнение 2

В следующем примере мы определили вывод и серверную логику модуля, принимающего на вход числовое значение и выводящего маркированный список с тремя статистическими показателями. Создайте функцию приложения, которая позволит вам экспериментировать. Функция должна принимать на вход датафрейм и использовать элемент ввода `numericVarSelectInput()` для выбора переменной агрегации:

```
summaryOutput <- function(id) {
  tags$ul(
    tags$li("Min: ", textOutput(NS(id, "min"), inline = TRUE)),
    tags$li("Max: ", textOutput(NS(id, "max"), inline = TRUE)),
    tags$li("Missing: ", textOutput(NS(id, "n_na"), inline = TRUE))
  )
}

summaryServer <- function(id, var) {
  moduleServer(id, function(input, output, session) {
    rng <- reactive({
      req(var())
      range(var(), na.rm = TRUE)
    })

    output$min <- renderText(rng()[[1]])
    output$max <- renderText(rng()[[2]])
    output$n_na <- renderText(sum(is.na(var())))
  })
}
```

Упражнение 3

В следующем интерфейсе модуля представлен текстовый элемент для ввода даты в формате ISO8601 (yyyy-mm-dd). Напишите серверную функцию модуля, в которой будет использоваться элемент вывода `output$errorg` для отображения сообщения об ошибке в случае некорректного формата даты. Модуль должен возвращать объект типа `Date` для правильно введенных дат. Подсказка: используйте функцию `strptime(x, "%Y-%m-%d")` для преобразования строки; она вернет NA, если значение не будет соответствовать заданному формату.

```
ymdDateUI <- function(id, label) {
  label <- paste0(label, " (yyyy-mm-dd)")
}
```

```
fluidRow(
  textInput(NS(id, "date"), label),
  textOutput(NS(id, "error"))
)
```

ПРАКТИЧЕСКИЕ ПРИМЕРЫ

Суммируя все сказанное выше, можно отметить следующее:

- входящие аргументы модуля (т. е. аргументы, поступающие на вход серверной функции) могут представлять собой как реактивы, так и константы. Выбор необходимо сделать на основании того, кто устанавливает аргумент и когда он меняется. Вы всегда должны проверять тип входного параметра, чтобы избежать появления непредвиденных ошибок;
- в отличие от серверной функции приложения и подобно традиционным функциям в R, серверная функция модуля может возвращать значение. Возвращаемое модулем значение должно являться реактивом или, если вы хотите вернуть сразу несколько значений, списком.

Чтобы вы могли лучше усвоить материал этой главы, я приведу еще несколько примеров работы с модулями. К сожалению, я не смогу подробно рассказать обо всех приемах работы с модулями с целью упрощения приложений, но попытаюсь отметить важные моменты, которые помогут вам изучить эту тему более скрупулезно.

Ограниченный выбор вариантов и пункт Другие

Еще одно важное применение модулей в Shiny связано с упрощением интерфейса для достаточно сложных элементов управления. В данном разделе мы посмотрим, как можно реализовать очень полезный элемент управления, не присутствующий в Shiny по умолчанию, а именно ограниченный набор переключателей с дополнительным вариантом *Other* (Другие). Внутри себя модуль будет содержать много разных элементов, тогда как внешне он будет работать как единый объект.

Мы параметризуем интерфейс пользователя при помощи аргументов `label`, `choices` и `selected`, которые будут переданы непосредственно в функцию `radioButtons()`. Также создадим элемент ввода `textInput()`, содержащий заменитель и по умолчанию установленный в значение "Other". Для объединения текстовых полей с переключателями воспользуемся тем, что аргумент `choiceNames` может быть представлен списком элементов HTML, включая другие виджеты. На рис. 19.3 показано, как будет выглядеть интерфейс нашего модуля.

```
radioExtraUI <- function(id, label, choices, selected = NULL, placeholder = "Other") {
  other <- textInput(NS(id, "other"), label = NULL, placeholder = placeholder)
```

```

names <- if (is.null(names(choices))) choices else names(choices)
values <- unname(choices)

radioButtons(NS(id, "primary"),
  label = label,
  choiceValues = c(names, "other"),
  choiceNames = c(as.list(values), list(other)),
  selected = selected
)
}

```

How do you usually read csv files?

☒ read.csv()
☐ readr::read_csv()
☐ data.table::fread()
☐

Рис. 19.3 ❖ Пример использования интерфейса radioExtraUI() для ответа на вопрос о том, при помощи какого инструмента вы обычно читаете файлы CSV

В серверной логике модуля я хочу сделать так, чтобы переключатель *Other* устанавливался автоматически при вводе текста в поле. Вы также можете настроить проверку ввода текста в поле при выборе варианта *Other*:

```

radioExtraServer <- function(id) {
  moduleServer(id, function(input, output, session) {
    observeEvent(input$other, ignoreInit = TRUE, {
      updateRadioButtons(session, "primary", selected = "other")
    })

    reactive({
      if (input$primary == "other") {
        input$other
      } else {
        input$primary
      }
    })
  })
}

```

Пришло время объединить обе функции в приложение и протестировать его. Мы используем троеточие (...) для передачи произвольного количества аргументов в функцию radioExtraUI():

```

radioExtraApp <- function(...) {
  ui <- fluidPage(
    radioExtraUI("extra", ...),

```

```

      textOutput("value")
    )
    server <- function(input, output, server) {
      extra <- radioExtraServer("extra")
      output$value <- renderText(paste0("Selected: ", extra()))
    }

    shinyApp(ui, server)
  }

```

На рис. 19.4 показан внешний вид приложения.

How do you usually read csv files?

- ☐ read.csv()
- ☐ readr::read_csv()
- ☐ data.table::fread()
- ☒ vroom::vroom()

Selected: vroom::vroom()

Рис. 19.4 ❖ Приложение radioExtraApp() с ответом на вопрос о способе чтения файлов CSV.

Теперь, если начать ввод в текстовое поле, установится соответствующий переключатель *Other*

Созданный модуль можно использовать для разных специфических целей. К примеру, вы можете задействовать его для единообразного выбора пола пользователем, поскольку люди очень по-разному указывают свой пол:

```

genderUI <- function(id, label = "Gender") {
  radioExtraUI(id,
    label = label,
    choices = c(
      male = "Male",
      female = "Female",
      na = "Prefer not to say"
    ),
    placeholder = "Self-described",
    selected = "na"
  )
}

```

В данном случае важно, чтобы были представлены основные пункты для мужчин и женщин, а также вариант для тех, кто не желает указывать свой пол. Поле со свободной записью можно оставить для пользователей, которые хотят выразить свою половую принадлежность другими словами. Также можно вовсе не указывать этот вариант.

Мастер

Далее мы рассмотрим пару примеров, которые позволят вам еще больше погрузиться в тонкости пространств имен. Интерфейс пользователя здесь будет генерироваться автоматически в разное время и разными людьми. Такие ситуации достаточно сложны, и чтобы в них разбираться, нужно очень хорошо понимать принципы работы пространств имен.

Начнем с модуля, реализующего интерфейс *мастера* (wizard). Мастер представляет собой процесс, разбитый на страницы, через которые пользователь проходит по очереди, как при установке программы. В главе 10 мы уже разрабатывали похожий интерфейс, а сейчас полностью автоматизируем процесс, что позволит вам сосредоточиться на содержимом каждой страницы, а не на том, как они связаны друг с другом.

При описании модуля мы будем двигаться поэтапно снизу вверх. Основу интерфейса мастера составляют кнопки. Почти на каждой странице их ровно две: одна для движения вперед, вторая для возвращения на предыдущую страницу. Начнем со вспомогательных функций для этих кнопок:

```
nextPage <- function(id, i) {
  actionButton(NS(id, paste0("go_", i, "_", i + 1)), "next")
}
prevPage <- function(id, i) {
  actionButton(NS(id, paste0("go_", i, "_", i - 1)), "prev")
}
```

Единственную сложность здесь может представлять `id`: поскольку каждый элемент управления должен иметь уникальный идентификатор, для каждой кнопки он будет включать в себя номера текущей страницы и страницы назначения.

Теперь давайте напомним функцию для генерирования страницы мастера. В нее будут входить *заголовок* (title), который не будет отображаться, но будет использоваться для навигации, содержание страницы, предоставленное пользователем, и две кнопки¹:

```
wrapPage <- function(title, page, button_left = NULL, button_right = NULL) {
  tabPanel(
    title = title,
    fluidRow(
      column(12, page)
    ),
    fluidRow(
      column(6, button_left),
      column(6, button_right)
    )
  )
}
```

¹ Не на всех страницах будут видны обе кнопки, о чем мы поговорим далее, поэтому я пометил их как необязательные, снабдив значением по умолчанию, равным `NULL`.

На данном этапе мы можем собрать наш мастер воедино, как показано на рис. 19.5. Для этого пройдемся по страницам, предоставленным пользователем, создадим кнопки, заключим каждую страницу в `tabPanel`, а затем объединим все панели в единый `tabsetPanel`. Обратите внимание на два важных нюанса, касающихся кнопок:

- на первой странице мастера не будет кнопки возвращения на предыдущую страницу. Здесь я использовал трюк, заключающийся в том, что оператор `if` возвращает значение `NULL`, если условие не выполняется и отсутствует блок `else`;
- на последней странице мастера мы будем выводить кнопку, предоставленную пользователем. Полагаю, это самый простой способ позволить пользователю осуществлять контроль за тем, что должно происходить по окончании работы мастера.

```
wizardUI <- function(id, pages, doneButton = NULL) {
  stopifnot(is.list(pages))
  n <- length(pages)

  wrapped <- vector("list", n)
  for (i in seq_along(pages)) {
    # На первой странице есть только кнопка движения вперед; а на последней –
    # кнопки возврата и завершения
    lhs <- if (i > 1) prevPage(id, i)
    rhs <- if (i < n) nextPage(id, i) else doneButton
    wrapped[[i]] <- wrapPage(paste0("page_", i), pages[[i]], lhs, rhs)
  }

  # Создаем tabsetPanel
  # https://github.com/rstudio/shiny/issues/2927
  wrapped$id <- NS(id, "wizard")
  wrapped$type <- "hidden"
  do.call("tabsetPanel", wrapped)
}
```

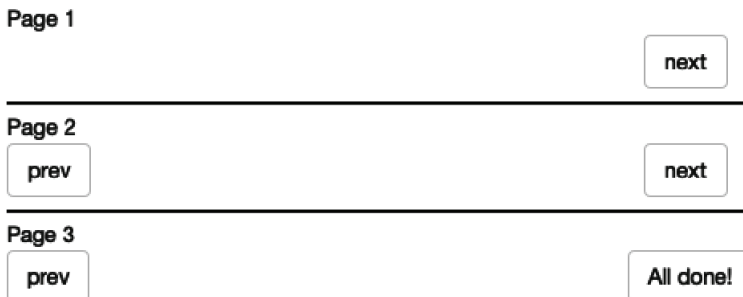


Рис. 19.5 ❖ Простой пример мастера

Код для создания вкладок требует небольшого пояснения. К сожалению, `tabsetPanel()` не позволяет передавать список вкладок, поэтому нам пришлось применить немного магии с использованием функции `do.call()`.

Вызов функции `do.call(function_name, list(arg1, arg2, ...))` эквивалентен записи `function_name(arg1, arg2, ...)`, так что здесь мы просто осуществляем вызов вида `tabsetPanel(pages[[1]], pages[[2]], ..., id = NS(id, "wizard"), type = "hidden")`. Надеюсь, в следующих версиях Shiny этот процесс будет упрощен.

Теперь, когда мы разделились с интерфейсом модуля, пришло время обратиться к его серверной логике. Суть здесь будет очень простая: нам нужно реализовать перемещение по страницам мастера при помощи кнопок. Для этого нам необходимо для каждой кнопки настроить функцию `observeEvent()` с вызовом `updateTabsetPanel()`. Это было бы весьма просто, если бы мы точно знали количество страниц в мастере. Но это не так, поскольку этим заведует сам пользователь.

Что ж, придется применить навыки функционального программирования для настройки $(n - 1) * 2$ наблюдателей (по два наблюдателя на каждую страницу, за исключением первой и последней, где будет по одному наблюдателю). В представленной ниже серверной функции мы объединяем весь код для кнопок в функции `changePage()`. Здесь мы используем записи `input[[]]`, как уже было показано в главе 10, чтобы можно было обращаться к элементу управления динамически. После этого применяем функцию `lapply()` для прохода по всем кнопкам возвращения (что необходимо для всех страниц, за исключением первой) и кнопкам перехода дальше (для всех страниц, кроме последней):

```
wizardServer <- function(id, n) {
  moduleServer(id, function(input, output, session) {
    changePage <- function(from, to) {
      observeEvent(input[[paste0("go_", from, "_", to)]], {
        updateTabsetPanel(session, "wizard", selected = paste0("page_", to))
      })
    }
    ids <- seq_len(n)
    lapply(ids[-1], function(i) changePage(i, i - 1))
    lapply(ids[-n], function(i) changePage(i, i + 1))
  })
}
```

Замечу, что в данном случае не получится использовать цикл `for` вместо функций `map()/lapply()`. Цикл `for` работает, изменяя значение одной и той же переменной `i`, так что к окончанию цикла все функции `changePage()` будут использовать одно и то же значение. Функции `map()` и `lapply()` создают собственные окружения, каждое со своим значением переменной `i`.

Теперь мы можем создать приложение с простым примером, чтобы проверить, что мы все собрали правильно:

```
wizardApp <- function(...) {
  pages <- list(...)

  ui <- fluidPage(
    wizardUI("whiz", pages)
```

```

    )
    server <- function(input, output, session) {
      wizardServer("whiz", length(pages))
    }
    shinyApp(ui, server)
  }

```

К сожалению, при написании модулей нам каждый раз приходится немало повторяться. Кроме того, нужно внимательно следить, чтобы аргумент `n` в функции `wizardServer()` в точности соответствовал аргументу `pages` в `wizardUI()`. Это характерное ограничение модульных систем, о котором мы поговорим подробнее совсем скоро.

А сейчас давайте попробуем использовать наш мастер в реальном примере, показанном на рис. 19.6. Обратите внимание, что хоть страницы и выводятся при помощи модуля, их идентификаторы находятся в полной власти пользователя модуля. Разработчик, создающий компонент, определяет только его имя, а сборку для показа на странице может осуществлять кто угодно:

```

page1 <- tagList(
  textInput("name", "What's your name?")
)
page2 <- tagList(
  numericInput("age", "How old are you?", 20)
)
page3 <- tagList(
  "Is this data correct?",
  verbatimTextOutput("info")
)

ui <- fluidPage(
  wizardUI(
    id = "demographics",
    pages = list(page1, page2, page3),
    doneButton = actionButton("done", "Submit")
  )
)

server <- function(input, output, session) {
  wizardServer("demographics", 3)

  observeEvent(input$done, showModal(
    modalDialog("Thank you!", footer = NULL)
  ))

  output$info <- renderText(paste0(
    "Age: ", input$age, "\n",
    "Name: ", input$name, "\n"
  ))
}

```

What's your name?

next

How old are you?

20

prev next

Is this data correct?

Age: 20
Name:

prev Submit

Рис. 19.6 ❖ Простой, но полноценный мастер, созданный при помощи нашего модуля

Динамический интерфейс пользователя

Завершим изучение модулей практическим примером на тему динамического построения пользовательского интерфейса, позаимствовав код из раздела с динамической фильтрацией из главы 10 и превратив его в модуль. Основная сложность при формировании динамического интерфейса внутри модуля заключается в более внимательном отношении к явным определениям пространств имен, что связано с тем, что мы будем генерировать код пользовательского интерфейса непосредственно в серверной функции.

Как обычно, начнем с интерфейса модуля. В данном случае он будет очень простым, поскольку мы определим только шаблон, а заполнять его будем динамически в серверной функции:

```
filterUI <- function(id) {
  uiOutput(NS(id, "controls"))
}
```

Для создания серверной функции модуля мы сначала скопируем вспомогательные функции из раздела «Динамическая фильтрация» главы 10: в функции `make_ui()` создаются элементы управления для каждого столбца, а функция `filter_var()` помогает сгенерировать окончательный логический вектор. По сравнению с исходным примером здесь есть лишь одно отличие: функция `make_ui()` принимает дополнительный аргумент `id` для организации пространства имен в модуле:

```
library(purrr)

make_ui <- function(x, id, var) {
```

```

if (is.numeric(x)) {
  rng <- range(x, na.rm = TRUE)
  sliderInput(id, var, min = rng[1], max = rng[2], value = rng)
} else if (is.factor(x)) {
  levs <- levels(x)
  selectInput(id, var, choices = levs, selected = levs, multiple = TRUE)
} else {
  # Не поддерживается
  NULL
}
}

filter_var <- function(x, val) {
  if (is.numeric(x)) {
    !is.na(x) & x >= val[1] & x <= val[2]
  } else if (is.factor(x)) {
    x %in% val
  } else {
    # Без фильтра
    TRUE
  }
}
}

```

Теперь напишем серверную функцию модуля. Здесь есть два важных момента:

- элементы управления мы будем генерировать при помощи функций `purrr::map()` и `make_ui()`. Обратите внимание, что здесь мы явным образом используем функцию `NS()`. Это необходимо, поскольку, хоть мы и находимся внутри серверной функции модуля, автоматически пространство имен применяется только для объектов `input`, `output` и `session`;
- на выходе функция вернет логический фильтрующий вектор.

```

filterServer <- function(id, df) {
  stopifnot(is.reactive(df))

  moduleServer(id, function(input, output, session) {
    vars <- reactive(names(df()))

    output$controls <- renderUI({
      map(vars(), function(var) make_ui(df()[[var]], NS(id, var), var))
    })

    reactive({
      each_var <- map(vars(), function(var) filter_var(df()[[var]], input[[var]]))
      reduce(each_var, '&')
    })
  })
}

```

Теперь пришло время собрать наше модульное приложение, в котором пользователь сможет выбрать встроенный набор данных и отфильтровать его по числовым или категориальным переменным:

```
filterApp <- function() {  
  ui <- fluidPage(  
    sidebarLayout(  
      sidebarPanel(  
        datasetInput("data", is.data.frame),  
        textOutput("n"),  
        filterUI("filter"),  
      ),  
      mainPanel(  
        tableOutput("table")  
      )  
    )  
  )  
}  
  
server <- function(input, output, session) {  
  df <- datasetServer("data")  
  filter <- filterServer("filter", df)  
  
  output$table <- renderTable(df()[filter(), , drop = FALSE])  
  output$n <- renderText(paste0(sum(filter()), " rows"))  
}  
shinyApp(ui, server)  
}
```

Большим преимуществом использования модуля здесь является то, что мы смогли включить в него массу продвинутых техник Shiny. Вы можете использовать модуль фильтрации, даже не понимая, как он работает, и не владея навыками функционального программирования, лежащего в его основе.

