

Rust

Профессиональное
программирование

Бренден Мэтьюз



MANNING

olit

Code Like a Pro in Rust

BRENDEN MATTHEWS



MANNING
SHELTER ISLAND

Бренден Мэтьюз

Rust

Профессиональное программирование

Астана
«АЛИСТ»
2025

УДК 004.4
ББК 32.973.26-02
М97

Мэттьюз Б.

М97 Rust. Профессиональное программирование: Пер. с англ. — Астана:
АЛИСТ, 2025. — 320 с.: ил.
ISBN 978-601-08-4833-7

Книга служит введением в продвинутые темы, необходимые для реализации полнофункциональных проектов на языке Rust. Rust рассматривается как сравнительно новый, но мощный и зрелый язык для серверного программирования. Рассмотрены паттерны проектирования, характерные для Rust, роль Rust в современном низкоуровневом программировании, приемы асинхронных взаимодействий и управление памятью. Проиллюстрированы способы создания HTTP REST API на Rust, интеграция кода Rust с кодом на других языках, типичные идиомы и структуры данных, применяемые при профессиональной работе с Rust.

Для Rust-разработчиков и специалистов по системному программированию

УДК 004.4
ББК 32.973.26-02

© 2025 ALIST LLP

Authorized translation of the English edition © 2024 Manning Publications. This translation is published and sold by permission of Manning Publications, the owner of all rights to publish and sell the same.

Авторизованный перевод английской редакции книги © 2024 Manning Publications. Перевод опубликован и продается с разрешения компании-правообладателя Manning Publications. Все права защищены.

ISBN 978-1-61729-964-3 (англ.)
ISBN 978-601-08-4833-7 (каз.)

© Manning Publications, 2024
© Издание на русском языке. ТОО "АЛИСТ", 2025

Краткое оглавление

Оглавление	7
Предисловие	13
Благодарности	15
Об этой книге	17
Об авторе	21
Об иллюстрации на обложке	23
Глава 1. Почувствуйте Rust!	25
ЧАСТЬ I. РАБОТА С RUST НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ	35
Глава 2. Управление проектами с помощью Cargo	37
Глава 3. Инструменты Rust	77
ЧАСТЬ II. ОСНОВНЫЕ ДАННЫЕ	101
Глава 4. Структуры данных	103
Глава 5. Работа с памятью	137
ЧАСТЬ III. КОРРЕКТНОСТЬ КОДА	173
Глава 6. Модульное тестирование	175
Глава 7. Интеграционное тестирование	199
ЧАСТЬ IV. АСИНХРОННЫЙ RUST	217
Глава 8. Асинхронное программирование в Rust	219
Глава 9. Создание сервиса HTTP REST API	251
Глава 10. Создание CLI-инструмента HTTP REST API	279
ЧАСТЬ V. ОПТИМИЗАЦИЯ	295
Глава 11. Оптимизация кода	297
Приложение	313
Предметный указатель	317

Оглавление

Краткое оглавление	5
Предисловие	13
Благодарности.....	15
Об этой книге	17
В чем особенность этой книги?.....	17
Для кого она предназначена?	17
Структура книги.....	18
О программном коде	19
Дискуссионный форум liveBook	20
Об авторе.....	21
Об иллюстрации на обложке.....	23
Глава 1. Почувствуйте Rust!	25
1.1. Так чем же примечателен Rust?	26
1.2. В чем же уникальность Rust?.....	28
1.2.1. Rust безопасен	28
1.2.2. Rust современен	30
1.2.3. У Rust абсолютно открытый исходный код	30
1.2.4. Сравнение Rust с другими популярными языками	31
1.3. В каких случаях стоит воспользоваться языком Rust?	32
1.4. Требуемые инструменты.....	33
Резюме	34
ЧАСТЬ I. РАБОТА С RUST НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ	35
Глава 2. Управление проектами с помощью Cargo	37
2.1. Ознакомительный тур по Cargo.....	38
2.1.1. Основное применение	38
2.1.2. Создание нового приложения или библиотеки	39
2.1.3. Компиляция, запуск и тестирование	41
2.1.4. Переключения между наборами инструментов	42
2.2. Управление зависимостями	43
2.3. Фича-флаги.....	46

2.4. Корректировка зависимостей	49
2.4.1. Косвенные зависимости	51
2.4.2. Лучшие методы корректировки зависимостей	51
2.5. Публикация крейтов	52
2.5.1. CI/CD-интеграция	52
2.6. Ссылки на библиотеки C	56
2.7. Бинарный дистрибутив	59
2.7.1. Кросс-компиляция	59
2.7.2. Создание статически связанных бинарных файлов	60
2.8. Документирование Rust-проектов	62
2.8.1. Примеры кода в документации	65
2.9. Модули	66
2.10. Рабочие пространства	69
2.11. Пользовательские сценарии сборки	71
2.12. Проекты Rust во встраиваемых средах	73
Резюме	75

Глава 3. Инструменты Rust.....	77
3.1. Общий обзор инструментария Rust	78
3.2. IDE-интеграция Rust: инструмент <i>rust-analyzer</i>	79
3.2.1. Установка <i>rust-analyzer</i>	79
3.2.2. Магические завершения	80
3.3. Поддержка аккуратности кода: инструмент <i>rustfmt</i>	83
3.3.1. Установка <i>rustfmt</i>	83
3.3.2. Конфигурирование <i>rustfmt</i>	84
3.4. Повышение качества кода: инструмент <i>Clippy</i>	85
3.4.1. Установка <i>Clippy</i>	86
3.4.2. <i>Clippy</i> -линты	86
3.4.3. Конфигурирование <i>Clippy</i>	88
3.4.4. Автоматическое применение предложений <i>Clippy</i>	89
3.4.5. Использование <i>Clippy</i> в CI/CD	89
3.5. Сокращение времени компиляции: инструмент <i>sccache</i>	90
3.5.1. Установка <i>sccache</i>	91
3.5.2. Конфигурирование <i>sccache</i>	91
3.6. Интеграция Rust с IDE-средами, включая Visual Studio Code	91
3.7. Использование наборов инструментов: стабильные и ночные версии	92
3.7.1. Функции, доступные только в ночной версии	93
3.7.2. Использование nightly-функций в публикуемых крейтах	94
3.8. Дополнительные инструменты: <i>cargo-update</i> , <i>cargo-expand</i> , <i>cargo-fuzz</i> , <i>cargo-watch</i> , <i>cargo-tree</i>	94
3.8.1. Поддержание пакетов в актуальном состоянии: инструмент <i>cargo-update</i>	95
3.8.2. Отладка макросов: инструмент <i>cargo-expand</i>	95
3.8.3. Тестирование с применением <i>libFuzzer</i>	96
3.8.4. Периодический запуск Cargo-команд: инструмент <i>cargo-watch</i>	97
3.8.5. Проверка зависимостей: инструмент <i>cargo-tree</i>	97
Резюме	99

ЧАСТЬ II. ОСНОВНЫЕ ДАННЫЕ.....	101
Глава 4. Структуры данных.....	103
4.1. Строковые типы <i>String</i> , <i>str</i> , <i>&str</i> и <i>&'static str</i>	104
4.1.1. Сравнение <i>String</i> и <i>str</i>	104
4.1.2. Эффективное применение строк	105
4.2. Что такое слайсы и массивы?	108
4.3. Векторы	111
4.3.1. Более глубокое погружение в <i>Vec</i>	111
4.3.2. Обертывание векторов	113
4.3.3. Типы, связанные с векторами	113
4.4. Отображения	114
4.4.1. Пользовательские функции хеширования	115
4.4.2. Создание хешируемых типов	116
4.5. Типы Rust: примитивы, структуры, перечисления и псевдонимы	117
4.5.1. Применение примитивных типов	117
Целочисленные типы	118
Размерные типы	119
Арифметика на примитивных типах	119
4.5.2. Использование кортежей	121
4.5.3. Применение структур	122
4.5.4. Применение перечислений	125
4.5.5. Применение псевдонимов	128
4.6. Обработка ошибок с помощью <i>Result</i>	129
4.7. Преобразование типов с помощью <i>From/Into</i>	130
4.7.1. Типажи <i>TryFrom</i> и <i>TryInto</i>	132
4.7.2. Наиболее рациональные приемы преобразования типов с использованием <i>From</i> и <i>Into</i>	133
4.8. Обеспечение совместимости интерфейса внешних функций с типами Rust	133
Резюме	135
Глава 5. Работа с памятью	137
5.1. Управление памятью: куча и стек	137
5.2. Представление о владении: копирование, заимствование, ссылки и перемещения	140
5.3. Глубокое копирование	142
5.4. Предотвращение копирования	144
5.5. Умные указатели: тип <i>Box</i>	146
5.6. Подсчет ссылок	151
5.7. Клонирование при записи	155
5.8. Пользовательские распределители	159
5.8.1. Создание пользовательского распределителя	160
5.8.2. Создание пользовательского распределителя для защищенной памяти	163
5.9. Кратко об умных указателях	169
Резюме	170
ЧАСТЬ III. КОРРЕКТНОСТЬ КОДА	173
Глава 6. Модульное тестирование	175
6.1. Чем примечательно тестирование в Rust?	175
6.2. Встроенные функции тестирования	177

6.3. Среды тестирования	179
6.4. Компилятор лучше вас знает, что не нужно тестировать.....	184
6.5. Работа с особыми случаями параллельного тестирования и глобального состояния.....	185
6.6. Размышления о реструктуризации	191
6.7. Инструменты реструктуризации	191
6.7.1. Переформатирование	192
6.7.2. Переименование	192
6.7.3. Перемещение	194
6.7.4. Переписывание	194
6.8. Охват кода	196
6.9. Работа с меняющейся экосистемой.....	198
Резюме	198

Глава 7. Интеграционное тестирование	199
7.1. Сравнение интеграционного и модульного тестирования	200
7.2. Стратегии интеграционного тестирования	203
7.3. Сравнение встроенного и внешнего интеграционного тестирования.....	205
7.4. Библиотеки и инструменты для проведения интеграционного тестирования.....	206
7.4.1. Использование <i>assert_cmd</i> для тестирования CLI-приложений	206
7.4.2. Использование с интеграционными тестами крейта <i>proptest</i>	210
7.4.3. Другие инструменты интеграционного тестирования.....	211
7.5. Fuzz-тестирование	211
Резюме	216

ЧАСТЬ IV. АСИНХРОННЫЙ RUST 217

Глава 8. Асинхронное программирование в Rust.....	219
8.1. Среды выполнения	221
8.2. Асинхронное мышление	222
8.3. Фьючерсы: обработка результатов выполнения асинхронных задач.....	224
8.3.1. Режим бездействия	224
8.3.2. Определение среды выполнения с помощью <code>#[tokio::main]</code>	227
8.4. Ключевые слова <i>async</i> и <i>.await</i> : когда и где их использовать?	227
8.5. Конкурентность и параллелизм с <i>async</i>	230
8.6. Реализация асинхронного наблюдателя.....	234
8.7. Смешивание синхронного и асинхронного кода	240
8.8. Когда не стоит применять асинхронность?	243
8.9. Трассировка и отладка асинхронного кода	243
8.10. Работа с асинхронностью при тестировании	247
Резюме	248

Глава 9. Создание сервиса HTTP REST API.....	251
9.1. Выбор веб-фреймворка	252
9.2. Построение архитектуры	253
9.3. Проектирование API.....	254
9.4. Библиотеки и инструменты.....	255
9.5. Создание шаблонов приложений	257
9.5.1. Функция <i>main()</i>	257
9.5.2. Инициализация трассировки: <i>init_tracing()</i>	259
9.5.3. Инициализация пула базы данных: <i>init_dbpool()</i>	260