Модули в ВИДЕ ЕДИНОГО ОБЪЕКТА

В заключение этой главы я хотел бы затронуть тему распространенной реакции на модули в Shiny. Вы можете не читать этот раздел, если у вас подобная реакция не возникла. Когда кто-то (вроде меня) сталкивается с модулями впервые, он сразу пытается объединить в один объект интерфейс и серверную функцию модуля. Для иллюстрации этого давайте обобщим первый пример данной главы, добавив к нему датафрейм в виде параметра:

```
histogramUI <- function(id, df) {  
  tagList(  
    selectInput(NS(id, "var"), "Variable", names(df)),  
    numericInput(NS(id, "bins"), "bins", 10, min = 1),  
    plotOutput(NS(id, "hist"))  
  )  
}  
  
histogramServer <- function(id, df) {  
  moduleServer(id, function(input, output, session) {  
    data <- reactive(df[[input$var]])  
    output$hist <- renderPlot({
```

```

      hist(data(), breaks = input$bins, main = input$var)
    }, res = 96)
  })
}

```

В результате получим приложение следующего вида:

```

ui <- fluidPage(
  tabsetPanel(
    tabPanel("mtcars", histogramUI("mtcars", mtcars)),
    tabPanel("iris", histogramUI("iris", iris))
  )
)

server <- function(input, output, session) {
  histogramServer("mtcars", mtcars)
  histogramServer("iris", iris)
}

```

Естественной реакцией разработчика будет попытаться избавиться от повторения идентификаторов и имен наборов данных в интерфейсной и серверной частях модуля, например путем объединения вызовов в единую функцию, которая будет возвращать оба объекта:

```

histogramApp <- function(id, df) {
  list(
    ui = histogramUI(id, df),
    server = histogramServer(id, df)
  )
}

```

После этого можно определить модуль за пределами интерфейса и серверной функции, извлекая нужные элементы из списка, следующим образом:

```

hist1 <- histogramApp("mtcars", mtcars)
hist2 <- histogramApp("iris", iris)

ui <- fluidPage(
  tabsetPanel(
    tabPanel("mtcars", hist1$ui()),
    tabPanel("iris", hist2$ui())
  )
)

server <- function(input, output, session) {
  hist1$server()
  hist2$server()
}

```

С этим кодом есть две проблемы. Во-первых, он не работает, поскольку функция `moduleServer()` должна быть вызвана внутри серверной функции. Но давайте представим, что с этим все в порядке и мы как-то сумели обойти это ограничение. В этом случае мы тут же столкнемся с еще одной пробле-

мой. Что, если мы захотим дать возможность пользователю выбирать набор данных, т. е. сделать аргумент `df` реактивным? Это не сработает, поскольку экземпляр модуля создается до серверной функции, а значит, и до того, как мы узнаем нужную нам информацию.

В Shiny интерфейс и серверная функция изначально отделены друг от друга: Shiny не знает, какой вызов интерфейса какой серверной сессии принадлежит. Вы можете наблюдать этот шаблон в Shiny повсюду: например, функции `plotOutput()` и `renderPlot()` связаны друг с другом только по общему идентификатору. И написание модулей в виде отдельных функций полностью отражает эти принципы: эти функции также связаны между собой исключительно по идентификатору.

ЗАКЛЮЧЕНИЕ

В данной главе вы научились использовать модули в Shiny, представляющие собой обобщенные функции, позволяющие определять связанные функции интерфейса и сервера в виде повторно используемых компонентов. Привыкнуть к модулям может быть не так просто, но как только вы освоитесь с ними, откроете для себя невероятно мощные техники для упрощения структуры своих приложений в целом.

В следующей главе вы узнаете, как можно преобразовать приложение Shiny в пакет с целью использования богатых возможностей для тестирования.

Глава 20

Пакеты

Если вы разрабатываете сложные приложения Shiny на постоянной основе, я бы очень рекомендовал вам организовывать их по примеру *пакетов* (package) в R. Для этого необходимо:

- перенести весь код в директорию *R/*;
- написать функцию для запуска приложения, т. е. такую, которая будет вызывать функцию `shinyApp()` с вашим интерфейсом и серверной частью;
- создать файл *DESCRIPTION* в корневой папке приложения.

Эта структура приближает вас к технологии разработки пакетов. Конечно, это не будет полноценный пакет, но, выполнив эти действия, вы сможете применять некоторые полезные инструменты, призванные облегчить работу со сложными приложениями. Кроме того, такая структура изрядно поможет вам на этапе тестирования, о котором мы будем говорить в главе 21, поскольку позволит воспользоваться специальным инструментарием для проверки приложения и узнавать, какие его части уже протестированы. Также вы получите возможность документировать сложные приложения при помощи пакета *roxygen2* (<https://roxygen2.r-lib.org>), хотя в данной книге мы не будем касаться этой темы.

Вам может казаться, что пакеты – это какие-то гигантские и сложные штуки вроде *Shiny*, *ggplot2* или *dplyr*. Но пакеты могут быть и очень простыми. Ключевая идея пакетов в R состоит в общем наборе правил по организации кода и других ресурсов, и если вы будете неукоснительно соблюдать эти правила, то в качестве бонуса сможете пользоваться разными полезными инструментами. В этой главе я расскажу вам о наиболее важных правилах и поделюсь несколькими хитростями и секретами.

Начав работать с приложениями-пакетами, вы наверняка испытаете новый прилив сил и захотите узнать о процессе создания пакетов больше. Я бы советовал начать с книги *R Packages* по адресу <https://r-pkgs.org> для знакомства с основами строения пакетов, после чего обратиться к книге *Engineering Production Grade Shiny Apps* от Колина Фэя (Colin Fay), Себастьяна Рошетта (Sébastien Rochette), Винсента Гюадера (Vincent Guyader) и Сервана Жирара (Cervan Girard), чтобы узнать больше о пакетах R применительно к Shiny.

Как обычно, начнем с загрузки пакета:

```
library(shiny)
```


ПРЕОБРАЗОВАНИЕ СУЩЕСТВУЮЩЕГО ПРИЛОЖЕНИЯ

Для преобразования готового приложения вам потребуется выполнить определенные действия. Предположим, ваше приложение называется *myApp* и уже находится в директории с именем *myApp/*. Вам необходимо выполнить следующие пункты:

- создайте папку с именем *R* и переместите в нее файл *app.R*;
- преобразуйте ваше приложение в обособленную функцию следующим образом:

```
library(shiny)

myApp <- function(...) {
  ui <- fluidPage(
    ...
  )
  server <- function(input, output, session) {
    ...
  }
  shinyApp(ui, server, ...)
}
```

- вызовите функцию `usethis::use_description()` для создания файла с описанием. Вам вряд ли часто понадобится заглядывать в этот файл, но он необходим для активации *режима разработки пакетов* (package development mode) в RStudio, позволяющего использовать определенные сочетания клавиш, о которых мы поговорим далее;
- если у вас еще не создан проект в RStudio, создайте его, вызвав функцию `usethis::use_rstudio()`;
- перезапустите RStudio и откройте проект заново.

Теперь вы можете нажать сочетание клавиш **Cmd/Ctrl+Shift+L** для вызова функции `devtools::load_all()` и загрузки всего кода с данными. Это означает, что вы можете:

- убрать все вызовы в функцию `source()`, поскольку функция `load_all()` автоматически извлекает все файлы *.R* в директории *R/*;
- если вы загружаете наборы данных с помощью функций вроде `read.csv()` или подобных ей, то можете вместо этого использовать функцию `usethis::use_data(mydataset)`, чтобы сохранить данные в папке *data/*. Функция `load_all()` автоматически загружает для вас данные.

Чтобы вы могли лучше понять суть процесса, мы рассмотрим простой пример, после чего поговорим о преимуществах выполненного преобразования.

Один файл

Представьте, что у вас есть относительно сложное приложение, код которого располагается в едином файле *app.R*:

```

library(shiny)

monthFeedbackUI <- function(id) {
  textOutput(NS(id, "feedback"))
}

monthFeedbackServer <- function(id, month) {
  stopifnot(is.reactive(month))

  moduleServer(id, function(input, output, session) {
    output$feedback <- renderText({
      if (month() == "October") {
        "You picked a great month!"
      } else {
        "Eh, you could do better."
      }
    })
  })
}

stones <- vroom::vroom("birthstones.csv")
birthstoneUI <- function(id) {
  p(
    "The birthstone for ", textOutput(NS(id, "month"), inline = TRUE),
    " is ", textOutput(NS(id, "stone"), inline = TRUE)
  )
}

birthstoneServer <- function(id, month) {
  stopifnot(is.reactive(month))

  moduleServer(id, function(input, output, session) {
    stone <- reactive(stones$stone[stones$month == month()])
    output$month <- renderText(month())
    output$stone <- renderText(stone())
  })
}

months <- c(
  "January", "February", "March", "April", "May", "June",
  "July", "August", "September", "October", "November", "December"
)

ui <- navbarPage(
  "Sample app",
  tabPanel("Pick a month",
    selectInput("month", "What's your favourite month?", choices = months)
  ),
  tabPanel("Feedback", monthFeedbackUI("tab1")),
  tabPanel("Birthstone", birthstoneUI("tab2"))
)

server <- function(input, output, session) {
  monthFeedbackServer("tab1", reactive(input$month))
  birthstoneServer("tab2", reactive(input$month))
}

shinyApp(ui, server)

```

В результате будет создано простое приложение с тремя вкладками, использующее модули для изоляции вкладок друг от друга. Это тестовое приложение, но вполне себе реальное. Единственное отличие от полноценного приложения состоит в простоте интерфейса и кода серверной функции.

Модульная структура

Перед преобразованием приложения в пакет давайте последуем совету из главы 19 и перенесем два модуля в отдельные файлы:

○ *R/monthFeedback.R*:

```
monthFeedbackUI <- function(id) {
  textOutput(NS(id, "feedback"))
}
monthFeedbackServer <- function(id, month) {
  stopifnot(is.reactive(month))

  moduleServer(id, function(input, output, session) {
    output$feedback <- renderText({
      if (month() == "October") {
        "You picked a great month!"
      } else {
        "Eh, you could do better."
      }
    })
  })
}
```

○ *R/birthstone.R*:

```
birthstoneUI <- function(id) {
  p(
    "The birthstone for ", textOutput(NS(id, "month"), inline = TRUE),
    " is ", textOutput(NS(id, "stone"), inline = TRUE)
  )
}
birthstoneServer <- function(id, month) {
  stopifnot(is.reactive(month))

  moduleServer(id, function(input, output, session) {
    stone <- reactive(stones$stone[stones$month == month()])
    output$month <- renderText(month())
    output$stone <- renderText(stone())
  })
}
```

В результате в файле *app.R* останется следующий код:

```
library(shiny)

stones <- vroom::vroom("birthstones.csv")
#> Rows: 12
```

```
#> Columns: 2
#> Delimiter: ","
#> chr [2]: month, stone
#>
#> Use `spec()` to retrieve the guessed column specification
#> Pass a specification to the `col_types` argument to quiet this message
months <- c(
  "January", "February", "March", "April", "May", "June",
  "July", "August", "September", "October", "November", "December"
)

ui <- navbarPage(
  "Sample app",
  tabPanel("Pick a month",
    selectInput("month", "What's your favourite month?", choices = months)
  ),
  tabPanel("Feedback", monthFeedbackUI("tab1")),
  tabPanel("Birthstone", birthstoneUI("tab2"))
)

server <- function(input, output, session) {
  monthFeedbackServer("tab1", reactive(input$month))
  birthstoneServer("tab2", reactive(input$month))
}

shinyApp(ui, server)
```

Выносить модули в отдельные файлы очень полезно хотя бы потому, что так гораздо лучше видна общая структура приложения. Когда мне нужно глубже разобраться в работе конкретного модуля, я просто открываю соответствующий файл.

Пакет

Теперь можно преобразовать приложение в пакет. Для начала запустим функцию `usethis::use_description()`, в результате чего будет создан файл *DESCRIPTION*. Затем переместим файл *app.R* в директорию *R/app.R* и заключим вызов `shinyApp()` в функцию:

```
library(shiny)

monthApp <- function(...) {
  stones <- vroom::vroom("birthstones.csv")
  months <- c(
    "January", "February", "March", "April", "May", "June",
    "July", "August", "September", "October", "November", "December"
  )

  ui <- navbarPage(
    "Sample app",
    tabPanel("Pick a month",
      selectInput("month", "What's your favourite month?", choices = months)
    ),

```

```

    tabPanel("Feedback", monthFeedbackUI("tab1")),
    tabPanel("Birthstone", birthstoneUI("tab2"))
  )

  server <- function(input, output, session) {
    monthFeedbackServer("tab1", reactive(input$month))
    birthstoneServer("tab2", reactive(input$month))
  }
  shinyApp(ui, server, ...)
}

```

В качестве дополнительной опции я преобразовал файл *birthstones.csv* в набор данных пакета, воспользовавшись функцией `usethis::use_data("birthstones")`. В результате был создан файл *data/birthstones.rda*, который будет загружаться автоматически при загрузке пакета. Теперь я могу удалить файл *birthstones.csv* и избавиться от строки чтения из него: `stones <- vroom::vroom("birthstones.csv")`.

Итоговый проект можно загрузить с GitHub по адресу <https://github.com/hadley/monthApp>.

ПРЕИМУЩЕСТВА

Зачем же нужно все это делать? Главным преимуществом преобразования приложения в пакет является облегчение работы с ним в отношении загрузки всего кода и перезапуска приложения. Кроме того, в этом случае упрощается процесс переноса кода между приложениями и появляется возможность поделиться своим приложением с другими.

Рабочий процесс

Упаковка кода приложения в пакет открывает вам новые возможности в плане рабочего процесса, включая:

- загрузку всего кода при помощи сочетания клавиш **Cmd/Ctrl+Shift+L**. В результате будет вызвана функция `devtools::load_all()`, что приведет к автоматическому сохранению всех файлов, чтению всех файлов из директории *R/* при помощи функции `source()`, загрузке всех наборов данных из папки *data/* и помещению курсора в консоль;
- перезапуск приложения при помощи функции `myApp()`.

С увеличением размера и сложности приложения полезно будет также узнать про следующие важные сочетания клавиш:

- **Ctrl/Cmd+.** (точка) позволяет выполнять нечеткий поиск по файлам и функциям. Напечатайте в открывшемся окне первые несколько букв названия файла или функции, которую хотите найти, выберите нужный вариант при помощи курсора и нажмите на клавишу **Enter**. Это поможет вам легко и быстро выполнять навигацию по приложению, не отрывая рук от клавиатуры;

- когда курсор находится на имени функции, клавиша **F2** позволит перейти к ее определению.

Если вы часто разрабатываете пакеты, вам наверняка хотелось бы автоматически загружать пакет *usethis*, чтобы можно было, например, просто писать `use_description()` вместо `usethis::use_description()`. Это можно сделать, добавив следующие строки в ваш *.Rprofile*. В этом файле содержится код на языке R, который выполняется при запуске R, так что это отличный способ настроить под себя окружение разработки:

```
if (interactive()) {
  require(usethis, quietly = TRUE)
}
```

Для редактирования этого файла проще всего будет вызвать функцию `usethis::edit_r_profile()`.

Совместное использование

Поскольку ваше приложение теперь заключено в функцию, можно легко объединить в одном пакете сразу несколько приложений. Таким образом, будет достаточно просто делиться кодом и данными между приложениями, а это является огромным преимуществом при наличии сразу нескольких приложений для решения схожих задач.

Кроме того, пакеты помогают делиться приложениями и с внешним миром. Службы *shinyapps.io* (<https://www.shinyapps.io>) и *RStudio Connect* (<https://www.rstudio.com/products/connect>) давно и активно используются с целью поделиться приложениями с людьми, незнакомыми с языком R. Но иногда вам необходимо дать доступ к вашему приложению коллегам, работающим с R. Например, вместо того чтобы пользователь загружал свой набор данных, вы хотите предоставить ему функцию, которую он будет вызывать со своим датафреймом. Следующее простое приложение позволяет пользователю R передать функции свой датафрейм для анализа.

```
dataSummaryApp <- function(df) {
  ui <- fluidPage(
    selectInput("var", "Variable", choices = names(df)),
    verbatimTextOutput("summary")
  )

  server <- function(input, output, session) {
    output$summary <- renderPrint({
      summary(df[[input$var]])
    })
  }

  shinyApp(ui, server)
}
```

На этой идее построен инструмент *RStudio Gadgets* (<https://shiny.rstudio.com/articles/gadgets.html>). По сути, это приложения Shiny, предполагающие

добавление нового интерфейса пользователя в оболочку разработки RStudio. Вы даже можете писать гаджеты, генерирующие код. Это позволит выполнять задачи в интерактивном режиме, а гаджет будет генерировать соответствующий код и сохранять его обратно в открытый файл.

Дополнительные шаги

Помимо описанных выше преимуществ, вы также можете легко развертывать свои приложения-пакеты и преобразовывать их в «настоящие» пакеты.

Развертывание приложения-пакета

Если вы хотите *развернуть* (deploy) приложение в службе *RStudio Connect* или *Shiny*¹, вам необходимо выполнить два дополнительных действия:

- файл *app.R* должен сообщать серверу развертывания, как запускать ваше приложение. Легче всего можно загрузить код с помощью пакета *pkgload*:

```
pkgload::load_all(".")
myApp()
```

С другими техниками можно ознакомиться в главе 13 книги *Engineering Shiny* по адресу <https://engineering-shiny.org/deploy.html>;

- обычно при развертывании приложения пакет *rsconnect* автоматически определяет все пакеты, используемые в коде. Но теперь, когда у вас есть файл *DESCRIPTION*, от вас требуется их явное перечисление. Самый простой способ сделать это – вызвать функцию *usethis::use_package()*. Начать следует с пакетов *shiny* и *pkgload*:

```
usethis::use_package("shiny")
usethis::use_package("pkgload")
```

Это не потребует от вас большого труда, а в качестве бонуса вы получите полный список пакетов, используемых вашим приложением.