9.6. Моделирование данных	262
9.6.1. SQL-схема.....	262
9.6.2. Взаимодействие с нашими данными.....	263
9.7. Объявление API-маршрутов	267
9.8. Реализация API-маршрутов	269
9.9. Обработка ошибок.....	272
9.10. Запуск сервиса	273
Резюме	278
Глава 10. Создание CLI-инструмента HTTP REST API.....	279
10.1. Выбор используемых инструментов и библиотек	280
10.2. Проектирование CLI.....	281
10.3. Объявление команд	282
10.4. Реализация команд	285
10.5. Реализация запросов	287
10.6. Надлежащая обработка ошибок	289
10.7. Тестирование нашего CLI	289
Резюме	293
ЧАСТЬ V. ОПТИМИЗАЦИЯ.....	295
Глава 11. Оптимизация кода	297
11.1. Абстракции с нулевой стоимостью	297
11.2. Векторы	299
11.2.1. Выделение памяти для вектора	299
11.2.2. Итераторы векторов.....	301
11.2.3. Быстрое копирование с помощью <i>Vec</i> и слайсов	303
11.3. Применение возможностей SIMD	305
11.4. Распараллеливание с применением Rayon	307
11.5. Использование Rust для ускорения программ на других языках	309
11.6. Что делать дальше?	311
Резюме	312
Приложение	313
Установка инструментов для примеров, приводимых в книге	313
Установка инструментов на macOS с помощью Homebrew.....	313
Установка инструментов на системах под управлением Linux	313
Установка <i>rustup</i> на Linux- или UNIX-системах	313
Установка инструментов на системах под управлением Windows.....	314
Управление <i>rustc</i> и другими Rust-компонентами с помощью <i>rustup</i>	314
Установка <i>rustc</i> и других компонентов	314
Переключение исходных наборов инструментов с помощью <i>rustup</i>	315
Обновление Rust-компонентов	315
Установка HTTPie	315
Предметный указатель	317

Предисловие

Мне нравится изучать новые языки программирования.

Созданием программ я занимаюсь довольно давно, но все же иногда упорно заставляю себя осваивать новые языки или инструменты. Язык Rust уникален во многих отношениях — в нем вводится целый ряд концепций, возможно, ранее кому-то, даже тем, кто уже давно в профессии программиста, еще вообще не встречавшихся.

Я потратил уйму времени на профессиональную работу с Rust, а также на участие в проектах сообщества программистов на Rust и написал эту книгу, чтобы поделиться тем, чему я уже научился по ходу дела. Не пожалев сил на прочтение этой книги, вы сможете сэкономить свое время, избегая распространенных ошибок и справляясь с проблемами, возникающими у всех начинающих работать с языком Rust.

Благодарности

Я хотел бы поблагодарить моих друзей Джавида Шейха (Javeed Shaikh) и Бена Лина (Ben Lin) за отзывы о ранних черновиках рукописи, а также издательство Manning Publications за терпеливую работу со мной над завершением этой книги.

Особую благодарность хочу выразить главному редактору Карен Миллер (Karen Miller), редактору-рецензенту Александру Драгосавлевичу (Aleksandar Dragosavljević), выпускающему редактору Дейдре Хайам (Deirdre Hiam), литературному редактору Кристиану Берку (Christian Berk) и корректору Кэти Теннант (Katie Tennant).

Я благодарю всех рецензентов: Адама Венделла (Adam Wendell), Александро Кампейса (Alessandro Campeis), Аруна Бхагвана Коммади (Arun Bhagvan Kommadi), Кристиана Уиттса (Christian Witts), Клиффорда Тербера (Clifford Thurber), Дэвида Мошала (David Moshal), Дэвида Паку (David Paccoud), Джанлуиджи Спаньоло (Gianluigi Spagnuolo), Хораси Масиаса (Horaci Macias), Жауме Лопеса (Jaume Lopez), Жана-Поля Малерба (Jean-Paul Malherbe), Жуана Педро де Ласерда (João Pedro de Lacerda), Джона Риддла (Jon Riddle), Джозефа Пачо (Joseph Pachod), Жюльена Кастелена (Julien Castelain), Кевина Оппа (Kevin Orr), Мадхава Аяягари (Madhav Ayyagari), Мартину Новака (Martin Nowack), Мэтта Сармьенто (Matt Sarmiento), Мэттью Уинтера (Matthew Winter), Маттиаса Буша (Matthias Busch), П. К. Четти (PK Chetty), Рохита Госвами (Rohit Goswami), Сатадру Роя (Satadru Roy), Сатеджа Кумара Сахи (Satej Kumar Sahu), Себастьяна Пальму (Sebastian Palma), Сета Макферсона (Seth MacPherson), Саймона Чёке (Simon Tschöke), Шри Кадимисетти (Sri Kadimisetty), Тима ван Дёрзена (Tim van Deurzen), Уильяма Уиллера (William Wheeler) и Зака Питерса (Zach Peters) — ваши предложения помогли сделать эту книгу лучше.

Особое внимание при работе над книгой уделялось вопросам возникновения потребностей во внесении изменений в функции или интерфейсы. Хотя основные функции языка могут и не подвергаться существенным изменениям, практическое применение Rust сопряжено с возможным включением сотен отдельно взятых библиотек и проектов. Чтение книги позволит вам ознакомиться со стратегиями и методами, помогающими сориентироваться в постоянно развивающейся экосистеме языка.

Следует заметить, что пока вы будете читать эту книгу, экосистема Rust продолжит свое бурное развитие. Работа над книгой шла с прицелом на будущее языка Rust, но я не в состоянии гарантировать, что после ее публикации не произойдет никакого существенного изменения языка и используемых в нем библиотек.

В чем особенность этой книги?

Основное внимание в книге сконцентрировано на практическом применении Rust, при этом в ней затрагиваются темы общего характера, описываются имеющиеся в Rust ограничения и его инструментарий, а также предлагаются способы быстрого освоения разработчиками приемов высокой продуктивности при создании программ на языке Rust. В тексте книги вы не найдете традиционного описания языка Rust, она также не заменит вам официальную документацию по этому языку. Книга станет вам лишь дополнением к существующей документации, опишет ресурсы, доступные для работы с Rust, а также предоставит наиболее важные сведения о нем, которые вы вряд ли найдете где-либо в одном месте в документации по Rust. Хотя книга и не является исчерпывающим справочником по языку Rust, в ней при необходимости указывается, куда следует обращаться за дополнительной информацией.