После этого, когда будете готовы развернуть обновленную версию приложения, вы можете вызвать функцию *rsconnect::deployApp()*.

R CMD check

Минимально пакет должен содержать директорию *R/*, файл *DESCRIPTION* и функцию для запуска вашего приложения. Как вы уже видели, преобразо-

¹ Полагаю, для большинства других способов развертывания приложений Shiny это также подойдет, поскольку *app.R* является наиболее распространенным способом структурирования приложений.

вание приложения в пакет помогает добавить определенной легкости в дальнейшей разработке за счет использования дополнительных полезных инструментов. Но что представляет собой «настоящее» приложение? Для меня оно в первую очередь связано с большими усилиями на прохождение теста *R CMD check*. Тест *R CMD check* включает в себя автоматизированную систему, проверяющую пакет на наличие распространенных ошибок. В RStudio тест *R CMD check* можно запустить, нажав сочетание клавиш **Cmd/Ctrl+Shift+E**.

Я бы не рекомендовал вам запускать этот тест в своем первом, втором и даже третьем пакете. Вам лучше будет поближе познакомиться и освоиться с базовой структурой пакетов, прежде чем двигаться дальше. Прохождение теста я обычно оставляю для важных приложений, особенно для тех, которые будут проходить этап развертывания. Пройти *R CMD check* бывает очень и очень непросто, а выгоды от этого в краткосрочной перспективе будет не так много. В то же время на дистанции это может уберечь вас от большого количества проблем, и поскольку прохождение *R CMD check* подтверждает соответствие пакета определенному стандарту, с которым знакомы разработчики R, сторонним программистам будет проще вносить изменения в ваше приложение.

Перед созданием своего первого полноценного пакета я бы настоятельно рекомендовал вам прочитать главу *The Whole Game* из книги *R Packages* по адресу <https://r-pkgs.org/whole-game.html>: из нее вы сможете наиболее полно узнать о структуре пакета и сопутствующих процедурах. После этого, когда будете готовы к созданию пакета, следуйте приведенным ниже советам по прохождению теста *R CMD check*:

- исключите любые вызовы функций `library()` и `require()` в коде, заменив их явными объявлениями в файле *DESCRIPTION*. Чтобы добавить в файл с описанием строку с нужным вам пакетом, выполните инструкцию `usethis::use_package("name")`¹. После этого вам нужно будет решить, хотите ли вы обращаться к каждой функции явно с использованием символов `::` или использовать инструкцию `@importFrom packageName functionName` для объявления импорта в отдельном месте. Как минимум вы должны применить инструкцию `usethis::use_package("shiny")`, а для приложений Shiny я рекомендую использовать `@import shiny`, чтобы сделать легкодоступными все функции в пакете Shiny (в общем случае использование инструкции `@import` не считается хорошей практикой, но в данной ситуации это вполне оправданно);
- выберите лицензию и примените соответствующую функцию `use_license_`. Для проприетарного кода вы можете использовать функцию `usethis::use_proprietary_license()`. За подробностями по этому вопросу можете обратиться к главе 9 книги *R Packages* по адресу <https://r-pkgs.org/license.html>;

¹ Разница между значениями аргумента `type` (`Imports` или `Suggests`) не так важна при разработке приложений-пакетов. Если вы хотите знать разницу, то в большинстве случаев значение `Imports` стоит использовать для пакетов, которые должны присутствовать на машине развертывания (чтобы приложение работало), а `Suggests` – для пакетов, которые должны быть на машине разработки (чтобы можно было разрабатывать приложение).

- добавьте файл *app.R* в *.Rbuildignore* при помощи инструкции *usethis::use_build_ignore('app.R')* или ей подобной;
- если ваше приложение содержит в себе небольшие наборы данных, перенесите их в директорию *data* или *inst/extdata*. Функцию *usethis::use_data()* мы уже обсуждали ранее; в качестве альтернативы вы можете разместить данные в папке *inst/ext* и загружать их при помощи функции *read.csv(system.file("extdata", "mydata.csv", package = "myApp"))* или подобной ей;
- вы также можете изменить файл *app.R* для использования пакета. Для этого ваш пакет должен располагаться в месте, доступном для вашей машины развертывания. Для публичных работ это могут быть пакеты с CRAN или GitHub, а для частных можно использовать инструменты *RStudio Package Manager* (<https://www.rstudio.com/products/package-manager>) или *drat* (<https://github.com/eddelbuettel/drat>):

```
myApp::myApp()
```

ЗАКЛЮЧЕНИЕ

В данной главе вы познакомились с тонкостями разработки пакетов. Кому-то может показаться, что пакеты – это обязательно что-то большое и сложное вроде *ggplot2* или *shiny*, но на самом деле они могут быть и довольно простыми. Фактически все, что необходимо любому проекту, чтобы считаться пакетом, – это файлы, расположенные в директории *R*, и наличие файла *DESCRIPTION*. По сути, пакет представляет собой набор требований, удовлетворение которому открывает возможность использовать в работе полезные инструменты. При чтении главы вы научились преобразовывать приложение в пакет, а также узнали доводы в пользу такого преобразования. А в следующей главе мы поговорим о самой важной причине представления приложений в виде пакетов – возможности облегчить процесс тестирования.

Глава 21

Тестирование

Для простых приложений достаточно запомнить специфику их работы, чтобы при внесении изменений в будущем не нарушить то, что было сделано до этого. Но с ростом сложности приложения становится практически невозможно удерживать все нюансы его работы в голове. *Тестирование* (testing) представляет собой способ захвата желаемого поведения приложения, чтобы в будущем можно было автоматически проверять его на работоспособность. Превратить существующие у вас неформальные тесты в код на первых порах бывает очень нелегко, поскольку приходится переводить на машинный язык все нажатия кнопок и щелчки мышью, но в перспективе эта работа очень важна, ведь с ее помощью вы сможете очень быстро запускать необходимые проверки.

Мы будем выполнять автоматизированные тесты при помощи пакета *testthat* (<https://testthat.r-lib.org>). Использование *testthat* требует предварительно преобразования приложения в пакет, но, как мы уже говорили в главе 20, это не займет у вас много времени, а пользы будет достаточно.

Тестирование с использованием пакета *testthat* производится следующим образом:

```
test_that("as.vector() strips names", {  
  x <- c(a = 1, b = 2)  
  expect_equal(as.vector(x), c(1, 2))  
})
```

Совсем скоро мы подробно разберем синтаксис этой конструкции. Сейчас же достаточно будет отметить, что тестирование начинается с объявления намерения "as.vector() strips names" (as.vector() обрезает имена), после чего идет код на языке R для генерирования тестовых данных. В заключение тестовые данные сравниваются с эталонными с помощью *ожидания* (expectation) – функции, начинающейся с *expect_*. Первым аргументом в нее передается код для запуска, а вторым – ожидаемый результат. В данном случае мы проверяем на равенство вывод функции as.vector(x) и вектор c(1, 2).

В этой главе мы поговорим о следующих четырех уровнях тестирования:

- начнем с тестирования неактивных функций. Это поможет вам освоить базовый рабочий процесс тестирования и проверить поведение кода, извлеченного из интерфейса или серверной функции приложения. Речь здесь пойдет о том же самом типе тестирования, которое используется при написании пакетов, так что дополнительную ин-

формацию по этой теме вы сможете найти в главе книги *R Packages*, посвященной тестированию, по адресу <https://r-pkgs.org/tests.html>;

- после этого вы узнаете, как можно протестировать реактивные потоки внутри серверной функции. Вы будете устанавливать значения элементов ввода и проверять выходные значения реактивов и элементов вывода на соответствие вашим ожиданиям;
- далее вы научитесь тестировать код Shiny, использующий JavaScript (например, функции семейства `update`), запуская приложение в фоновом браузере. Этот вид симуляции отличается высокой точностью, поскольку запуск происходит в настоящем браузере. Из минусов здесь стоит отметить низкую скорость проверки и невозможность так же легко, как прежде, заглядывать в приложение;
- наконец, мы протестируем визуальные элементы приложения путем сохранения скриншотов. Это необходимо для проверки макета приложения, CSS, графиков и виджетов HTML. В то же время это довольно ненадежный способ, поскольку скриншоты могут меняться по самым разным причинам. Это означает, что в процессе тестирования потребуется вмешательство человека для определения приемлемости изменений. По этой причине такой вид проверки можно назвать наиболее трудозатратным.

Перечисленные выше уровни тестирования образуют естественную иерархию, поскольку каждая следующая техника представляет более полное моделирование пользовательского взаимодействия с приложением. Из недостатков увеличения точности моделирования можно отметить уменьшение скорости тестирования и снижение его надежности по причине вмешательства все большего количества внешних факторов. Вы должны всегда стремиться работать на минимально допустимом уровне тестирования, чтобы проверки выполнялись достаточно быстро и надежно. Со временем это начнет оказывать влияние на написание вами кода: зная, какой код легче поддается тестированию, вы естественным образом будете склоняться к более простым реализациям. Рассказывая об этих уровнях тестирования, я параллельно буду давать советы по выполнению проверок и затрону тему, касающуюся общей философии тестирования. Давайте приступим:

```
library(shiny)
library(testthat) # >= 3.0.0
library(shinytest)
```

ТЕСТИРОВАНИЕ ФУНКЦИЙ

Наиболее просто выполняется тестирование фрагментов кода, меньше всего связанных с Shiny, т. е. функций, извлеченных из интерфейса и серверной логики приложения, о которых мы говорили в главе 18. Начнем с описания общего подхода к тестированию нереактивных функций и заодно посмотрим на базовую структуру выполнения операций при помощи пакета *testthat*.

Базовая структура

Тестирование включает в себя три уровня:

- **файл.** Все файлы для тестирования должны располагаться в папке *tests/testthat* и соответствовать файлам в директории *R/*. Например, код из модуля *R/module.R* должен тестироваться при помощи файла *tests/testthat/test-module.R*. К счастью, вам совсем не обязательно запоминать это соглашение: достаточно использовать функцию `usethis::use_test()` для создания или поиска тестового файла, соответствующего открытому в данный момент файлу из папки *R*;
- **тест.** Каждый файл разбивается на тесты, то есть на вызовы функции `test_that()`. Один тест, как правило, проверяет одно свойство функции. Достаточно трудно объяснить, что под этим имеется в виду, но вы должны стремиться к тому, чтобы вам было легко описать конкретный тест в первом аргументе функции `test_that()`;
- **ожидание.** Каждый тест содержит одно или несколько ожиданий, представляющих собой функции, начинающиеся с `expect_`. В них вы определяете желаемые действия, касается ли это возвращаемого значения, генерирования ошибки или чего-то еще. В данной главе мы поговорим о наиболее важных ожиданиях применительно к приложениям Shiny, а полный список функций ожидания вы можете посмотреть на сайте пакета *testthat* по адресу <https://testthat.r-lib.org/reference/index.html#section-expectations>.

Искусство тестирования заключается в написании тестов, полностью определяющих ожидаемое поведение функции и не зависящих от параметров, которые могут измениться в будущем.

Основной рабочий процесс

Теперь, когда вы понимаете базовые принципы процесса тестирования, давайте погрузимся в практические примеры. Начнем с простого приложения, которое мы обсуждали в главе 18. Для удобства я извлек часть кода серверной функции в отдельную функцию, которую назвал `load_file()`:

```
load_file <- function(name, path) {
  ext <- tools::file_ext(name)
  switch(ext,
    csv = vroom::vroom(path, delim = ",", col_types = list()),
    tsv = vroom::vroom(path, delim = "\t", col_types = list()),
    validate("Invalid file; Please upload a .csv or .tsv file")
  )
}
```

Представим, что этот код находится в файле *R/load.R*, так что тесты для него должны располагаться в файле *tests/testthat/test-load.R*. Самый

простой способ создать этот файл – вызвать функцию `usethis::use_test()` с файлом `load.R`¹.

В данной функции я хочу провести следующие проверки: может ли она загружать файлы `CSV` и `TSV` и выдает ли ошибку для других типов файлов. Для выполнения этих тестов мне понадобятся проверочные файлы, которые я положу во временную папку сессии, чтобы они автоматически очищались после запуска тестов. Далее я добавлю три ожидания: два из них будут проверять эквивалентность загруженного файла и исходных данных, а третье – получение ошибки:

```
test_that("load_file() handles all input types", {
  # Создание тестовых данных
  df <- tibble::tibble(x = 1, y = 2)
  path_csv <- tempfile()
  path_tsv <- tempfile()
  write.csv(df, path_csv, row.names = FALSE)
  write.table(df, path_tsv, sep = "\t", row.names = FALSE)
  expect_equal(load_file("test.csv", path_csv), df)
  expect_equal(load_file("test.tsv", path_tsv), df)
  expect_error(load_file("blah", path_csv), "Invalid file")
})
#> Test passed
```

Существует четыре способа запуска этого теста:

- в момент разработки я каждую строку запускаю интерактивно в консоли. Если функция ожидания возвращает ошибку, я сразу ее правлю;
- после окончания разработки я запускаю весь тестовый блок. Если тест будет успешно пройден, я получу сообщение *Test passed*. В противном случае меня оповестят, что именно пошло не так;
- если у меня есть несколько тестов, я могу запустить их все для текущего файла² при помощи функции `devtools::test_file()`. Поскольку я делаю это довольно часто, то даже повесил сочетание клавиш на эту функцию, чтобы максимально облегчить себе жизнь. Скоро я покажу вам, как это можно сделать;
- время от времени я прогоняю все тесты для всего пакета в целом с помощью функции `devtools::test()`. Так я проверяю, не сломалось ли что-то за пределами текущего файла.

Основные функции ожидания

Существует две основные функции ожидания, которыми вы будете пользоваться при выполнении тестов в большинстве случаев: `expect_equal()` и `expect_error()`. Как и в случае со всеми функциями ожидания, первым ар-

¹ Если вы не используете RStudio, вам придется передать функции `use_test()` имя файла, например `this::use_test("load")`.

² Как и функция `usethis::use_test()`, этот прием работает только при использовании RStudio.

гументом мы передаем блок кода для проверки, а вторым – ожидаемый исход, представляющий собой плановое значение в случае с функцией `expect_equal()` и ожидаемый текст ошибки для `expect_error()`.

Чтобы лучше понять, как работают эти функции, полезно бывает вызывать их напрямую, не в составе теста.

При вызове функции `expect_equal()` помните, что вы не обязаны тестировать целый объект, обычно лучше проверять только интересующий вас компонент:

```
complicated_object <- list(
  x = list(mtcars, iris),
  y = 10
)
expect_equal(complicated_object$y, 10)
```

Помимо этого, существует целый ряд вспомогательных функций для частных случаев проверки равенства, которые помогут вам сэкономить время:

- функции `expect_true(x)` и `expect_false(x)` эквивалентны вызову `expect_equal(x, TRUE)` и `expect_equal(x, FALSE)` соответственно;
- функция `expect_null(x)` эквивалентна записи `expect_equal(x, NULL)`;
- функция `expect_named(x, c('a', 'b', 'c'))` эквивалентна `expect_equal(names(x), c("a", "b", "c"))`, но у нее есть дополнительные опции `ignore.order` и `ignore.case`;
- функция `expect_length(x, 10)` эквивалентна вызову `expect_equal(length(x), 10)`.

Есть также особые разновидности функции `expect_equal()` для работы с векторами:

- функция `expect_setequal(x, y)` проверяет, что каждое значение из вектора `x` присутствует в векторе `y` и наоборот;
- функция `expect_mapequal(x, y)` проверяет, что векторы `x` и `y` имеют одинаковые имена и `x[names(y)]` эквивалентно `y`.

Часто бывает необходимо проверить, что код генерирует ошибку, и в этих случаях вам поможет функция `expect_error()`:

```
expect_error("Hi!")
#> Error: "Hi!" did not throw the expected error.
expect_error(stop("Bye"))
```

Обратите внимание, что функция `expect_error()` может принимать второй аргумент в виде регулярного выражения, служащего для поиска короткого фрагмента текста, соответствующего ожидаемой ошибке и с большой вероятностью не соответствующего ошибке, которую вы не ожидаете:

```
f <- function() {
  stop("Calculation failed [location 1]")
}

expect_error(f(), "Calculation failed [location 1]")
#> Error in f(): Calculation failed [location 1]
expect_error(f(), "Calculation failed \\[[location 1]\\]")
```

Но все же лучше будет выбрать короткий текст для проверки соответствия:

```
expect_error(f(), "Calculation failed")
```

Можно также использовать функцию `expect_snapshot()`, о которой мы поговорим позже. Вместе с функцией `expect_error()` еще можно применять ее вариации `expect_warning()` и `expect_message()`, позволяющие отследить появление предупреждений и сообщений таким же образом, как вы проверяете ошибки. При тестировании приложений Shiny эти функции применяются нечасто, чего не скажешь о процессе проверки пакетов.

Функции интерфейса пользователя

Вы можете использовать те же идеи для тестирования функций, извлеченных из кода интерфейса пользователя. Правда, здесь вам понадобятся другие функции ожидания. Вводить весь код разметки HTML вручную было бы очень утомительно, поэтому в данном случае применяется *Снимочное тестирование* (snapshot test)¹. Снимочное ожидание отличается от остальных главным образом тем, что ожидаемый результат сохраняется в отдельном файле-снимке, а не непосредственно в коде. Такой вид тестирования наиболее полезен при разработке масштабных систем со сложным интерфейсом пользователя, что не касается большинства приложений Shiny. Поэтому я коротко расскажу ключевые идеи, лежащие в основе этой проверки, после чего дам ссылку на дополнительную литературу.

Взгляните на эту функцию пользовательского интерфейса, которую мы написали ранее:

```
sliderInput01 <- function(id) {
  sliderInput(id, label = id, min = 0, max = 1, value = 0.5, step = 0.1)
}

cat(as.character(sliderInput01("x")))
#> <div class="form-group shiny-input-container">
#>   <label class="control-label" id="x-label" for="x">x</label>
#>   <input class="js-range-slider" id="x" data-skin="shiny" data-min="0"
#>     data-max="1" data-from="0.5" data-step="0.1" data-grid="true"
#>     data-grid-num="10" data-grid-snap="false" data-prettify-separator=","
#>     data-prettify-enabled="true" data-keyboard="true" data-data-type="number"/>
#> </div>
```

Как проверить, что вывод функции соответствует нашим ожиданиям? Мы могли бы использовать уже знакомую нам функцию `expect_equal()` следующим образом:

```
test_that("shinyInput01() creates expected HTML", {
  expect_equal(
```

¹ Снимочное тестирование требует наличия третьей версии пакета *testthat*. Новые пакеты будут автоматически использовать *testthat 3e* (<https://testthat.r-lib.org/articles/third-edition.html>), тогда как старые придется обновлять вручную.

```

as.character(sliderInput01("x")),
"<div class=\"form-group shiny-input-container\">\n
  <label class=\"control-label\" id=\"x-label\" for=\"x\">x</label>\n
  <input class=\"js-range-slider\" id=\"x\" data-skin=\"shiny\" data-min=\"0\"
    data-max=\"1\" data-from=\"0.5\" data-step=\"0.1\" data-grid=\"true\"
    data-grid-num=\"10\" data-grid-snap=\"false\" data-prettify-separator=\", \"
    data-prettify-enabled=\"true\" data-keyboard=\"true\"
    data-data-type=\"number\"/>\n
</div>\"
)
})
#> Test passed

```

Однако присутствие большого количества кавычек и переносов строки вынудило нас использовать массу экранирующих символов, что мешает разглядеть не только то, что мы на самом деле желаем увидеть в качестве результата, но и что вообще происходит.

Ключевая идея снимочного тестирования состоит в хранении ожидаемого результата в отдельном файле – это позволяет исключить из тестового кода объемные данные и не беспокоиться об экранирующих символах. Так можно использовать функцию `expect_snapshot()` для проверки вывода, отображаемого на консоли:

```

test_that("shinyInput01() creates expected HTML", {
  expect_snapshot(sliderInput01("x"))
})

```

Основным отличием `expect_snapshot()` от других функций ожидания является отсутствие второго аргумента, описывающего результат, который вы ожидаете увидеть. Вместо этого результат сохраняется в виде отдельного файла *.md*. К примеру, если ваш код располагается в файле *R/slider.R*, а тест – в *tests/testthat/test-slider.R*, то снимок будет сохранен в файле *tests/testthat/_snaps/slider.md*. При первом запуске теста функция `expect_snapshot()` автоматически создаст вывод, который будет выглядеть так:

```
# shinyInput01() creates expected HTML
```

Code

```
sliderInput01("x")
```

Output

```

<div class="form-group shiny-input-container">
  <label class="control-label" id="x-label" for="x">x</label>
  <input class="js-range-slider" id="x" data-skin="shiny" data-min="0"
    data-max="1" data-from="0.5" data-step="0.1" data-grid="true"
    data-grid-num="10" data-grid-snap="false" data-prettify-separator=", "
    data-prettify-enabled="true" data-keyboard="true" data-data-type="number"/>
</div>

```

Если позже вывод изменится, тест будет провален. Вам придется либо исправить ошибку, приводящую к этому, либо – в случае, если изменение было намеренным, – обновить снимок, вызвав функцию `testthat::snapshot_accept()`.

Перед тем как использовать этот тест, необходимо внимательно присмотреться к выводу. Что вы на самом деле тестируете? Если вам нужно понять, как элементы ввода генерируют вывод, то вы заметите, что по большей части вывод был сгенерирован Shiny, и лишь незначительная его часть основывается на вашем коде. Это означает, что данный тест не будет особенно полезным: если вывод изменится, это, скорее всего, произойдет по причине изменений в Shiny, а не в вашем коде. Именно это делает тест ненадежным. Если он провалится, это вряд ли произойдет в результате вашей ошибки, и исправить ситуацию вы, вероятно, не сможете.

Подробнее почитать о снимочном тестировании можно в статье по адресу <https://testthat.r-lib.org/articles/snapshotting.html>.

РАБОЧИЙ ПРОЦЕСС

Прежде чем приступить к тестированию функций, использующих реактивы или JavaScript, сделаем небольшое отступление и поговорим об организации рабочего процесса.

Покрытие кода

В процессе тестирования очень важно убедиться в том, что ваши проверки работают там, где вы планировали. Сделать это можно при помощи механизма *покрытия кода* (code coverage), запускающего тесты и отслеживающего строки кода, к которым они применяются. После этого вы можете посмотреть на результаты, чтобы понять, какие строки кода не были затронуты в процессе тестирования. На основании увиденного вы можете решить, попали ли наиболее уязвимые и сложные фрагменты кода в область покрытия. Но это не отменяет важности размышлений о самом коде – у вас может быть 100-процентное покрытие кода, но при этом присутствовать ошибки. Сам же инструмент определения покрытия кода несет в себе большую пользу, поскольку позволяет понять, какие участки в вашем коде наиболее важны, особенно когда речь идет о сложных приложениях с вложенным кодом.

Мы не будем говорить об этом механизме подробно, но я очень рекомендую вам поэкспериментировать с функциями `devtools::test_coverage()` и `devtools::test_coverage_file()`. Главное, что нужно знать, – зеленым помечаются протестированные строки, а красным – непротестированные.

Процесс покрытия кода включает в себя следующие этапы.

1. Применение функции `test_coverage()` или `test_coverage_file()` для проверки того, какие строки не были протестированы.
2. Разработка отдельных тестов для не покрытых на предыдущем шаге строк.
3. Повторный запуск, пока все важные строки кода не будут покрыты. Зачастую не имеет смысла стремиться к покрытию всех без исключения строк кода, достаточно убедиться, что выполняется тестирование наиболее важных фрагментов.

Функции покрытия кода также работают с инструментами тестирования кода, включающего реактивные элементы и (до определенной степени) JavaScript, так что обладать связанным с этим навыком очень важно.

Сочетания клавиш

Если вы последовали нашим советам из главы 20, значит, уже можете запускать процесс тестирования, просто вводя в консоли инструкции `test()` или `test_file()`. Но вы будете так часто выполнять проверку кода, что желательно сразу обзавестись для этого удобными сочетаниями клавиш. В RStudio по умолчанию уже настроено одно удобное сочетание: нажатие клавиш **Cmd/Ctrl+Shift+T** приведет к запуску функции `devtools::test()`. Я бы рекомендовал вам дополнить его следующими полезными сочетаниями:

- **Cmd/Ctrl+T** для `devtools::test_file()`;
- **Cmd/Ctrl+Shift+R** для `devtools::test_coverage()`;
- **Cmd/Ctrl+R** для `devtools::test_coverage_file()`.

Разумеется, вы вольны выбирать собственные сочетания клавиш, но предложенный мной вариант подразумевает логичную структуру, заключающуюся в том, что сочетания с использованием клавиши **Shift** применяются ко всему пакету, а без **Shift** – только к текущему файлу.

В табл. 21.1 показаны мои сочетания клавиш на Mac.

Таблица 21.1. Сочетания клавиш на Mac

Покрытие кода для файла	Cmd+R
Покрытие кода для пакета	Shift+Cmd+R
Запуск теста для файла	Cmd+T

Резюме по рабочему процессу

Давайте подведем итог по тому, о чем мы говорили в данном разделе:

- в файле R вызовите функцию `usethis::use_test()` для создания тестового файла (при первом запуске) или доступа к нему (если он уже существует);
- напишите код и тесты. Нажмите сочетание клавиш **Cmd/Ctrl+T** для запуска тестов и вывода результатов в консоли. Повторяйте этот шаг, пока в этом будет необходимость;
- если в процессе тестирования вы обнаружили новую ошибку, начните локализовывать ее путем сокращения объема кода. Это зачастую поможет вам понять, где кроется ошибка, а наличие теста не позволит поддаться заблуждению о том, что вы ее исправили, если это не так;
- нажмите сочетание клавиш **Cmd/Ctrl+R**, чтобы убедиться в том, что вы на самом деле тестируете то, что собирались;
- нажмите клавиши **Cmd/Ctrl+Shift+T** для проверки того, что вы в процессе тестирования не сломали что-то еще.

ТЕСТИРОВАНИЕ РЕАКТИВОВ

Теперь, когда вы освоили процесс тестирования обычного нереактивного кода, пришло время перейти к вещам, специфичным для фреймворка Shiny. И первое, о чем мы поговорим, – это проверка реактивного кода. Как вы уже знаете, вы не сможете запустить реактивный код интерактивно:

```
x <- reactive(input$y + input$z)
x()
#> Error: Operation not allowed without an active reactive context.
#> * You tried to do something that can only be done from inside a reactive
#> consumer.
```

Вы могли бы задуматься об использовании вспомогательной функции `activeConsole()`, о которой мы уже говорили в главе 15. К сожалению, такое моделирование реактивности полагается на интерактивную консоль, так что для проведения тестирования этот вариант не годится.

И дело здесь не только в появлении ошибки – даже если бы она не возникла, значения `input$y` и `input$z` не были бы определены. Чтобы посмотреть, как это работает, давайте начнем с простого приложения с тремя элементами ввода, одним элементом вывода и тремя реактивами:

```
ui <- fluidPage(
  numericInput("x", "x", 0),
  numericInput("y", "y", 1),
  numericInput("z", "z", 2),
  textOutput("out")
)
server <- function(input, output, session) {
  xy <- reactive(input$x - input$y)
  yz <- reactive(input$z + input$y)
  xyz <- reactive(xy() * yz())
  output$out <- renderText(paste0("Result: ", xyz()))
}
```

Для тестирования этого кода мы будем использовать функцию `testServer()`. Эта функция принимает два аргумента: серверную функцию и код для выполнения. Код запускается в особом окружении, внутри серверной функции, что открывает вам доступ к элементам вывода, реактивам и специальному объекту `session`, позволяющему моделировать взаимодействие с пользователем. В основном вы будете использовать этот объект для вызова функции `session$setInputs()`, позволяющей устанавливать значения элементов ввода, как если бы вы работали с приложением в браузере:

```
testServer(server, {
  session$setInputs(x = 1, y = 1, z = 1)
  print(xy())
  print(output$out)
})
#> [1] 0
#> [1] "Result: 0"
```

Вы можете использовать функцию `testServer()` для перехода в интерактивное окружение, поддерживающее реактивность: `testServer(myApp(), browser())`.

Обратите внимание, что мы выполняем тестирование только серверной функции, компонента интерфейса пользователя мы никак не касаемся. Это можно понять, посмотрев на значения элементов ввода: в отличие от настоящего приложения Shiny, все элементы ввода изначально будут установлены в `NULL`, поскольку исходные значения записаны в интерфейсе пользователя. Мы вернемся к тестированию пользовательского интерфейса далее в этой главе.

```
testServer(server, {
  print(input$x)
})
#> NULL
```

Теперь, когда у вас есть способ запуска кода в реактивном окружении, вы можете совместить его со знаниями в области тестирования кода и получить нечто вроде этого:

```
test_that("reactives and output updates", {
  testServer(server, {
    session$setInputs(x = 1, y = 1, z = 1)
    expect_equal(xy(), 0)
    expect_equal(yz(), 2)
    expect_equal(output$out, "Result: 0")
  })
})
#> Test passed
```

Зная, как работает функция `testServer()`, вы сможете проводить тестирование реактивного кода так же легко, как и неактивного. Основная сложность в данном случае заключается в отладке проваленных тестов: вы не сможете пройти по ним построчно, как в случае с обычными тестами, так что вам придется вставлять вызовы функции `browser()` внутрь `testServer()` для возможности проводить интерактивные эксперименты с целью выявления проблемы.

Модули

Модули вы можете тестировать так же, как и обычные функции приложения, при этом вам должно быть понятно, что мы говорим только о серверной логике модуля. Давайте начнем с простого модуля, в котором используется три элемента вывода для отображения трех статистических показателей:

```
summaryUI <- function(id) {
  tagList(
    outputText(ns(id, "min")),
    outputText(ns(id, "mean")),
    outputText(ns(id, "max")),
  )
}
```

```

}
summaryServer <- function(id, var) {
  stopifnot(is.reactive(var))

  moduleServer(id, function(input, output, session) {
    range_val <- reactive(range(var(), na.rm = TRUE))
    output$min <- renderText(range_val()[[1]])
    output$max <- renderText(range_val()[[2]])
    output$mean <- renderText(mean(var()))
  })
}

```

Как и раньше, мы будем использовать функцию `testServer()`, но ее вызов в этом случае будет немного отличаться. Первым аргументом мы снова передадим серверную функцию (на этот раз это будет серверная функция модуля), а в качестве дополнительных аргументов будет выступать список с именем `args`, представляющий собой список параметров серверной функции модуля (параметр `id` не обязателен, функция `testServer()` подставит его автоматически). В результате получим следующий код для запуска:

```

x <- reactiveVal(1:10)
testServer(summaryServer, args = list(var = x), {
  print(range_val())
  print(output$min)
})
#> [1] 1 10
#> [1] "1"

```

Опять же, мы можем превратить эту функцию в автоматизированный тест, заключив ее в функцию `test_that()` и вызвав внутри нужные нам функции ожидания, начинающиеся с `expect_`. Ниже представлен тест для проверки того, как реагирует модуль на изменение реактивного ввода:

```

test_that("output updates when reactive input changes", {
  x <- reactiveVal()
  testServer(summaryServer, args = list(var = x), {
    x(1:10)
    session$flushReact()
    expect_equal(range_val(), c(1, 10))
    expect_equal(output$mean, "5.5")
    x(10:20)
    session$flushReact()
    expect_equal(range_val(), c(10, 20))
    expect_equal(output$min, "10")
  })
})
#> Test passed

```

Здесь показан один интересный и важный трюк: поскольку реактивное значение `x` было создано за пределами функции `testServer()`, его изменение не будет автоматически влиять на реактивный график, так что нам необходимо вызывать функцию `session$flushReact()` вручную.

Если ваш модуль возвращает значение (в виде реактива или списка реактивов), вы можете захватить его с помощью функции `session$getReturned()`. После этого вы можете проверить значение возвращенной переменной так же точно, как и любого другого реактива:

```
datasetServer <- function(id) {
  moduleServer(id, function(input, output, session) {
    reactive(get(input$dataset, "package:datasets"))
  })
}

test_that("can find dataset", {
  testServer(datasetServer, {
    dataset <- session$getReturned()
    session$setInputs(dataset = "mtcars")
    expect_equal(dataset(), mtcars)
    session$setInputs(dataset = "iris")
    expect_equal(dataset(), iris)
  })
})
#> Test passed
```

Надо ли нам выполнять проверку на случай, если `input$dataset` окажется не набором данных? В этом случае нет, поскольку мы знаем, что в интерфейсе модуля выбор был ограничен доступными значениями. Это не вполне очевидно при анализе одной серверной функции.

Ограничения

Функция `testServer()` имитирует выполнение приложения. Такие имитации очень важны, поскольку позволяют быстро протестировать реактивный код. В то же время они не являются полноценными по следующим причинам:

- в отличие от реального мира, время во время симуляций не идет. Так что в случае, если вы хотите проверить код, базирующийся на функциях `reactiveTimer()` или `invalidateLater()`, вам придется вручную запускать время при помощи функции `session$elapse(millis = 300)`;
- функция `testServer()` игнорирует интерфейс пользователя. Это означает, что элементы ввода не принимают значения по умолчанию, а скрипты JavaScript не запускаются. Но важнее то, что вы не можете тестировать функции семейства `update`, поскольку они в процессе выполнения посылают код JavaScript браузеру для имитации действий пользователя. Для проверки этого кода вам придется использовать показанную ниже технику.

ТЕСТИРОВАНИЕ JAVASCRIPT

Функция `testServer()` позволяет тестировать только полноценные приложения Shiny, так что код, полагающийся на работу в настоящем браузере, вы

проверить не сможете. В частности, это означает, что вам будет недоступно тестирование скриптов JavaScript. Вам это может показаться незначительным ограничением, поскольку мы не так много в этой книге говорили о JavaScript, но существует немало популярных функций Shiny, неявным образом использующих в своей работе JavaScript. К ним относятся, например:

- все функции семейства `update`, о которых мы говорили в главе 10;
- функции `showNotification()` / `removeNotification()`, которые мы упоминали в главе 8;
- функции `showModal()` / `hideModal()`, также упоминавшиеся в главе 8;
- функции `insertUI()` / `removeUI()` / `appendTab()` / `insertTab()` / `removeTab()`, о которых мы еще будем говорить далее в этой книге.

Для проверки работы всех этих функций вам необходимо запустить приложение Shiny в браузере. Конечно, вы можете сделать это, запустив функцию `runApp()`, но хотелось бы автоматизировать этот тип тестирования, чтобы вы могли выполнять его на постоянной основе. Мы сделаем это, воспользовавшись пакетом *shinytest* (<https://rstudio.github.io/shinytest>). Вы можете применять этот пакет так, как рекомендовано в инструкции на сайте, – автоматически генерируя тестовый код при использовании приложения, но поскольку мы уже знакомы с пакетом *testthat*, попробуем другой подход, – будем создавать тесты вручную.

Мы будем работать с объектом R6 из пакета *shinytest*, а именно с *ShinyDriver*. Создание нового экземпляра класса *ShinyDriver* запускает процесс R с вашим приложением Shiny в консольном (headless) браузере без графического интерфейса. В этом режиме браузер работает так же, как обычно, но без физического окна, с которым вы могли бы взаимодействовать, – все взаимодействие осуществляется в этом случае исключительно посредством кода. Главными недостатками такого подхода является его медлительность по сравнению с другими методами (потребуется минимум секунда даже для самого простого приложения) и то, что вы можете тестировать только внешнюю часть приложения, т. е. увидеть значения реактивных переменных вам будет сложнее.