Для кого она предназначена?

Читатели книги должны быть в определенной степени уже знакомы с Rust, будучи уже достаточно компетентными или хотя бы начинающими программистами на этом языке. При полном отсутствии знакомства с Rust читать эту книгу вам будет сложно, поскольку в ней содержится много отсылок к специфичным для Rust возможностям практически без раскрытия их нюансов. Если вы запутаетесь в синтак-

сисе Rust или каких-либо его технических деталях, я рекомендую начать освоение языка с книги Тима Макнамары «Rust в действии»¹ (Tim McNamara, *Rust in Action*, Manning, 2021) или подробной документации по этому языку, публикуемой сообществом программистов на Rust².

Главы книги, в зависимости от того, о чем вы конкретно хотели бы узнать, можно читать в любом порядке. Было бы, конечно, неплохо, если бы вы прочитали все ее главы от начала до конца, но я понимаю, что у читателей могут быть разные цели и уровни опыта. Большинство всех глав книги основываются на содержании предыдущих, поэтому, хотя это и не обязательно, рекомендуется всё же прочитать главы книги в порядке их следования — именно так вам удастся извлечь из нее максимальную пользу.

Опытным программистам, особенно тем, кто уже работал на Rust, часть материала книги может показаться знакомой — и они могут сразу перейти к тем главам, которые представляют для них наибольший интерес. Новичкам же в программировании в целом или конкретно на языке Rust я бы однозначно посоветовал прочитать всю книгу от корки до корки.

По мере необходимости в книге встречаются ссылки на те или иные главы или темы, связанные с текущим изложением материала, — переходя по ним, вы сможете более детально ознакомиться с изучаемой темой.

Структура книги

- *Глава 1* выступает здесь в роли своего рода введения — в ней дается общий обзор языка Rust и всего, что делает его особенным.
- *Часть I* посвящена знакомству с основами Rust и его инструментами:
 - в *главе 2* представлен Cargo — инструмент управления Rust-проектами;
 - в *главе 3* приводится обзор основных инструментов Rust.
- В *части II* описываются структуры данных Rust и управление памятью:
 - в *главе 4* рассматриваются структуры данных Rust;
 - в *главе 5* подробно описана модель управления памятью Rust.
- В *части III* внимание уделяется методам тестирования на корректность:
 - в *главе 6* представлен обзор функций модульного тестирования Rust;
 - в *главе 7* рассматриваются вопросы интеграционного и нечеткого тестирования.
- *Часть IV* знакомит читателей с асинхронным программированием в Rust:
 - в *главе 8* содержится обзор возможностей проведения в Rust асинхронных операций;

¹ См. <https://bhv.ru/product/rust-v-dejstvii/>.

² См. <https://doc.rust-lang.org/book/>.

- в *главе 9* рассматривается реализация асинхронного HTTP-сервера;
 - в *главе 10* представлена реализация асинхронного HTTP-клиента.
- *Часть V* посвящена вопросам оптимизации — в *главе 11*, единственной главе этой части, дается углубленное представление подробностей оптимизации программ, созданных на языке Rust.
- В *приложении* приводятся инструкции по установке утилит командной строки, необходимых для компиляции и запуска приводимых в книге примеров кода.

О программном коде

В книге содержится множество примеров исходного кода — как непосредственно в виде строк кода в самом тексте, так и в нумерованных листингах. В обоих случаях код форматируется таким вот образом, чтобы его можно было отличить от обычного текста. Иногда код также выделяется *полужирным* шрифтом — им оформляются его измененные фрагменты, например при добавлении к уже имевшейся строке кода новых функциональных возможностей.

Во многих случаях — чтобы исходный код помещался на странице книги — его строки подвергались специальной перекомпоновке: к ним добавлялись разрывы строк и делались необходимые отступы. Но и этого иногда бывало недостаточно, и в листинги включались маркеры переноса строки: ➔. Кроме того, из исходного кода листингов часто удалялись комментарии — в тех случаях, когда описание элементов кода приводится в окружающем листинг тексте. Многие листинги также сопровождаются аннотациями кода, подчеркивающими его важные концепции.

Исполняемые фрагменты кода вы можете извлечь из liveBook (онлайн-версии этой книги)³ простым копированием. Полный код примеров, включенных в книгу, доступен для загрузки с веб-сайта издательства Manning⁴, а также с GitHub⁵. Копия этого кода может быть скачана вами на свой компьютер с помощью следующей Git-команды:

```
$ git clone https://github.com/brndnmthws/code-like-a-pro-in-rust-book
```

Код, приводимый в книге, имеет разрешительную MIT-лицензию, позволяющую вам копировать примеры кода и использовать их по своему усмотрению, даже в качестве основы для вашей собственной работы.

В книге содержится множество ссылок на проекты с открытым исходным кодом, которые могут быть использованы вами в качестве вспомогательных учебных средств. Исходный код для большинства этих проектов (или крейтов) можно получить из их репозиториев (табл. П.1).

³ См. <https://livebook.manning.com/book/code-like-a-pro-in-rust>.

⁴ См. <https://www.manning.com/books/code-like-a-pro-in-rust>.

⁵ См. <https://github.com/brndnmthws/code-like-a-pro-in-rust-book>.