Основные операции

Для демонстрации базовых методов этого вида тестирования давайте создадим простое приложение, приветствующее пользователя по имени, с кнопкой очистки поля ввода:

```
ui <- fluidPage(
  textInput("name", "What's your name"),
  textOutput("greeting"),
  actionButton("reset", "Reset")
)

server <- function(input, output, session) {
  output$greeting <- renderText({
    req(input$name)
```

```

    paste0("Hi ", input$name)
  })
  observeEvent(input$reset, updateTextInput(session, "name", value = ""))
}

```

Чтобы воспользоваться пакетом *shinytest*, запустим приложение при помощи инструкции `app <- ShinyDriver$new()`, взаимодействовать с ним будем посредством функции `app$setInputs()` и ей подобных, а получать значения, вызывая функцию `app$getValue()`, как показано в примере ниже:

```

app <- shinytest::ShinyDriver$new(shinyApp(ui, server))
app$setInputs(name = "Hadley")
app$getValue("greeting")
#> [1] "Hi Hadley"
app$click("reset")
app$getValue("greeting")
#> [1] ""

```

Каждое использование пакета *shinytest* начинается с создания экземпляра класса *ShinyDriver* с помощью вызова метода `ShinyDriver$new()`, принимающего на вход объект приложения *Shiny* или путь к нему. Метод возвращает объект R6, с которым вы можете взаимодействовать примерно как с объектом *session*, – посредством функции `app$setInputs()`: на вход она принимает набор пар имя-значение, устанавливает соответствующие элементы в браузере и ожидает, пока обновятся все реактивные элементы в приложении.

Различие заключается в том, что вам необходимо явно извлекать необходимые значения при помощи функции `app$getValue(name)`. В отличие от `testServer()`, вы не можете получить доступ к реактивам посредством *ShinyDriver*, поскольку этот объект видит только то, что видит пользователь приложения. Однако есть особая функция *Shiny* с именем `exportTestValues()`, способная создавать вывод, который доступен пакету *shinytest*, но не пользователю.

Существуют еще две важные функции, позволяющие программно взаимодействовать с приложением:

- функция `app$click(name)` выполняет нажатие на кнопку с именем `name`;
- функция `app$sendKeys(name, keys)` посылает нажатие клавиш объекту с именем `name`. Аргумент `keys` обычно представляет собой строку, как в примере `app$sendKeys(id, "Hi!")`. Также вы можете посылать специальные клавиши при помощи `webdriver::key`, например `app$sendKeys(id, c(webdriver::key$control, "x"))`. Обратите внимание, что все клавиш-модификаторы будут применяться ко всем последующим нажатиям клавиш, так что если вы хотите, чтобы одни клавиши нажимались с модификаторами, а другие – без, необходимо использовать множественные вызовы.

Используйте справку `?ShinyDriver` для просмотра дополнительной информации и практических советов.

Как обычно, определившись с последовательностью действий, вы можете превратить их в тест путем заключения в функцию `test_that()` и вызова соответствующих ожиданий:


```
test_that("can set and reset name", {
  app <- shinytest::ShinyDriver$new(shinyApp(ui, server))
  app$setInputs(name = "Hadley")
  expect_equal(app$getValue("greeting"), "Hi Hadley")

  app$click("reset")
  expect_equal(app$getValue("greeting"), "")
})
```

Фоновое приложение Shiny и браузер автоматически закроются, когда объект `app` будет удален и очищен из памяти. Если вы не в курсе, как происходит очистка «мусора», можете почитать об этом в соответствующей главе книги *Advanced R* по адресу <https://adv-r.hadley.nz/names-values.html#gc>.

Практический пример

Завершим изучение этой темы при помощи практического примера с совместным использованием функции `testServer()` и пакета *shinytest*. Мы применим уже известные вам переключатели с возможностью свободного ввода – похожий интерфейс мы использовали в главе 19:

```
ui <- fluidPage(
  radioButtons("fruit", "What's your favourite fruit?",
    choiceNames = list(
      "apple",
      "pear",
      textInput("other", label = NULL, placeholder = "Other")
    ),
    choiceValues = c("apple", "pear", "other")
  ),
  textOutput("value")
)

server <- function(input, output, session) {
  observeEvent(input$other, ignoreInit = TRUE, {
    updateRadioButtons(session, "fruit", selected = "other")
  })

  output$value <- renderText({
    if (input$fruit == "other") {
      req(input$other)
      input$other
    } else {
      input$fruit
    }
  })
}
```

В действительности вычисление здесь довольно простое. Мы могли бы извлечь выражение, заключенное в функции `renderText()`, в отдельную функцию, как показано ниже:

```
other_value <- function(fruit, other) {
  if (fruit == "other") {
    other
  } else {
    fruit
  }
}
```

Но я не думаю, что это того стоит, поскольку логика здесь довольно простая и не обобщается на другие ситуации. Полагаю, обособление этого кода в отдельный файл только затруднило бы понимание приложения.

Итак, начнем с базовой проверки реактивных потоков: получим ли мы правильный результат, установив элемент `fruit` в один из существующих вариантов? И что будет, если мы выберем пункт со свободным выбором и введем произвольный текст?

```
test_that("returns other value when primary is other", {
  testServer(server, {
    session$setInputs(fruit = "apple")
    expect_equal(output$value, "apple")

    session$setInputs(fruit = "other", other = "orange")
    expect_equal(output$value, "orange")
  })
})
#> Test passed
```

Здесь не выполняется проверка на то, что вариант `other` автоматически выбирается при начале ввода текста в поле. К сожалению, такой тип взаимодействия невозможно отследить при помощи функции `testServer()`:

```
test_that("returns other value when primary is other", {
  testServer(server, {
    session$setInputs(fruit = "apple", other = "orange")
    expect_equal(output$value, "orange")
  })
})
#> — Failure (<text>:2:3): returns other value when primary is other —
#> output$value (`actual`) not equal to "orange" (`expected`).
#>
#> `actual`:  "apple"
#> `expected`: "orange"
#> Backtrace:
#> 1. shiny::testServer(...)
#> 22. testthat::expect_equal(output$value, "orange")
```

Поэтому нам нужно воспользоваться объектом *ShinyDriver*:

```
test_that("automatically switches to other", {
  app <- ShinyDriver$new(shinyApp(ui, server))
  app$setInputs(other = "orange")
  expect_equal(app$getValue("fruit"), "other")
})
```

```
expect_equal(app$getValue("value"), "orange")
})
```

Обычно лучше использовать функцию `testServer()`, пока вам хватает ее возможностей, а к помощи объекта *ShinyDriver* прибегать только в случае необходимости отследить взаимодействия с пользователем в настоящем браузере.

ТЕСТИРОВАНИЕ ВИЗУАЛЬНЫХ ЭЛЕМЕНТОВ

А как насчет таких компонентов, как графики или виджеты HTML, корректное поведение которых достаточно трудно описать при помощи кода? В этом случае вы можете использовать последний из описываемых в этой книге, наиболее богатый, но одновременно и наименее надежный метод тестирования, заключающийся в сохранении скриншотов компонентов. Этот метод объединяет в себе технику сохранения скриншотов из пакета *shinytest* с созданием полноценных снимков при помощи пакета *testthat*. Работает это подобно описанному выше, за исключением того, что создается файл с расширением *.png*, а не *.md*. Кроме того, в данном случае невозможно увидеть различия между файлами в консоли, так что необходимо вызвать функцию *testthat::snapshot_review()*, использующую приложение Shiny для визуализации различий.

Главным недостатком тестирования с использованием скриншотов является необходимость контроля со стороны для подтверждения результатов даже при малейших изменениях. Это является проблемой, поскольку очень трудно заставить разные компьютеры генерировать воспроизводимые по пикселям скриншоты. Различия в операционных системах, версиях браузеров и даже шрифтах приводят к появлению скриншотов, выглядящих одинаково, но при этом имеющих небольшие различия. Это означает, что визуальные тесты лучше всего проводить на одном и том же компьютере и не включать их в инструменты непрерывной интеграции. Конечно, существуют способы обойти эти сложности, но они достаточно специфичны и выходят за рамки данной книги.

Создание скриншотов отдельных элементов при помощи пакета *shinytest* и полноценных снимков посредством *testthat* – это довольно новый функционал, и пока не до конца ясно, какой интерфейс применительно к ним можно считать идеальным. Давайте посмотрим, как работает следующий код:

```
path <- tempfile()
app <- ShinyDriver$new(shinyApp(ui, server))

# Сохраняем скриншот во временный файл
app$takeScreenshot(path, "plot")
#
expect_snapshot_file(path, "plot-init.png")
```

```
app$setValue(x = 2)
app$takeScreenshot(path, "plot")
expect_snapshot_file(path, "plot-update.png")
```

Вторым аргументом функции `expect_snapshot_file()` задается имя файла, в котором будет сохранено изображение в директории со снимками. Если тесты собраны в файле с именем `test-app.R`, то наши два снимка будут сохранены под именами `tests/testthat/_snaps/app/plot-init.png` и `tests/testthat/_snaps/app/plot-update.png`. Советую делать имена файлов достаточно короткими, но при этом интуитивно понятными, чтобы вы могли легко узнать, что именно тестируете, когда что-то пойдет не так.

Философия

В данной главе мы основное внимание уделяли технике и методам тестирования приложений. Но когда вы все это освоите в достаточной степени, перед вами встанут гораздо более глобальные вопросы, связанные со структурой и философией тестирования.

В этой связи полезно помнить о таких феноменах, как *ложноположительное* (false positive) и *ложноотрицательное* (false negative) срабатывание: можно написать тесты, которые будут успешно срабатывать, хотя не должны, и наоборот. При начале работы в области тестирования больше всего проблем возникает именно с ложноположительными результатами: как убедиться, что ваши тесты на самом деле отлавливают ошибки? Но я уверен, что вы достаточно быстро справитесь с этими вопросами.

Когда стоит писать тесты?

Тесты необходимо писать в трех случаях:

- перед написанием кода. Этот стиль программирования получил название *разработка на основе тестирования* (test-driven development). Если вы точно знаете, как должна работать ваша будущая функция, имеет смысл заложить это знание в виде кода еще до реализации функции;
- после написания кода. Во время написания программы у вас в голове будет формироваться своеобразный список мест в коде, вызывающих наибольшее беспокойство. По окончании работы вы можете преобразовать свои опасения в тесты, чтобы гарантировать правильную работу функции. При написании тестов опасайтесь делать это слишком рано. Если вы продолжаете развивать и расширять свою функцию, поддерживать связанные с ней тесты в актуальном состоянии бывает очень утомительно. Это явный признак того, что вы поспешили;
- после нахождения ошибки. Каждый раз, когда вы находите ошибку в коде, подумайте о том, чтобы превратить ее в автоматизированный тест. В этом есть два положительных момента. Во-первых, для создания полноценного полезного теста вам необходимо постепенно упрощать

задачу с целью получения минимально возможного воспроизводимого примера, который и стоит включать в тест. Во-вторых, таким образом вы гарантируете, что найденная ошибка не повторится в будущем.

ЗАКЛЮЧЕНИЕ

В данной главе мы говорили о том, как правильно преобразовать приложение в пакет, чтобы можно было воспользоваться всеми преимуществами мощнейших инструментов, входящих в состав пакета *testthat*. Если вы ни разу до этого не создавали пакет, вам, возможно, казалось, что этот процесс может быть слишком сложным. Но, как видите, пакет представляет собой не что иное, как набор соглашений, который вы с легкостью можете адаптировать к своему приложению Shiny. Создание пакета потребует от вас кое-какой дополнительной работы, но в результате вы получите ощутимый бонус в виде возможности создавать автоматизированные тесты, что, в свою очередь, поможет в разработке действительно сложных приложений.

В заключительных двух главах вы узнаете о безопасности приложений Shiny и способах повышения их производительности.

Глава 22

Безопасность

Большинство приложений Shiny разворачиваются за *брандмауэром* (firewall) компании, и, поскольку обычно вам незачем беспокоиться о том, что кто-то из коллег захочет взломать ваше приложение¹, всерьез задумываться о *безопасности* (security) нет необходимости. Но если в вашем приложении содержатся данные, которые должны быть доступны только определенным сотрудникам, или вы хотите развернуть приложение публично, вам стоит задаться вопросом о приемлемом уровне безопасности. При защите приложений стоит основное внимание уделить безопасности двух компонентов:

- данных: вы должны сделать все возможное, чтобы злоумышленники не смогли добраться до конфиденциальной информации;
- вычислительных ресурсов: вам необходимо убедиться, что ваши серверы не используются третьими лицами для майнинга биткоинов или рассылки спама.

К счастью для вас, обеспечение безопасности – это командный вид спорта. Кто бы ни разворачивал ваше приложение, он должен нести ответственность за обеспечение безопасности *между* приложениями, чтобы приложение А не могло получить доступ к коду и данным приложения Б, а также занять всю доступную память и вычислительные ресурсы сервера. В вашу зону ответственности входит обеспечение безопасности *внутри* приложения, чтобы злоумышленник не смог достичь своих целей. В данной главе мы поговорим об основах безопасности в Shiny, отдельно затронув темы сохранности данных и вычислительных ресурсов.

Если вас интересует тема безопасности в R в целом, я могу посоветовать вам посмотреть выступление Колина Гиллеспі (Colin Gillespie) на конференции *useR! 2019* по адресу <https://www.youtube.com/watch?v=5odJxZj9LE4>. А мы, как и всегда, начнем с загрузки пакета:

```
library(shiny)
```

¹ Если вы не можете на это рассчитывать, у вас могут быть серьезные проблемы. Хотя существуют компании с *нулевым уровнем доверия* (zero-trust), и вам необходимо заранее обсудить принятую в вашей организации модель с отделом информационной поддержки.

ДАННЫЕ

Традиционно наиболее засекреченными данными в компании являются *сведения, идентифицирующие личность* (personally identifying information – PII), регламентные документы, данные о кредитных картах, состоянии здоровья сотрудников и другая информация, публикация которой в открытом доступе может стать настоящим кошмаром для компании. К счастью, большинство приложений Shiny не работают с такого рода конфиденциальными данными¹, но вам тоже есть о чем беспокоиться – а именно о паролях пользователей. Первое, что вам необходимо знать, – это то, что пароли никогда не следует хранить в исходном коде приложения. Вместо этого их лучше держать в переменных окружения или, если у вас много паролей, воспользоваться пакетом *config* (<https://github.com/rstudio/config>). Так или иначе, убедитесь, что пароли никогда не включаются в вашу систему контроля версий исходного кода – для этого можно воспользоваться файлом *.gitignore*. Я также рекомендую документировать способ получения новым разработчиком доступа к личным данным.

Ваше приложение может оперировать данными ваших пользователей. Если вам необходимо выполнять авторизацию пользователей по их имени и паролю, не пытайтесь реализовать эту часть приложения самостоятельно. Это довольно сложная область, и вы можете упустить что-то важное. Вместо этого вам лучше объединить усилия с отделом информационных технологий для реализации совместного решения. Много полезного в этой области можно почитать в следующих документациях RStudio: <https://solutions.rstudio.com/sys-admin/auth/kerberos>, <https://db.rstudio.com/best-practices/deployment>. Обратите внимание, что код внутри функции `server()` изолирован, чтобы не пересекались области видимости разных пользовательских сессий. Единственное исключение может быть, если вы используете механизм кеширования, о чем мы будем говорить в заключительной главе.

Наконец, заметьте, что в элементах ввода Shiny применяется проверка данных на клиенте, т. е. правильность ввода контролируется при помощи скриптов JavaScript в браузере, а не в R. А это дает возможность хорошо осведомленному злоумышленнику послать значения, которые вы не ожидаете получить. Давайте рассмотрим следующий простой пример:

```
secrets <- list(
  a = "my name",
  b = "my birthday",
  c = "my social security number",
  d = "my credit card"
)

allowed <- c("a", "b")
ui <- fluidPage(
  selectInput("x", "x", choices = allowed),
  textOutput("secret")
)
```

¹ Если это не так, то я настоятельно рекомендую разрабатывать приложение в сотрудничестве с инженером-программистом, обладающим знаниями в области защиты данных.

```
)

server <- function(input, output, session) {
  output$secret <- renderText({
    secrets[[input$x]]
  })
}
```

Вы предполагаете, что пользователь может иметь доступ только к информации об имени и дате рождения, но никак не о номере социального страхования и кредитной карте. Но, увы, злоумышленник может открыть консоль JavaScript в браузере и запустить код `Shiny.setInputValue("x", "c")`, который позволит ему добраться до данных о социальном страховании. Чтобы предотвратить такую ситуацию, вы должны контролировать ввод информации пользователем в коде R:

```
server <- function(input, output, session) {
  output$y <- renderText({
    req(secrets$x %in% allowed)
    secrets$y[[secrets$x == input$x]]
  })
}
```

Я умышленно не стал создавать дружественное сообщение об ошибке. Единственный вариант, когда эта ошибка возникнет, возможен в случае попытки взлома нашего приложения, а мы не собираемся помогать злоумышленникам.

ВЫЧИСЛИТЕЛЬНЫЕ РЕСУРСЫ

Надеюсь, вам понятно, что приложение, показанное ниже, несет в себе явную угрозу, поскольку позволяет пользователю запустить любой код на R. При желании он может удалить важные файлы, изменить данные или получить конфиденциальную информацию:

```
ui <- fluidPage(
  textInput("code", "Enter code here"),
  textOutput("results")
)

server <- function(input, output, session) {
  output$results <- renderText({
    eval(parse(text = input$code))
  })
}
```

Вообще, использование комбинации функций `parse()` и `eval()` является дурным тоном для любого приложения Shiny¹: они моментально делают

¹ Единственное исключение составляет случай, когда в них никак не задействованы данные, предоставляемые пользователем.

ваше приложение уязвимым. Также вы никогда не должны использовать функции `source()` и `markdown::render()` с загруженными файлами `.R` и `.Rmd`. Но это должно быть вполне очевидно, поэтому вряд ли может стать источником серьезных проблем.

Гораздо сложнее дело обстоит с функциями `parse()` и `eval()`, которые вместе или по отдельности используют другие функции, о чем вы можете даже не догадываться. Вот наиболее популярные из них.

- **Функции построения моделей.** Вы можете построить модель, которая будет запускать произвольный код на R:

```
df <- data.frame(x = 1:5, y = runif(5))
mod <- lm(y ~ {print("Hi!"); x}, data = df)
#> [1] "Hi!"
```

По этой причине бывает опасно давать пользователю возможность безопасно определять собственные модели.