Таблица П.1. Список проектов, на которые даются ссылки в этой книге

Название	Описание	Домашняя страница	URL репозитория
dryoc	Криптографическая библиотека	https://crates.io/crates/dryoc	https://github.com/brndnmthws/dryoc.git
rand	Предоставляет случайные значения	https://rust-random.github.io/book	https://github.com/rust-random/rand.git
Rocket	HTTP/веб-фреймворк	https://rocket.rs	https://github.com/SergioBenitez/Rocket.git
num_cpus	Возвращает количество логических ядер ЦП	https://crates.io/crates/num_cpus	https://github.com/seanmonstar/num_cpus.git
zlib	Библиотека сжатия	https://zlib.net/	https://github.com/madler/zlib.git
lazy_static	Библиотека глобальных статических переменных	https://crates.io/crates/lazy_static	http://mng.bz/E9rD
Tokio	Асинхронные операции времени выполнения	https://tokio.rs	https://github.com/tokio-rs/tokio
Syn	Парсер Rust-кода	https://crates.io/crates/syn	https://github.com/dtolnay/syn
axum	Асинхронный веб-фреймворк	https://docs.rs/axum/latest/axum/	https://github.com/tokio-rs/axum

Дискуссионный форум liveBook

Приобретение этой книги дает вам право бесплатного доступа к онлайн-платформе для чтения книг издательства Manning. Используя эксклюзивные возможности обсуждения, предоставляемые этой платформой, вы можете оставлять там комментарии к книге в целом или к определенным ее разделам или абзацам. На форуме liveBook очень просто что-то помечать для себя, задавать технические вопросы и отвечать на них, а также обращаться за помощью к автору книги и другим посетителям. Чтобы получить доступ к форуму⁶, следует зайти на него и зарегистрироваться в открывющейся при этом форме. На сайте liveBook также можно найти дополнительные сведения о форумах Manning и о правилах поведения на них⁷.

Обязательства издательства Manning перед читателями заключаются в предоставлении площадки, где можно вести содержательные диалоги между отдельными читателями и между читателями и автором. Автор не берет на себя обязательства по какому-либо конкретному объему участия со своей стороны, и его участие в работе форума является добровольным (и неоплачиваемым). Но вы можете попробовать задать автору несколько сложных вопросов, чтобы у него не пропал интерес к дискуссии! Форум и архивы предыдущих обсуждений будут доступны на сайте издательства до тех пор, пока книга издается и переиздается.

⁶ См. <https://livebook.manning.com/book/code-like-a-pro-in-rust>.

⁷ См. <https://livebook.manning.com/discussion>.

Об авторе

Бренден Мэттьюз (Brenden Matthews) — программист, предприниматель и весьма активный участник проектов с открытым исходным кодом. Он использует Rust в профессиональных целях с первых дней существования этого языка и внес ощутимый вклад в несколько инструментов Rust и проектов с открытым исходным кодом на нем. Бренден Мэттьюз является автором Conky, популярного системного монитора, и членом Apache Software Foundation с более чем 25-летним опытом работы в этой сфере. Бренден регулярно размещает инструктивные материалы на YouTube и опубликовал множество статей по Rust и другим языкам программирования в различных периодических изданиях. Бренден выступал с докладами на нескольких технологических конференциях, включая QCon, LinuxCon, ContainerCon, MesosCon, All Things Open и Rust meetups. Он более 13 лет участвует в работе GitHub, опубликовал там несколько Rust-крайтов, внес вклад в ряд проектов с открытым исходным кодом на Rust и продолжает создавать профессиональные Rust-приложения промышленного уровня. Бренден также является автором книги «Rust Design Patterns»¹, являющейся продолжением «Code Like a Pro in Rust».

¹ См. [https://www.amazon.co.uk/Rust-Design-Patterns-Code-Rustacean/dp/1633437469/
ref=monarch_sidesheet_title](https://www.amazon.co.uk/Rust-Design-Patterns-Code-Rustacean/dp/1633437469/ref=monarch_sidesheet_title).

Об иллюстрации на обложке

Фигура на обложке книги — *Femme de l'Argou* («Женщина из Ааргау», Швейцария), взятая из коллекции Жака Грассе де Сен-Совера (*Jacques Grasset de Saint-Sauveur*), опубликованной в 1797 году. Каждая иллюстрация в ней тщательно прорисована и раскрашена вручную.

В те дни по одежде было легко определить, где жили люди, чем они занимались или какое положение занимали. Издательство Manning отмечает изобретательность и инициативность компьютерного бизнеса с помощью изображений на книжных обложках, за основу для которых берется богатое разнообразие образцов региональной культуры многовековой давности, возвращенное к жизни фотокопиями из таких коллекций, как эта.

Почувствуйте Rust!

В этой главе:

- несколько вводных слов о книге и о Rust;
- сравнение Rust с другими языками программирования;
- способы извлечения максимальной пользы из книги.

Эта книга поможет начинающим Rust-разработчикам как можно быстрее освоить язык, его инструментарий, структуры данных, управление памятью, тестирование, асинхронное программирование и наиболее эффективные приемы работы с языком. Освоив материал книги, вы должны обрести чувство уверенности при создании серьезных программных систем производственного уровня с идиоматически присущими языку Rust приемами программирования. Книга, конечно, не является полноценным справочником по языку Rust или его инструментарию, и все же основной упор в ней сделан на изложение исключительно высококачественного материала.

Тем, кто собирается создавать быстродействующие и безопасные программы, Rust предлагает целый набор весьма привлекательных функций. Бытует мнение, что обучение языку Rust отличается излишней замысловатостью и интенсивностью, но эта книга поможет вам преодолеть сложности, разъяснит основные концепции языка и предоставит немало практических советов.

Книга написана для тех, кто уже знаком с языком программирования Rust. Кроме того, читателю может пригодиться и опыт системного программирования с использованием других языков — таких как C, C++ и Java. Для усвоения материала книги не нужно быть экспертом по Rust, но задерживаться на разъяснении основ его синтаксиса, предыстории или концепций программирования я всё же не стану.

Многие примеры кода приводятся в книге в сокращенном виде, однако полные работоспособные их версии можно будет найти на GitHub¹. Доступность кода обу-

¹ См. <http://mng.bz/BA70>.

словлена лицензией MIT, позволяющей его свободное использование, копирование и модификацию. Для получения полноценной отдачи от работы с книгой я рекомендую по возможности изучать полные листинги кода. Имеющиеся в репозитории примеры кода упорядочены по главам, но некоторые из них могут охватывать материал сразу нескольких разделов или глав, поэтому их названия соответствуют иллюстрируемой тематике.

1.1. Так чем же примечателен Rust?

Rust (рис. 1.1) — современный язык программирования с прицелом на высокую производительность и безопасность. В нем имеется всё, что бы вам хотелось задействовать или же ожидать от современного языка программирования, включая замыкания, обобщения, асинхронный ввод/вывод, мощный инструментарий, интегрированные среды разработки, линтеры и инструменты проверки стиля. Наряду с широким набором функций Rust может похвастаться весьма энергичным и всё возрастающим сообществом разработчиков и приверженцев.

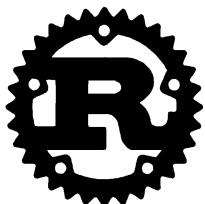


Рис. 1.1. Логотип языка Rust
(Источник: Основная команда разработчиков Rust. Распространяется под лицензией CC BY 4.0.)

Rust — весьма эффективный язык с целым рядом применений, включая веб-разработку. Хотя создавался он в качестве языка программирования системного уровня, он также вполне подходит и для других областей, выходящих далеко за пределы системного программирования, — например, для веб-программирования с использованием WebAssembly (Wasm), веб-стандарта для выполнения байт-кода. На рис. 1.2 показано место, обычно занимаемое языком Rust в стеке языков программирования, но эту позицию ни в коем случае не стоит считать окончательной.

Создатели Rust предполагали, что в основном он будет использоваться для написания системного кода и библиотек, где решающее значение имеют производительность и безопасность. Гарантии безопасности не обходятся без издержек, выражающихся в дополнительной сложности языка и продолжительности компиляции кода.

Rust способен конкурировать с такими высокоуровневыми языками, как Python или Ruby, но основным его недостатком в сравнении с ними является отсутствие в Rust интерпретатора среды выполнения, поскольку его код компилируется в зависимую от используемой платформы двоичную форму. Следовательно, разработчик должен распространять свои Rust-программы в виде двоичных файлов (или же предоставлять каким-то образом компилятор).

В целом ряде случаев Rust гораздо предпочтительнее таких языков сценариев, как Python или Ruby, — например, во встраиваемых или же в ограниченных ресурсами

средах. Посредством Wasm, чья популярность в последнее время существенно возросла, Rust также может быть напрямую скомпилирован для запуска программ в веб-браузерах. Wasm при этом просто считается своего рода движком, нацеленным на использование процессоров вроде x86-64 или AArch64, но только теперь в качестве центрального процессора выступает веб-браузер.

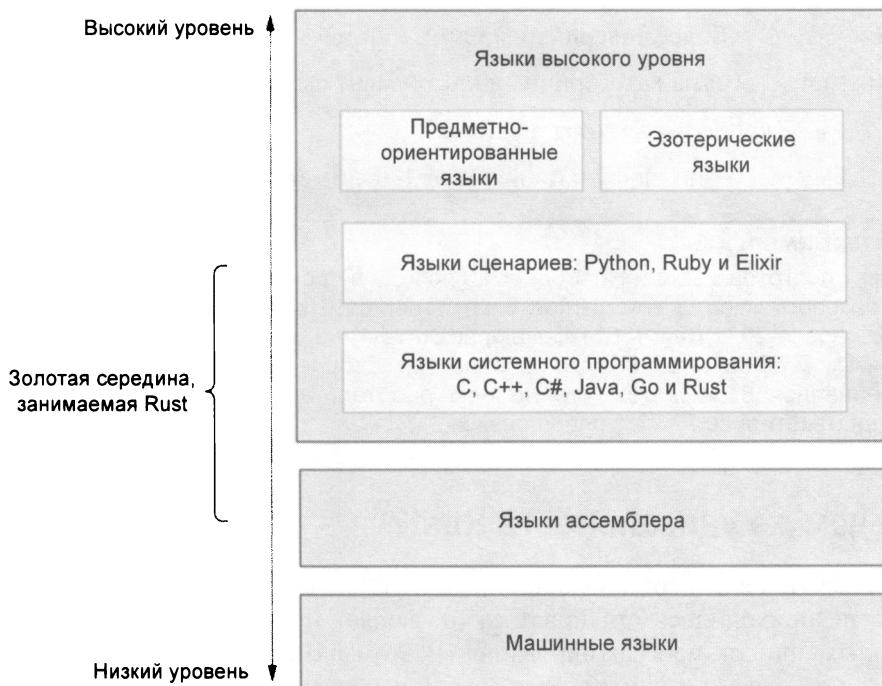


Рис. 1.2. Место, занимаемое Rust в классификации языков программирования

К числу основных особенностей языка Rust можно отнести следующие положения:

- Rust имеет основной набор инструментов для работы с языком, в неполный перечень которых входят:
 - `rustc` — официальный компилятор Rust;
 - `cargo` — диспетчер пакетов и средство сборки;
 - <https://crates.io> — реестр пакетов;
- в Rust имеется целый ряд возможностей, присущих современным языкам программирования, среди которых:
 - средство проверки заимствований, применяющее использующуюся в Rust модель управления памятью;
 - статическая типизация;
 - асинхронный ввод/вывод;
 - замыкания (closures);

- обобщения (generics);
 - макросы;
 - типажи (traits);
- для повышения качества и производительности кода в Rust предлагается несколько инструментов, созданных сообществом его приверженцев:
- rust-clippy — более совершенный линтер и инструмент стиля;
 - Rustfmt — весьма категоричный инструмент форматирования кода;
 - sccache — кеш компилятора для rustc;
 - rust-analyzer — полнофункциональная IDE-интеграция для языка Rust.