- **Glue-метки.** Пакет *glue* позволяет очень удобно создавать строки на основе данных:

```
title <- "foo"
number <- 1
glue::glue("{title}-{number}")
#> foo-1
```

В то же время функция *glue()* дает возможность выполнять любой код внутри фигурных скобок (`{}`):

```
glue::glue("{title}-{print('Hi'); number}")
#> [1] "Hi"
#> foo-1
```

Если вы хотите предоставить пользователю возможность самому генерировать метки, лучше будет использовать функцию *glue::glue_safe()*, позволяющую извлекать значения из переменных, но не дающую выполнять произвольный код:

```
glue::glue_safe("{title}-{number}")
#> foo-1
glue::glue_safe("{title}-{print('Hi'); number}")
#> Error in .transformer(expr, env): object 'print('Hi'); number' not found
```

- **Преобразование переменных.** Не существует способа позволить пользователю безопасно предоставить сниппет (фрагмент кода) для преобразования переменных при работе с пакетами *dplyr* или *ggplot2*. Вы можете предполагать, что пользователь напишет `log10(x)`, а он может написать `{print("Hi"); log10(x)}`.

По этой же причине не рекомендуется использовать старую версию функции *ggplot2::aes_string()* совместно с пользовательским вводом. Вместо этого лучше применять техники, описанные в главе 12.

Аналогичные проблемы возникают и с инструкциями на языке запросов SQL. Представьте, что вы решили собрать запрос SQL при помощи функции `paste()` следующим образом:

```
find_student <- function(name) {
  paste0("SELECT * FROM Students WHERE name = ('", name, "');")
}
find_student("Hadley")
#> [1] "SELECT * FROM Students WHERE name = ('Hadley');"
```

В этом случае злоумышленник может передать вместо имени свою строку¹:

```
find_student("Robert"); DROP TABLE Students; --")
#> [1] "SELECT * FROM Students WHERE name = ('Robert'); DROP TABLE Students; --');"
```

Эта строка выглядит странно, но она представляет собой вполне допустимый запрос с точки зрения языка SQL:

- запрос `SELECT * FROM Students WHERE name = ('Robert');` осуществляет поиск студентов по имени Роберт;
- запрос `DROP TABLE Students;` удаляет все записи из таблицы `Students` (!!!);
- `--'` – это комментарий для предотвращения появления синтаксической ошибки.

Во избежание подобных проблем никогда не генерируйте запрос SQL при помощи функции `paste()`, а вместо этого применяйте инструменты, исключающие задействование пользовательского ввода (например, *dbplyr* (<https://dbplyr.tidyverse.org>)), или используйте защищенную функцию `glue::glue_sql()`, как показано ниже:

```
con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
find_student <- function(name) {
  glue::glue_sql("SELECT * FROM Students WHERE name = ({name});", .con = con)
}
find_student("Robert"); DROP TABLE Students; --")
#> <SQL> SELECT * FROM Students WHERE name = ('Robert'); DROP TABLE Students; --');
```

Это не так просто понять, но представленный способ действительно является безопасным, поскольку переданная одинарная кавычка будет преобразована в две кавычки, и поиск в таблице `Students` будет выполнен по полной строке `"Robert"); DROP TABLE Students; -"`.

¹ Пример был взят с сайта <https://xkcd.com/327>.

Глава 23

Производительность

В приложениях Shiny могут одновременно комфортно работать тысячи и десятки тысяч пользователей, если оно спроектировано и разработано правильно. Однако большинство приложений для экспресс-аналитики пишутся в такой спешке, что появляются на свет совершенно непригодными для эффективного использования. Это особенность Shiny: фреймворк позволяет так быстро создавать приложения для реальных задач, что далеко не все разработчики успевают подумать над тем, как повысить их эффективность при одновременной работе большого количества пользователей. К счастью, даже небольшие правки в приложении могут привести к 10–100-кратному приросту его *производительности* (performance). Именно о способах ускорить работу приложения мы в данной главе и поговорим.

Начнем с одной любопытной метафоры, сравнивающей приложение Shiny с рестораном. Позже вы узнаете, как проводить *сравнительное*, или *эталонное*, *тестирование* (benchmarking) приложения с использованием пакета *shinyloadtest* для имитирования одновременной работы с приложением множества пользователей. Именно с этого зачастую стоит начинать, поскольку так вы сможете быстро определить проблемные участки приложения и оценить влияние сделанных вами изменений.

После этого посмотрим, как можно профилировать приложения с помощью пакета *profvis* для выявления узких мест.

Наконец, вы освоите полезные техники для оптимизации кода, которые помогут повысить эффективность приложения. Вы научитесь кешировать реактивы, выносить код для подготовки данных за пределы приложения и использовать прикладную психологию для создания ощущения предельной скорости работы приложения.

Я очень рекомендую вам посмотреть выступление Джо Ченга на конференции *rstudio::conf(2019)* по адресу <https://www.rstudio.com/resources/rstudio-conf-2019/shiny-in-production-principles-practices-and-tools/>, в котором он описывает процесс оценки приложений, их профилирования и оптимизации. В своем выступлении Джо рассказывает о всех стадиях подготовки приложения на реалистичном примере, с которым можно ознакомиться по адресу <https://rstudio.github.io/shinyloadtest/articles/case-study-scaling.html>.

```
library(shiny)
```

Особую благодарность хочу выразить коллегам по RStudio Джо Ченгу, Шону Лоппу (Sean Lopp) и Алану Диперту (Alan Dipert), выступления которых на конференции *RStudio::conf()* помогли мне в написании этой главы.

УЖИН В РЕСТОРАНЕ SHINY

Говоря о производительности фреймворка Shiny, бывает полезно представить приложение в виде ресторана¹. Посетитель (пользователь) приходит в ресторан (сервер) и делает заказ (запрос), который исполняет шеф-повар (процесс R). Это сравнение действительно очень полезно, поскольку, как и в ресторане, один процесс R может обслуживать сразу несколько клиентов одновременно, и способы справляться с повышенной загрузкой очень похожи.

Для начала можно исследовать возможности для повышения производительности нашего шеф-повара (оптимизации кода R). Для этого необходимо потратить какое-то время на анализ работы шефа на предмет наличия узких мест (профилирование) и выработку мер по ее ускорению (оптимизацию). Например, можно нанять помощника шеф-повара, который будет приходить еще до первого посетителя и делать овощные заготовки (подготовку данных), или купить оборудование для ускорения рабочего процесса (использование более быстрых пакетов R).

Кроме того, вы можете также задуматься об увеличении количества поваров (процессов) в ресторане (на сервере). К счастью, добавить процессы сегодня гораздо проще, чем найти толковых шефов². Если нанимать поваров бесконечно, то вам просто не хватит для них места на кухне, и вам придется расширяться (память и процессор). Добавление аппаратных ресурсов для возможности запуска новых процессов называется масштабированием, или *вертикальным масштабированием* (scaling up).

В какой-то момент вы все доступные рабочие площади на кухне заставите поварями, но вам и этого будет мало. Тогда придется открывать новые рестораны. Этот процесс называется расширением, или *горизонтальным масштабированием* (scaling out). Применительно к Shiny это означает использование большего количества серверов. Горизонтальное масштабирование может позволить вам обслуживать любое количество клиентов, пока у вас есть деньги на оплату имущества и инфраструктуры. В данной главе мы больше не будем затрагивать тему горизонтального масштабирования – здесь все достаточно просто, но всецело зависит от вашей инфраструктуры развертывания приложения.

Есть одно место, в котором искусно придуманная метафора ломается: обычно шеф-повар способен готовить несколько блюд одновременно, ловко и профессионально распределяя время между рецептами в зависимости от их специфики. В то же время R является *однопоточным* (single-threaded),

¹ Спасибо Шону Лоппу за эту аналогию, проведенную во время выступления на конференции *rstudio::conf(2018)*, которое можно посмотреть по адресу <https://www.rstudio.com/resources/rstudioconf-2018/scaling-shiny>. Я очень рекомендую посмотреть это видео тем, кто сомневается, что приложения Shiny способны работать в среде из тысяч пользователей.

² Опять же, это напрямую зависит от того, как развернуто ваше приложение, но обычно вы можете динамически контролировать количество задействованных процессов в зависимости от числа подключенных пользователей. Советую почитать статью по адресу <https://support.rstudio.com/hc/en-us/articles/231874748> для получения дополнительной информации.

а это означает, что он не может заниматься несколькими делами одновременно. Это приемлемо, когда все блюда из меню ресторана готовятся достаточно быстро, но если у вас в меню есть свиная корейка су-вид, требующая приготовления в течение суток, все остальные посетители вынуждены будут ждать целый день, чтобы повар уделил им внимание. К счастью, в программировании есть средство от этого, именуемое *асинхронностью* (async programming). Это очень большая тема, выходящая за рамки данной книги.

ОЦЕНКА ПРОИЗВОДИТЕЛЬНОСТИ

Вы почти всегда начинаете разработку приложений для себя, так что, продолжая аналогию с рестораном, у вас есть личный шеф-повар, обслуживающий единственного посетителя – вас! И хотя лично вы можете быть довольны производительностью приложения в таком режиме, всегда приходит время, когда вы задаетесь вопросом о том, сможет ли оно столь же эффективно работать при десяти одновременных подключениях. *Оценка производительности* (benchmarking) позволяет измерить эффективность вашего приложения при работе одновременно с несколькими пользователями без необходимости размещать потенциально медленный продукт в публичном доступе. А если вы хотите, чтобы с вашим приложением одновременно работали сотни или тысячи людей, оценка производительности поможет рассчитать, сколько пользователей сможет обслуживать один процесс, а значит, и сколько серверов вам понадобится.

В оценке производительности приложения нам будет помогать пакет *shinyloadtest* (<https://rstudio.github.io/shinyloadtest>), и работа с ним подразумевает выполнение трех шагов.

1. Используйте функцию `shinyloadtest::record_session()` для записи скрипта, имитирующего типичного пользователя.
2. Запускайте скрипт одновременно от множества пользователей с помощью инструмента командной строки *shinycannon*.
3. Воспользуйтесь функцией `shinyloadtest::report()` для анализа результатов.

Теперь посмотрим, как каждый из этих шагов работает по отдельности. Если вам понадобится больше деталей, вы можете обратиться к документации пакета *shinyloadtest*.

Запись

Если вы проводите оценку производительности на ноутбуке, вам необходимо будет использовать два отдельных процесса R: один для Shiny и один для пакета *shinyloadtest*¹:

¹ Легче всего это можно сделать, открыв два экземпляра RStudio. Также вы можете открыть терминал и ввести R.

- в одном процессе запустите ваше приложение и скопируйте предложенную вам ссылку:

```
runApp("myapp.R")
#> Listening on http://127.0.0.1:7716
```

- во втором процессе передайте полученную ссылку на вход функции `record_session()`:

```
shinyloadtest::record_session("http://127.0.0.1:7716")
```

Функция `record_session()` откроет новое окно с вашим приложением с возможностью записи всех ваших действий. Поработайте с приложением как рядовой пользователь. Я советую заранее расписать все свои действия – так вам будет легче повторять их в будущем, если поймете, что упустили что-то важное. Успех оценки производительности целиком зависит от полноты ваших действий при работе с приложением, так что уделите некоторое время разработке оптимальной последовательности манипуляций с приложением. Например, вы должны не забывать делать паузы для раздумий, чтобы максимально точно имитировать действия человека.

После завершения работы с приложением закройте окно, и пакет *shinyloadtest* создаст файл *recording.log* в вашей рабочей директории. В этом файле будут сохранены все ваши действия при работе с приложением в виде, доступном для повторных запусков. Не потеряйте файл – он вам понадобится на следующем шаге.

Хотя оценку производительности вполне можно проводить и на рабочем ноутбуке, для максимально приближенной к реальности имитации процесса развертывания приложения. Так что если в вашей компании предусмотрены правила для развертывания приложений Shiny, попробуйте договориться с парнями из отдела информационных технологий, чтобы они позволили вам настроить свое окружение для оценки производительности на сервере.

Запуск

Теперь в вашем распоряжении есть скрипт с записанными действиями одного пользователя, и далее мы используем его для имитации многопользовательской среды при помощи инструмента *shinycannon*. Установку этого инструмента вам придется произвести отдельно, поскольку он не является пакетом R. Написан *shinycannon* на Java, идеально подходящем для обработки десятков или сотен веб-запросов одновременно с использованием минимума ресурсов. Это позволит даже на вашем ноутбуке запустить и приложение, и множество имитаций пользователей. Для установки *shinycannon* следуйте инструкции, расположенной на сайте RStudio по адресу <https://rstudio.github.io/shinyloadtest/#shinycannon>.

Запустите *shinycannon* в терминале, как показано ниже:

```
shinycannon recording.log http://127.0.0.1:7911 \
--workers 10 \
```

```
--loaded-duration-minutes 5 \
--output-dir run1
```

Инструмент *shinycannon* принимает следующие аргументы:

- первый аргумент представляет собой путь к записанному ранее скрипту;
- второй аргумент – это путь к вашему приложению, который вы копи-ровали и вставляли;
- аргументом `--workers` задается количество одновременно запускае-мых пользователей. В нашем примере мы запустили имитацию десяти одновременно работающих с приложением пользователей;
- аргумент `--loaded-duration-minutes` определяет время, в течение кото-рого будет длиться оценка. Если это время превышает длительность записи действий пользователя, по окончании запись будет запускаться снова;
- аргумент `--output-dir gives` отвечает за путь, где будут сохранены ре-зультаты оценки производительности. Скорее всего, по ходу оптими-зации приложения вам придется многократно запускать оценку его производительности, так что отнеситесь к выбору пути для хранения результатов ответственно.

При первом запуске оценки производительности рекомендуется вы-бирать небольшое количество пользователей и непродолжительное вре-мя – это позволит вам достаточно быстро отловить первые серьезные проблемы.

Анализ

Теперь, когда вы провели оценку производительности вашего приложения с имитацией нескольких пользователей, пришло время взглянуть на резуль-таты. Для начала загрузите данные в R при помощи функции `load_runs()`:

```
library(shinyloadtest)
df <- load_runs("scaling-testing/run1")
```

В результате вы получите объект *tibble*, который можно анализировать любыми доступными средствами. Обычно же вы будете формировать стан-дартный отчет пакета *shinyloadtest*. Это стандартный отчет в HTML с группи-ровкой по наиболее полезным показателям:

```
shinyloadtest_report(df, "report.html")
```

Я не буду приводить здесь все страницы отчета, а сосредоточусь на наи-более важном, по моему мнению, графике, отражающем *длительность сессий* (session duration). Если вы хотите подробнее узнать об остальных страницах отчета, я настоятельно рекомендую почитать статью по адресу <https://rstudio.github.io/shinyloadtest/articles/analyzing-load-test-logs.html>.

На графике длительности сессий каждому пользователю соответствует отдельная строка. События представлены прямоугольниками, ширина которых пропорциональна длительности событий, а цвет указывает на тип. Красная линия показывает текущее время записи.

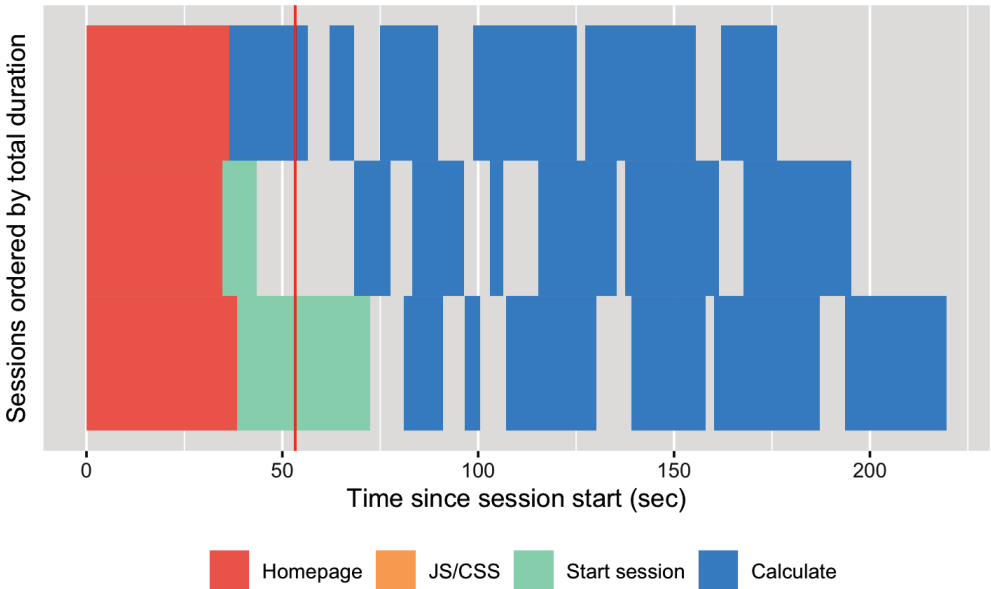


Рис. 23.1 ❖ Анализ длительности сессий

Анализируя этот график, нужно постараться ответить на следующие вопросы:

- сопоставима ли производительность приложения при работе одного пользователя и нескольких одновременно? Если да, что ж, поздравляем, ваше приложение уже оптимизировано, и дальше эту главу вы можете не читать. Есть ли задержки на странице *Homepage*? Если да, вероятно, вы выполняете слишком много работы в функции интерфейса пользователя `ui()`;
- наблюдаются ли замедления на странице *Start session*? Скорее всего, это может свидетельствовать о чрезмерной загруженности вашей серверной функции. Обычно запуск серверной функции должен производиться быстро, поскольку все, что вы делаете здесь, – это определяете реактивный график, который запускается на следующем шаге. Если на этом этапе есть задержка, значит, необходимо попробовать перенести ресурсоемкий код за пределы функции `server()`, чтобы он выполнялся один раз после запуска приложения, или поместить в реактив, который будет вычисляться по требованию. В большинстве же случаев наиболее длительные задержки будут присутствовать на страницах *Calculate*, что указывает на медленные вычисления реактивов. Борьбе с такими проявлениями и будет посвящена оставшаяся часть главы.

Профилирование

Если в вашем приложении были обнаружены задержки на этапе вычислений (*Calculate*), необходимо выяснить, какие именно операции отнимают так много времени, а значит, надо выполнить операцию *профилирования* (*profile*) кода для определения его узких мест. Для этого воспользуемся пакетом *profvis* (<https://rstudio.github.io/profvis>), который поможет нам визуализировать данные о профилировании, собранные функцией *utils::Rprof()*. Начнем с представления так называемого огненного графика, применяемого при профилировании кода, после чего посмотрим, как можно использовать пакет *profvis* для профилирования кода R и приложений Shiny.

Огненный график

В программировании в целом наиболее популярным типом визуализации данных о профилировании кода является *огненный график* (*flame graph*). Для лучшего его понимания мы сначала повторим основы выполнения кода в рамках приложения Shiny, после чего постепенно, шаг за шагом, построим итоговую визуализацию.

Мы будем работать с представленным ниже кодом, в котором я использовал функцию *pause()* (об этом далее) для индикации выполнения работы:

```
library(profvis)

f <- function() {
  pause(0.2)
  g()
  h()
  10
}

g <- function() {
  pause(0.1)
  h()
}

h <- function() {
  pause(0.3)
}
```

Если бы я попросил вас мысленно запустить функцию *f()* и пояснить ход выполнения программы, вы наверняка сделали бы это примерно так:

- запускается функция *f()*;
- функция *f()* вызывает функцию *g()*;
- функция *g()* вызывает функцию *h()*;
- функция *f()* вызывает функцию *g()*.

Этот процесс довольно непросто отследить, поскольку мы не видим, как вызовы вложены друг в друга. Давайте попробуем применить несколько иную концепцию записи:

- f ;
- $f > g$;
- $f > g > h$;
- $f > h$.

Таким образом, мы отобразили *стек вызовов* (call stack), о котором уже упоминали в главе 5, когда говорили о процессе отладки приложений. Стек вызовов представляет собой полную последовательность вызовов, ведущую к функции.

Этот стек можно попробовать преобразовать в диаграмму, заключив каждый вызов в прямоугольник, как показано на рис. 23.2.

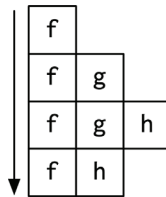


Рис. 23.2 ❖ Диаграмма вызовов функций

Стрелку, направленную сверху вниз, можно воспринимать как время – так же мы воспринимаем и процесс выполнения кода. Однако, следуя общему соглашению, огненный график изображается таким образом, чтобы ось времени располагалась слева направо, так что давайте повернем нашу диаграмму на 90 градусов, как показано на рис. 23.3.

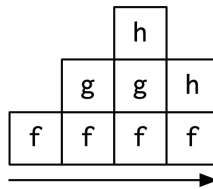


Рис. 23.3 ❖ Перевернутая диаграмма вызовов функций

Теперь сделаем диаграмму более информативной, продлив ширину каждого вызова пропорционально занимаемому времени, как показано на рис. 23.4. Также я нанес линию сетки на фоне, чтобы было легче проверить правильность начертаний.

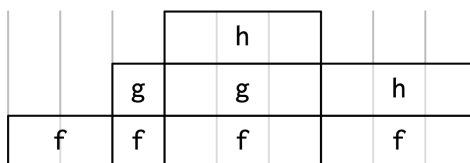


Рис. 23.4 ❖ Диаграмма вызовов функций с длительностями

Наконец, объединим соседствующие вызовы одних и тех же функций во-едино, что приведет к образованию графика, показанного на рис. 23.5.

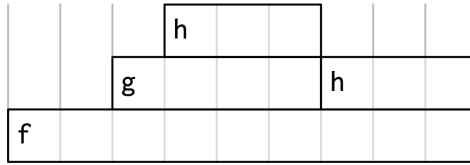


Рис. 23.5 ❖ Диаграмма вызовов функций с длительностями и объединенными вызовами

Это и есть огненный график! Теперь мы видим, как долго выполняется функция `f()` и почему она так долго выполняется, т. е. на что расходуется ее время.

Вас может удивить название огненный график. На практике такие графики зачастую раскрашивают случайным образом в теплые тона, что символизирует перегрев компьютера от нагрузки. Однако, поскольку сами цвета не несут здесь никакой смысловой нагрузки, мы обычно оставляем график черно-белым. О предпочтительной цветовой схеме, альтернативах и истории огненного графика вы можете почитать в статье по адресу <https://queue.acm.org/detail.cfm?id=2927301>.



Рис. 23.6 ❖ Огненный график в цвете

Профилирование кода R

Теперь, когда вы понимаете, что такое огненный график, давайте применим полученные знания на практике с помощью пакета *profvis*. Использовать этот пакет очень просто: достаточно заключить профилируемый код в функцию `profvis::profvis()`, как показано ниже:

```
profvis::profvis(f())
```

После выполнения кода пакет *profvis* откроет окно с интерактивной визуализацией, показанной на рис. 23.7. Как видите, эта визуализация очень похожа на график, который мы рисовали от руки. Но длительности отличаются. Причина в том, что профилировщик останавливает выполнение кода каждые 10 мс и записывает стек вызовов. К сожалению, невозможно производить остановки в точности тогда, когда мы хотим, поскольку R в этот момент может быть занят операцией, которую прервать просто нельзя. В результате

мы получаем определенные случайные колебания на диаграмме. Если пере-профилеровать этот код еще раз, вывод может оказаться несколько иным.

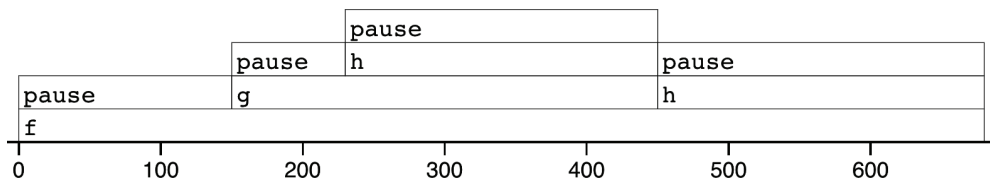


Рис. 23.7 ❖ Результат профилерования функции `f()` при помощи пакета *profvis*.
На оси *X* располагается время в мс, а на оси *Y* – глубина стека вызовов

Подобно огненному графику, профилер *profvis* старается найти и отобразить выполняющийся исходный код, чтобы вы могли щелкнуть по функции и увидеть, что в ней происходит.

Профилерование приложения Shiny

При выполнении профилерования приложения Shiny все происходит почти так же. Чтобы увидеть различия, давайте рассмотрим простое приложение:

```
ui <- fluidPage(
  actionButton("x", "Push me"),
  textOutput("y")
)

server <- function(input, output, session) {
  output$y <- eventReactive(input$x, f())
}

# Note the explicit call to runApp() here: this is important
# as otherwise the app won't actually run.
profvis::profvis(runApp(shinyApp(ui, server)))
```

Результат показан на рис. 23.8.

Вывод очень похож на предыдущий, за исключением следующих различий:

- функция `f()` более не располагается на нижнем слое стека вызовов. Она перебралась на четвертый этаж, поскольку ее вызывает функция `eventReactiveHandler()` – внутренняя функция, запускаемая в результате вызова функции `eventReactive()`. На втором этаже стека располагается `output$y`, которая заключена в функцию `runApp()`;
- на графике видны две высокие башни. Обычно при анализе работы приложения их можно игнорировать – они занимают не так много времени, а их возникновение будет варьироваться от запуска к запуску по причине вероятностного характера модели. Если вы хотите узнать, что происходит в этих башнях, наведите на них курсор мыши. В нашем случае узкая и высокая башня в левой части графика характеризует

- загрузка данных из интернета происходит в отдельном процессе, не контролируемом R.

ПОВЫШЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ

Наиболее эффективным способом повысить производительность приложения являются поиск узких мест при помощи профилирования и их оптимизация. Определив проблемный участок кода, убедитесь, что он находится в отдельной функции. После этого создайте минимальный фрагмент кода, воспроизводящий проблемы со скоростью, и проведите его контрольное профилирование, чтобы убедиться, что правильно определили узкое место. Запускайте этот фрагмент, параллельно выполняя оптимизацию кода. Также я рекомендую написать несколько тестов, которые мы обсуждали в главе 21, поскольку – скажу по своему опыту – легче всего ускорить код можно, внеся в него ошибки.

Код приложения Shiny – это в первую очередь программный код на языке R, так что большинство приемов по оптимизации здесь будут схожи. Почитать подробнее о повышении производительности кода можно в соответствующей главе книги *Advanced R* по адресу <https://adv-r.hadley.nz/perf-improve.html> и в книге *Efficient R programming* авторства Колина Гиллеспи (Colin Gillespie) и Робина Лавлейса (Robin Lovelace) по адресу <https://csgillespie.github.io/efficientR>. Я не буду здесь пересказывать советы из этих учебных пособий, а вместо этого сосредоточусь на техниках, наиболее применимых к приложениям Shiny. Также я очень рекомендую послушать выступление Алана Диперта (Alan Dipert) на конференции *rstudio::conf(2018)* по адресу <https://www.rstudio.com/resources/rstudioconf-2018/make-shiny-fast-by-doing-as-little-work-as-possible/>.

Начните с определения фрагментов кода, которые запускаются чаще, чем вы ожидаете. Убедитесь, что одна и та же работа не выполняется в разных реактивах и что реактивный график обновляется не чаще, чем это необходимо (мы подробно говорили об этом в главе 14).

В следующем разделе мы обсудим один из наиболее простых способов повысить производительность приложения, заключающийся в кешировании результатов длительных вычислений, после чего обратимся к теме выделения ресурсоемких операций предварительной обработки данных в отдельные шаги и поговорим о методах улучшения визуального восприятия скорости выполнения ваших приложений пользователем.

КЕШИРОВАНИЕ

Кеширование (caching) представляет собой очень мощную технику, позволяющую повысить производительность приложения. Основная идея заключается в записи входных и выходных значений при каждом вызове функции. При повторном вызове кешированной функции с тем же набором вход-

ных параметров можно не производить вычисление заново, а воспроизвести ранее записанный выход функции. Пакеты, подобные *memoise* (<https://memoise.r-lib.org>), предлагают необходимый функционал для выполнения кеширования в R.

Кеширование особенно эффективно применительно к приложениям Shiny, поскольку сохраненные данные о вызовах функций могут использоваться совместно несколькими пользователями. Это означает, что в многопользовательском режиме работы приложения только первый пользователь вынужден будет ждать результат выполнения вычисления, а остальные при аналогичных вызовах функции смогут получать итог из сохраненного кеша.

Фреймворк Shiny предлагает свой инструмент для кеширования любых реактивных выражений и функций отображения из семейства *render*. Этот инструмент реализован при помощи функции *bindCache()*¹. Как вы знаете, реактивные выражения сами по себе кешируют последнее вычисленное значение, а функция *bindCache()* позволяет сохранить в кеше любое количество значений с возможностью их совместного использования. В данном разделе я познакомлю вас с основами применения функции *bindCache()* и покажу несколько полезных примеров ее использования. Для получения дополнительной информации по теме вы можете обратиться к статьям по адресам <https://shiny.rstudio.com/articles/caching.html> и <https://shiny.rstudio.com/app-stories/weather-lookup-caching.html> на сайте RStudio.

Основы

Функцию *bindCache()* использовать крайне просто. Для этого достаточно при помощи оператора конвейера отправить в эту функцию реактивное выражение или функцию отображения семейства *render*, как показано ниже:

```
r <- reactive(slow_function(input$x, input$y)) %>%
  bindCache(input$x, input$y)

output$text <- renderText(slow_function2(input$z)) %>%
  bindCache(input$z)
```

Дополнительные аргументы представляют собой *ключи кеширования* (*cache key*): эти значения используются для определения того, было ли уже ранее произведено искомое вычисление. Мы поговорим о ключах кеширования подробнее после рассмотрения практических примеров.

Кеширование реактивов

Часто кеширование используется совместно с *веб-API* (web API): даже если сам API обрабатывает данные быстро, вам все равно необходимо послать запрос интерфейсу, подождать ответа от сервера и затем преобразовать по-

¹ Эта функция была представлена в Shiny версии 1.6.0 как обобщение устаревшей функции *renderCachedPlot()*, которая могла быть применена только к графикам.

лученный результат в нужный вам вид. Таким образом, кеширование результатов обращения к API позволяет значительно повысить общую производительность приложения. Давайте проиллюстрируем это на простом примере с использованием пакета *gh* (<https://gh.r-lib.org>), облегчающего доступ к API GitHub.

Представьте, что вы пишете приложение, в котором хотите отображать недавние разработки программиста. Я написал небольшую функцию, получающую данные с GitHub по API и преобразующую ее в *tibble*:

```
library(purrr)

latest_events <- function(username) {
  json <- gh::gh("/users/{username}/events/public", username = username)
  tibble::tibble(
    repo = json %>% map_chr(c("repo", "name")),
    type = json %>% map_chr("type"),
  )
}

system.time(hadley <- latest_events("hadley"))
#>   user  system elapsed
#> 0.138   0.033   0.743
head(hadley)
#> # A tibble: 6 x 2
#>   repo                                type
#>   <chr>                               <chr>
#> 1 hadley/r4ds                        IssuesEvent
#> 2 hadley/mastering-shiny             IssuesEvent
#> 3 hadley/mastering-shiny             IssueCommentEvent
#> 4 hadley/mastering-shiny             IssuesEvent
#> 5 hadley/mastering-shiny             IssueCommentEvent
#> 6 hadley/mastering-shiny             IssuesEvent
```

Теперь преобразуем функцию в простое приложение:

```
ui <- fluidPage(
  textInput("username", "GitHub user name"),
  tableOutput("events")
)

server <- function(input, output, session) {
  events <- reactive({
    req(input$username)
    latest_events(input$username)
  })
  output$events <- renderTable(events())
}
```

Это приложение будет работать довольно вяло, поскольку каждый раз при вводе имени будет происходить обращение на сервер, даже если вы просматривали эту информацию 15 секунд назад. С помощью функции `bindCache()` можно существенно повысить быстродействие нашего приложения:


```
server <- function(input, output, session) {
  events <- reactive({
    req(input$username)
    latest_events(input$username)
  }) %>% bindCache(input$username)
  output$events <- renderTable(events())
}
```

Вы могли обнаружить одну проблему, связанную с этим подходом, – а что будет, если открыть это приложение на следующий день и снова запросить информацию по этому пользователю? Да, вы получите вчерашние данные, даже если у пользователя была за это время новая активность. Здесь есть неявная зависимость от времени, которую нам предстоит сделать явной. Это можно реализовать, добавив к нашему ключу кеширования системную дату в виде *Sys.Date()*, чтобы данные кешировались только в рамках одного дня:

```
server <- function(input, output, session) {
  events <- reactive({
    req(input$username)
    latest_events(input$username)
  }) %>% bindCache(input$username, Sys.Date())
  output$events <- renderTable(events())
}
```

Вы можете беспокоиться тем, что в кеше будут постоянно накапливаться данные, к которым вы никогда не обратитесь в будущем. Спешу вас обрадовать – кеш обладает ограниченным размером и достаточным интеллектом, чтобы освобождать пространство при необходимости.

Кеширование графиков

Чаще всего вы будете кешировать реактивные выражения, но вы также можете использовать функцию *bindCache()* совместно с функциями отображения. Большинство функций отображения работают очень быстро, но есть среди них одна, которая может тормозить при прорисовке сложных графиков. Речь идет о функции *renderPlot()*.

Давайте рассмотрим показанное ниже приложение. При запуске вы заметите, что первая отрисовка графика может занимать до секунды из-за необходимости разместить на диаграмме приблизительно 50 000 точек данных. В следующие разы прорисовка графика будет выполняться практически мгновенно по причине извлечения информации из кеша:

```
library(ggplot2)

ui <- fluidPage(
  selectInput("x", "X", choices = names(diamonds), selected = "carat"),
  selectInput("y", "Y", choices = names(diamonds), selected = "price"),
  plotOutput("diamonds")
)

server <- function(input, output, session) {
```

```
output$diamonds <- renderPlot({
  ggplot(diamonds, aes(.data[[input$x]], .data[[input$y]])) +
    geom_point()
}) %>% bindCache(input$x, input$y)
}
```

Если синтаксис с использованием `.data` вам непонятен, обратитесь к главе 12 за подробностями.

При кешировании графиков необходимо помнить об одной особенности. Каждый график может прорисовываться в разных размерах, поскольку по умолчанию графики занимают всю доступную ширину, а этот показатель меняется с изменением размеров браузера. Такая гибкость отображения может сыграть злую шутку с кешированием, ведь даже разница в один пиксель может привести к невозможности восстановить график из кеша. Во избежание этих проблем функция `bindCache()` кеширует графики с фиксированным размером. Значения по умолчанию при этом выбираются таким образом, чтобы в большинстве случаев все «просто работало», но если вам нужно, вы можете контролировать размеры графиков при помощи аргумента `sizePolicy`. Подробнее об этом можно почитать в справке `?sizeGrowthRatio`.

Ключи кеширования

Также стоит немного поговорить о *ключях кеширования* (`cache key`), представляющих собой набор значений, позволяющих определить, был ли запрошенный результат вычислен ранее. Эти значения также используются для определения реактивных зависимостей, как в случае с первым аргументом функций `observeEvent()` и `eventReactive()`. Неверный выбор ключей кеширования может приводить к неожиданным результатам. Представьте, что у нас есть такой кеш реактивного выражения:

```
r <- reactive(input$x + input$y) %>% bindCache(input$x)
```

При изменении элемента ввода `input$y` реактивное выражение `r()` не будет пересчитываться. И если извлечь результат вычисления из кеша, мы получим сумму текущего значения `x` и значения `y`, каким оно было в момент сохранения в кеше.

Таким образом, ключ кеширования всегда должен включать в себя все реактивные элементы ввода выражения. Также вы можете включить другие значения, не используемые в реактивном выражении. Например, вы можете добавить текущую дату или определенным образом округленное текущее время, чтобы кешированные значения использовались на протяжении ограниченного времени.

Кроме того, вы можете указывать в качестве ключей кеширования другие реактивы, но они должны быть представлены в достаточно простом виде, т. е. в виде атомарных векторов или простых списков из атомарных векторов. Не используйте в этом качестве громоздкие наборы данных, поскольку время, которое потребуется на извлечение информации из кеша в этом случае, перекроет все преимущества техники кеширования.