Самый любимый язык

На момент подготовки книги Rust одержал победу в проводимом с 2016 года ежегодном опросе разработчиков Stack Overflow в категории самых любимых языков программирования. В опросе 2021 года (2021 Developer Survey²) из 82 914 ответов Rust назвали любимым языком 86,98% тех, кто его использовал. Второе место занял язык Clojure, который понравился 81,12% тех, кто на нем работал, а в числе приверженцев языка TypeScript отметились 72,73% опрошенных.

1.2. В чем же уникальность Rust?

Rust за счет уникального набора абстракций, с рядом которых вам, возможно, еще никогда не приходилось сталкиваться, позволяет не допускать наиболее распространенных ошибок программирования. В этом разделе будет дан краткий обзор функций, позволяющих считать Rust особенным языком.

1.2.1. Rust безопасен

Безопасность — одна из характерных особенностей Rust, а его меры безопасности — во многом именно то, что отличает его от других языков программирования. Rust может дать абсолютные гарантии безопасности благодаря наличию в нем такой функции, как *проверка заимствований*.

В языках типа C и C++ управление памятью — отчасти ручной процесс, и разработчики при рассмотрении подходов к нему должны быть в курсе подробностей конкретной реализации памяти. А в таких языках, как Java, Go и Python, используется автоматическое управление памятью — сборка мусора, которая вносит путаницу в детали управления памятью и ее выделения с компромиссом в виде некоторых издержек производительности.

Имеющееся в Rust средство проверки заимствований проверяет ссылки *в ходе компиляции*, не ведя подсчет ссылок и не проводя сбор мусора в ходе выполнения

² См. <http://mng.bz/ddON>.

программы. Это уникальная особенность, усложняющая ко всем прочему написание программ, особенно если сталкиваться со средством проверки заимствований вам еще никогда не приходилось.

Проверка заимствований является составной частью Rust-компилятора `rustc`, следящей за тем, чтобы на любой отдельно взятый объект или переменную в текущий момент времени было не более одной изменяемой ссылки. На объекты или переменные можно иметь сразу несколько неизменяемых ссылок (например, ссылки только для чтения), но активная изменяемая ссылка может быть лишь одна. На рис. 1.3 показано, что Rust гарантирует безопасное использование памяти, проверяя, что у изменяемых и неизменяемых ссылок нет никаких перекрытий.



Рис. 1.3. Диаграмма Венна, демонстрирующая правила имеющейся в Rust проверки заимствований

Для отслеживания момента появления переменных и ссылок на них в области видимости и их исчезновения из него в Rust применяется концепция, согласно которой *получение ресурса есть инициализация* (Resource Acquisition Is Initialization, RAII). Как только они исчезают из области видимости, память может быть высвобождена. Проверка заимствований не позволяет ссылаться на переменные, находящиеся вне области видимости, и допускает наличие только одной изменяемой ссылки или нескольких неизменяемых ссылок, но никак и той и других одновременно.

Средство проверки заимствований обеспечивает также безопасность при конкурентном программировании. При совместном использовании данных, например разными потоками, может возникать состояние гонки. В большинстве случаев причина ее возникновения одна и та же: одновременное совместное использование изменяемых ссылок. В Rust невозможно иметь более одной изменяемой ссылки, чем, собственно, и гарантуется возможность предотвращения проблем синхронизации или как минимум их непреднамеренного создания.

Поначалу освоиться с применением проверки заимствований весьма непросто, но уже очень скоро станет ясно, что это одна из самых привлекательных особенностей

языка Rust. Как и в языках наподобие Haskell, удачной компиляции кода зачастую вполне достаточно (в сочетании с тестированием) для гарантии его работоспособности с полным исключением сбоев (тестирование будет рассматриваться в *глазах 6 и 7*). Из этого правила есть, конечно, исключения, но в общем и целом код, написанный на Rust, из-за самых распространенных ошибок, связанных с использованием памяти, вроде чтения с выходом за пределы буфера или неверного обращения с выделениями и высвобождениями памяти, сбить никогда не будет.

1.2.2. Rust современен

Особое внимание разработчики Rust уделяют поддержке современных парадигм программирования. Специалисты, работающие с другими языками программирования, могут заметить, что в Rust сочетаются как старые, так и новые подходы. Но в целом в Rust прослеживается отстранение от таких парадигм, как объектно-ориентированное программирование в пользу использования типажей, обобщений и функционального программирования.

В частности, в Rust выделяются следующие парадигмы и характерные особенности:

- *функциональное программирование* — замыкания, безымянные функции и итераторы;
- *обобщения* — также называемые *дженериками*;
- *типажи* (трейты) — то, что в других языках нам представляется как *интерфейсы*;
- *времена жизни* — для работы со ссылками;
- *метапрограммирование* — посредством собственной макросистемы;
- *асинхронное программирование* — посредством `async/await`;
- *управление пакетами и зависимостями* — посредством Cargo;
- *абстракции с нулевыми затратами*.

Традиционные объектно-ориентированные особенности в Rust отсутствуют. И хотя в действительности в Rust можно смоделировать паттерны, похожие на классы и наследование, терминология в нем совершенно иная, и Rust вполне подходит для функционального программирования. Чтобы привыкнуть к этому, специалистам с опытом объектно-ориентированного программирования на таких языках, как C++, Java или C#, может потребоваться некоторое время. Многие программисты, освоившие новые паттерны, находят определенное удовольствие и ощущают свободу при избавлении от строгостей объектно-ориентированной идеологии.

1.2.3. У Rust абсолютно открытый исходный код

При выборе языков и платформ для дальнейшей работы с прицелом на долгосрочное сопровождение любого проекта, важно учитывать деятельность сообщества. Ряд языков и платформ имеют открытый исходный код, но управляются главным образом крупными компаниями — например, Go (Google), Swift (Apple) и .NET (Microsoft), что привносит тем самым вполне конкретные риски, связанные с возможностью принятия решений в пользу продуктов этих самых компаний.

Rust является проектом, управляемым сообществом, возглавляемым в основе своей некоммерческой организацией Mozilla Foundation. А у самого языка программирования Rust имеется двойная лицензия — от Apache и MIT. Отдельно взятые проекты в экосистеме Rust лицензируются в индивидуальном порядке, но основная масса компонентов и библиотек пребывает под предполагающими открытый исходный код лицензиями, выданными в том числе Apache и MIT.