Область видимости кеша

По умолчанию кешируемые графики сохраняются в памяти, занимая максимум 200 Мб, доступны всем пользователям в одном процессе и удаляются при перезапуске приложения. Вы имеете возможность изменить это поведение для отдельных реактивов или для всей сессии в целом:

- `bindCache(..., cache = "session")` приведет к хранению отдельных кешей для каждой пользовательской сессии. Это позволит сохранять конфиденциальные данные, но при этом может негативно сказаться на эффективности кеширования;
- используйте вызов функции `shinyOptions(cache = cachem::cache_mem())` или `shinyOptions(cache = cachem::cache_disk())`, чтобы позволить кешу действовать в рамках всего приложения. Таким образом вы можете гарантировать, что кеш будет распространяться на несколько процессов и сохраняться между перезапусками приложения. За подробностями вы можете обратиться к справке `?bindCache`.

Также вы можете объединять несколько кешей в единую цепочку или написать свой собственный серверный модуль хранилища. Подробнее узнать об этом можно в документации к пакету *cachem* (<https://cachem.r-lib.org>), лежащему в основе функции `bindCache()`.

ДРУГИЕ СПОСОБЫ ОПТИМИЗАЦИИ

Есть еще два важных метода оптимизации, которые применяются во множестве приложений. Первый из них касается выполнения загрузки и преобразования исходных данных по расписанию, а второй работает в области управления ожиданиями пользователя от приложения.

Запланированные преобразования данных

Представьте, что в вашем приложении используется набор данных, требующий определенной предварительной очистки. В целом подготовка исходных данных может занимать довольно много времени, и вы поняли, что в вашем случае именно эта операция является узким местом.

Давайте предположим, что вы уже выделили проблемный код со сложным преобразованием данных в отдельную функцию, которая выглядит так:

```
my_data_prep <- function() {
  df <- read.csv("path/to/file.csv")
  df %>%
    filter(!not_important) %>%
    group_by(my_variable) %>%
    some_slow_function()
}
```

Сейчас вы вызываете ее из серверной функции следующим образом:

```
server <- function(input, output, session) {
  df <- my_data_prep()
  # Остальной код
}
```

Серверная функция вызывается каждый раз при запуске новой сессии, но данные на сервере остаются прежними, так что вы вполне можете повысить производительность приложения, вынеся функцию по обработке информации за пределы функции `server()`:

```
df <- my_data_prep()
server <- function(input, output, session) {
  # Остальной код
}
```

Поскольку мы обратились к коду загрузки данных, можно сначала попытаться его оптимизировать:

- если вы работаете с *простым неструктурированным файлом* (flat file), попробуйте использовать функции `data.table::fread()` или `vroom::vroom()` вместо `read.csv()` или `read.table()`;
- если у вас есть датафрейм, попробуйте записывать и считывать его при помощи функций `arrow::write_feather()` и `arrow::read_feather()` соответственно. *Feather* представляет собой двоичный формат файла, на чтение и запись которого может потребоваться значительно меньше времени¹;
- если вы работаете не с датафреймами, можете опробовать в действии функции `qs::qread()` / `qs::qsave()` вместо `readRDS()` / `saveRDS()`.

Если этих улучшений оказалось недостаточно для ускорения работы приложения, вы можете настроить планировщик задач на вызов функции `my_data_prep()` по расписанию. В результате ваше приложение сможет пользоваться заранее подготовленными данными, не затрачивая лишнее время на преобразования. Если продолжать аналогию с рестораном, то это решение похоже на принятие на работу помощника шеф-повара, который приходит на работу ночью, когда нет посетителей, и делает все необходимые заготовки, чтобы днем его старший коллега не тратил на это свое драгоценное время.

Управление ожиданиями пользователя

Напоследок хочется рассказать о нескольких трюках, которые помогут создать у пользователя ощущение быстро работающего приложения. Приведем четыре приема, которые вы можете использовать в приложениях по своему усмотрению:

¹ Сравнение с другими форматами файлов приведено в статье по адресу <https://ursalabs.org/blog/2020-feather-v2>.

- разбивайте свои приложения на вкладки при помощи функции `tab-setPanel()`. Пересчитываться данные будут только на открытой в конкретный момент вкладке у пользователя, что позволит сосредоточить внимание на важных для него расчетах;
- длительные операции запускайте по нажатию на кнопку. Кроме того, в процессе выполнения операции оповещайте пользователя о происходящем при помощи уведомлений или индикатора прогресса, о которых мы рассказывали в главе 8. В конце концов, наличие таких индикаторов действительно помогает создать ощущение более быстрой работы приложения, о чем можно почитать в статье по адресу <https://www.nngroup.com/articles/progress-indicators>;
- если приложению требуется выполнить какую-то длительную работу в момент открытия (и вы не можете оптимизировать эту задачу посредством предварительной обработки данных), постарайтесь сделать так, чтобы пользователь видел интерфейс и понимал, что ему необходимо немного подождать;
- наконец, если вы хотите, чтобы ваше приложение отвечало на запросы в процессе выполнения длительной операции, значит, пришло время для изучения принципов асинхронного программирования, и начать можно отсюда: <https://rstudio.github.io/promises/index.html>.

ЗАКЛЮЧЕНИЕ

В данной главе вы научились оценивать производительность приложений Shiny и предпринимать меры по ее повышению. Вы узнали, как можно применять пакет *shinyloadtest* для измерения эффективности кода, а инструмент *shinycannon* позволил вам имитировать одновременную работу с приложением нескольких пользователей. Помимо этого, вы научились использовать пакет *profvis* для нахождения узких мест в коде и применять различные методики для повышения производительности приложения в целом.

Это была заключительная глава данной книги, и я очень благодарен вам за то, что дочитали ее до конца. Надеюсь, вам удалось освоить важные навыки для разработки полноценных приложений Shiny. Мне было бы очень полезно услышать ваши отзывы о книге и предложения для будущих улучшений. Меня всегда можно найти в Twitter: @hadleywickham и на GitHub: <https://github.com/hadley/mastering-shiny>. Еще раз благодарю вас за чтение и желаю успехов в разработке приложений Shiny!

Предметный указатель

Символы

%<-%, 301
%||%, 197
.data, 216
.env, 216
@importFrom, 323
.Rbuildignore, 324
.Rprofile, 321

A

across(), 224
actionButton(), 42, 70
actionLink(), 42
all_of(), 224
any_of(), 224
app\$click(), 339
app\$sendKeys(), 339
arrow::read_feather(), 367
arrow::write_feather(), 367

B

base_font, 125
bg, 125
bindCache(), 362
bookmarkButton(), 207
Bootstrap, 43, 123
bootswatch, 124
brush, 134
brushedPoints(), 134
brushOpts(), 135
bslib, 124
bslib::bs_theme(), 124
bslib::bs_theme_preview, 125

C

callr, 173
checkboxGroupInput(), 40
checkboxInput(), 41
choiceNames, 39
choiceValues, 39
class, 42
clickOpts(), 133
column(), 118
config, 274

D

data.table::fread(), 367
dataTableOutput(), 45
dateInput(), 38
dateRangeInput(), 38
datesdisabled, 38
daysofweekdisabled, 38
dblclickOpts(), 133
dbplyr, 349
DESCRIPTION, 315
devtools::load_all(), 316
devtools::test(), 328
devtools::test_coverage(), 332
devtools::test_coverage_file(), 332
devtools::test_file(), 328
do.call(), 307
downloadButton(), 48
downloadButton(id), 169
downloadHandler(), 169
downloadLink(), 48
downloadLink(id), 169
dput(), 107

E

eval(), 347
eventReactiveHandler(), 359
expect_, 325
expect_equal(), 328
expect_error(), 328, 329
expect_false(), 329
expect_length(), 329
expect_mapequal(), 329
expect_message(), 330
expect_named(), 329
expect_null(), 329
expect_setequal(), 329
expect_snapshot_file(), 343
expect_true(), 329
expect_warning(), 330
exportTestValues(), 339

F

feedback(), 145
feedbackDanger(), 145

feedbackSuccess(), 145
feedbackWarning(), 145
fg, 125
fileInput(), 41, 166
file.mtime(), 258
fillPage(), 116
fixedPage(), 115
fluidPage(), 27, 115
fluidRow(), 118
forcats, 83
freezeReactiveValue(), 185

G

ggplot2::aes_string(), 348
Git, 275
glue, 348
glue::glue(), 104
glue::glue_safe(), 348
glue::glue_sql(), 349
glue(), 348

H

h1(), 128
hoverOpts(), 133
HTML(), 127
htmltools::htmlDependency(), 127

I

incProgress(), 155
input, 51
inputId, 36
invalidateLater(), 257
isolate(), 255

J

janitor, 174

L

lapply(), 196, 308
load_runs(), 354

M

mainPanel(), 116
map_chr(), 197
memoise, 362
message(), 104
modalDialog(), 161
moduleServer(), 289

N

navbarMenu(), 122
navbarPage(), 122
navlistPanel(), 121
nearPoints(), 131
NS(), 289

numericInput(), 37

O

observe(), 253
observeEvent(), 73, 164, 253
on.exit(), 153, 252
output, 52
outputOptions(), 254

P

p(), 128
parse(), 347
passwordInput(), 36
pkgload, 322
plotOutput(), 47, 129
print(), 104
profvis, 356, 358
profvis::pause(), 360
profvis::profvis(), 358
purrr::map(), 196
purrr::pmap(), 282
purrr::reduce(), 196

Q

qs::qread(), 367
qs::qsave(), 367

R

R6, 231
R CMD check, 323
radioButtons(), 39, 303
reactive, 30
reactive(), 65
reactiveConsole(), 232
reactivePoll(), 258
reactiveTimer(), 69
reactiveVal(), 135, 232, 249
reactiveValues(), 249
reactlog, 247
read.csv(), 367
readRDS(), 367
read.table(), 367
record_session(), 353
Reduce(), 196
removeModal(), 162
renderCachedPlot(), 362
renderDataTable(), 45
renderImage(), 141
renderPlot(), 47, 364
renderPrint(), 45
renderTable(), 45
renderText(), 45
renderUI(), 194
renv, 274

repeatable(), 211
 req(), 131, 147
 rmarkdown::render(), 172, 348
 RMarkdown, 172
 roxygen2, 315
 rsconnect, 322
 rsconnect::deployApp(), 322
 RStudio, 18
 RStudio Gadgets, 321

S

saveRDS(), 367
 selectInput(), 27, 39
 server(), 25
 session\$flushReact(), 336
 session\$getReturned(), 337
 setBookmarkExclude(), 211
 setNames(), 82
 shiny::reactlogShow(), 247
 Shiny, 15
 shinyApp(), 25
 shiniycannon, 353
 shinycssloaders, 161
 ShinyDriver, 338
 shinyFeedback, 145
 shinyloadtest, 352
 shinyloadtest::record_session(), 352
 shinyloadtest::report(), 352
 shinymeta, 174
 shinytest, 338
 showModal(), 162
 showNotification(), 152
 sidebarLayout(), 116
 sidebarPanel(), 116
 sliderInput(), 37
 stopifnot(), 297
 str(), 105
 suspendWhenHidden, 254
 switch(), 217
 Sys.Date(), 364
 Sys.sleep(), 156, 360

T

tableOutput(), 27, 45
 tabPanel(), 119
 tabsetPanel(), 119
 tagList(), 289
 tags, 128
 testthat, 325
 testthat::snapshot_accept(), 331
 testthat::snapshot_review(), 342
 textAreaInput(), 37
 textInput(), 36

textOutput(), 44
 thematic, 125
 thematic_shiny(), 125
 tidy eval, 213
 tidy-selection, 223
 titlePanel(), 116
 tools::file_ext(), 168
 traceback(), 96
 try(), 252
 tryCatch(), 252
 t.test(), 177

U

ui, 51
 UI, 24
 uiOutput(), 194
 updateSliderInput(), 180
 updateTextInput(), 180
 use_license_, 323
 useShinyFeedback(), 145
 usethis, 321
 usethis::edit_r_profile(), 321
 usethis::use_build_ignore(), 324
 usethis::use_data(), 324
 usethis::use_description(), 316
 usethis::use_package(), 322
 usethis::use_proprietary_license(), 323
 usethis::use_rstudio(), 316
 usethis::use_test(), 327, 328
 use_waitress(), 157
 utils::Rprof(), 356

V

validate(), 151
 verbatimTextOutput(), 27, 44
 vroom::vroom(), 367

W

waiter, 157
 withProgress(), 155

Z

zeallot, 301

A

Асинхронность, 352

Б

Безопасность, 345
 Бэкенд, 23

В

Веб-API, 362
 Вертикальное масштабирование, 351
 Воспроизводимое окружение, 274

Воспроизводимый пример, 91, 106
Вращающийся индикатор прогресса, 158
Выделение прямоугольной области, 134

Г

Горизонтальное масштабирование, 351

Д

Датафрейм, 45
Дебаунсинг, 54
Декларативное программирование, 55
Диалоговые окна, 203
Динамизм, 245
Динамическая фильтрация, 198

З

Загрузка файлов, 166
Запуск приложения, 25

И

Императивное программирование, 55
Индикатор прогресса, 155
Интегрированная среда разработки, 18
Интерактивные графики, 129
Интерактивный отладчик, 98
Интерфейсная функция модуля, 289

К

Кеширование, 361
Ключ кеширования, 362, 365
Корзина, 165
Косвенная адресация, 214

Л

Ленивые вычисления, 55

М

Макет, 114
Маскирование данных, 215
Мастер, 192
Мемоизация, 68
Многостраничное приложение, 119
Модуль, 66, 286
Модульность, 66

Н

Наблюдатель, 73, 253
Набор вкладок, 189
Набор инструментов пользовательского интерфейса, 17
Непрерывная интеграция, 275

О

Объект R6, 157
Огненный график, 356
Оповещение, 152

Отладка, 91, 95
Оценка производительности, 352

П

Пакеты, 315
Передача файлов, 166
Переменные данных, 214
Переменные окружения, 214
План тестирования, 273
Покрытие кода, 332
Ползунок, 37
Пользовательский интерфейс, 35
Порядок выполнения кода, 57
Приложение Shiny, 24
Принт-отладка, 104
Проект, 92
Производительность, 350
Пространство имен, 286
Профилирование, 356

Р

Рабочий процесс, 91
Радиокнопки, 39
Развертывание, 322
Реактивное значение, 135, 232, 249
Реактивное окружение, 67
Реактивное программирование, 50, 229
Реактивность, 29
Реактивные выражения, 30, 57
Реактивный график, 56, 235
Реактивный контекст, 52
Реактивный лог, 247
Реактивный поставщик, 60, 235
Реактивный потребитель, 60, 235
Резервная ячейка, 165

С

Связь, 241
Серверная функция модуля, 289
Система контроля версий, 275
Скачивание файлов, 169
Снимочное тестирование, 330
Сниппет, 25, 92
Событийно-ориентированное программирование, 230
Создание реактивного выражения, 65
Ссылочная семантика, 250
Стандартный вывод, 104
Стандартный вывод ошибок, 104
Стек вызовов, 95, 357

Т

Тема, 124
Тестирование, 325

Топологическая сортировка, 58

Точка останова, 99

Трассировка, 95

Ф

Фаза инвалидации, 241

Флажки, 40

Фронтенд, 23

Функции ожидания, 325

Функции отображения, 28

Функции семейства update, 179

Функция server, 24

Ц

Циклические ссылки, 186

Цикл разработки, 91

Э

Элементы ввода, 36

Элементы вывода, 43, 253

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@aliants-kniga.ru**.

При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Хэдли Уикем

Изучаем Shiny

Главный редактор	<i>Мовчан Д. А.</i> dmkpress@gmail.com
Зам. главного редактора	<i>Сенченкова Е. А.</i>
Перевод	<i>Гинько А. Ю.</i>
Корректор	<i>Синяева Г. И.</i>
Верстка	<i>Чаннова А. А.</i>
Дизайн обложки	<i>Мовчан А. Г.</i>

Гарнитура PT Serif. Печать цифровая.

Усл. печ. л. 30,39. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

Изучите веб-фреймворк Shiny и выведите свои навыки владения языком программирования R на новый уровень. Оставьте в прошлом статистические отчеты — с Shiny вы сможете создавать полностью интерактивные веб-приложения для анализа данных. Пользователи смогут легко перемещаться между наборами данных, создавать и исследовать подмножества, выборки и срезы, запускать модели с нужными им значениями параметров, разрабатывать собственные визуализации и многое другое.

Хэдли Уикем из RStudio покажет аналитикам данных, статистикам и научным исследователям, не обладающим глубокими познаниями в области HTML, CSS и JavaScript, как создавать мощные веб-приложения на языке R. Книга, которую вы держите в руках, является полноценным руководством по фреймворку Shiny, который поможет вам из новичка в этой области стать настоящим экспертом и писать масштабные, сложные и эффективные приложения.

«Эта книга является полным руководством по библиотеке Shiny. Ее можно использовать для быстрого старта, но также в ней приведено подробное описание более сложных концепций, таких как реактивное программирование и модули. Читая это руководство, вы не просто научитесь работать с Shiny, но и обретете глубокое понимание того, почему все работает так, а не иначе».

*Джо Ченг,
главный технический директор
RStudio и создатель Shiny*

- **Приступаем к работе:** собираем из частей целое — основы построения приложения Shiny.
- **Shiny в действии:** исследуем функционал Shiny на примере листингов, приложений и полезных техник.
- **Осваиваем реактивность:** погружаемся в теорию и практику реактивного программирования и исследуем реактивные графические компоненты.
- **Эффективные приемы:** рассматриваем применение оптимальных техник и повышаем эффективность приложений Shiny.



Хэдли Уикем является ведущим научным сотрудником в RStudio, лауреатом премии COPSS Presidents' Award 2019 года и членом организации R Foundation. Он занимается разработкой вычислительных инструментов, призванных облегчить и ускорить науку о данных, а также сделать ее более занимательной. В активе Хэдли множество пакетов для научной работы с данными. Также он пишет книги, преподает и выступает на конференциях, способствуя применению языка R в обработке данных.

Подробнее можно узнать на его сайте hadley.nz.

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК «Галактика»
books@aliants-kniga.ru


www.dmk.рф

ISBN 978-5-97060-964-4



9 785970 609644 >