Разработка Rust активно поддерживается крупными технологическими компаниями — Amazon, Facebook, Google, Apple, Microsoft и не только планируют использование Rust или заявляют об обязательной его поддержке. Этот язык, не будучи привязанным ни к какой конкретной организации, является весьма перспективным выбором, которому практически не грозят никакие конфликты интересов.

ПРИМЕЧАНИЕ

Официальный список пользователей языка Rust ведется командой его разработчиков на сайте языка по адресу <https://www.rust-lang.org/production>.

1.2.4. Сравнение Rust с другими популярными языками

В табл. 1.1 приводится краткая сводка отличий Rust от других популярных языков программирования.

Таблица 1.1. Сравнение Rust с другими языками

Язык	Парадигмы	Типизация	Модель памяти	Ключевые особенности
Rust	Конкурентность, функциональность, использование обобщений, императивность	Статическая, строгая	RAII, явная	Безопасность, производительность, асинхронность
C	Императивность	Статическая, слабая	Явная	Эффективность, переносимость, низкоуровневое управление памятью, широкая поддержка
C++	Императивность, объектная ориентированность, использование обобщений, функциональность	Статическая, смешанная	RAII, явная	Эффективность, переносимость, низкоуровневое управление памятью, широкая поддержка
C#	Объектная ориентированность, императивность, событийная ориентированность, функциональность, рефлексивность, конкурентность	Статическая, динамическая, строгая	Со сборкой мусора	Поддержка на платформах Microsoft, обширная экосистема, расширенные языковые возможности
JavaScript	Использование прототипов, функциональность, императивность	Динамическая, утиная, слабая	Со сборкой мусора	Широкая поддержка, асинхронность

Таблица 1.1 (окончание)

Язык	Парадигмы	Типизация	Модель памяти	Ключевые особенности
Java	Использование обобщений, объектная ориентированность, императивность, рефлексивность	Статическая, строгая	Со сборкой мусора	Постоянно готовая к работе виртуальная машина Java на основе байт-кода, широкая поддержка, обширная экосистема
Python	Функциональность, императивность, объектная ориентированность, рефлексивность	Динамическая, утиная, строгая	Со сборкой мусора	Интерпретируемость, высокая переносимость, распространённость использования
Ruby	Функциональность, императивность, объектная ориентированность, рефлексивность	Динамическая, утиная, строгая	Со сборкой мусора	Особый синтаксис, всё в виде выражений, простая модель конкурентных вычислений
TypeScript	Функциональность, использование обобщений, императивность, объектная ориентированность	Статическая, динамическая, утиная, смешанная	Со сборкой мусора	Указание типов, совместимость с JavaScript, асинхронность

1.3. В каких случаях стоит воспользоваться языком Rust?

Rust является языком системного программирования, который обычно используется для написания низкоуровневых системных программ в тех же ситуациях, при которых действуются языки C или C++. При этом Rust вряд ли может отвечать стремлению оптимизировать продуктивность разработчиков, поскольку создавать код на языке Rust зачастую гораздо сложнее, чем на таких популярных языках, как Go, Python, Ruby или Elixir.

С появлением Wasm язык Rust стал отличным кандидатом для веб-программирования. Rust позволяет создавать приложения и библиотеки, компилировать их для Wasm и пользоваться преимуществами модели безопасности Rust с переносимостью, предоставляемой веб-технологией.

Для применения Rust не существует каких-либо конкретных условий — его следует использовать там, где это имеет смысл. Лично мной Rust использовался для создания многих мелких одноразовых проектов просто потому, что написание кода на нем приносит мне радость, а как только такой код проходит компиляцию, можно всецело рассчитывать на его работоспособность. При должном использовании компилятора Rust и его инструментария в создаваемом коде будет гораздо меньше ошибок или же проявления какого-либо неопределенного поведения, что, конечно же, всегда приветствуется при разработке любого проекта.

СОВЕТ

Использование удачно подобранного рабочего инструмента играет важную роль в достижении успеха любого амбициозного проекта, но чтобы разобраться в том, какой именно инструмент вам подойдет, сначала нужно набраться соответствующего опыта, попробовав различные инструменты для решения разных задач.

Вот список вариантов использования Rust, где в полной мере могут пригодиться все его преимущества:

- **ускорение кода** — Rust позволяет ускорить выполнение функций, взятых из таких языков программирования, как Python, Ruby или Elixir;
- **конкурентные системы** — гарантии безопасности, предоставляемые Rust, распространяются и на конкурентный код. Поэтому Rust становится идеальным языком для программирования высокопроизводительных конкурентных систем;
- **криптография** — Rust идеально подходит для реализации криптографических алгоритмов;
- **программирование для встроенных систем** — компилятором Rust генерируются двоичные файлы, связывающие все зависимости, за исключением системной библиотеки языка C или любых системных библиотек C от сторонних разработчиков. Это позволяет получить относительно простой двоичный дистрибутив, удобный для встроенных систем. Кроме того, имеющаяся в Rust модель управления памятью отлично подходит для систем, требующих ее минимальных затрат;
- **использование в агрессивной среде** — в ситуациях, когда безопасность выдвигается на первый план, идеально подходят гарантии, обеспечиваемые языком Rust;
- **критично высокая производительность** — Rust оптимизирован под достижение высоких уровней как безопасности, так и производительности. На языке Rust легко пишется код, выполняемый невероятно быстро, причем без компромиссов в обеспечении безопасности;
- **обработка строк** — Rust особенно хорош для решения такой весьма проблемной задачи, как обработка строк, поскольку он позволяет без особого труда создавать код, не допускающий переполнения;
- **замена устаревших программ на C или на C++** — Rust будет отличным выбором для замены устаревшего кода, написанного на C или на C++;
- **безопасное веб-программирование** — Rust может быть нацелен на применение Wasm, позволяя создавать веб-приложения с присущей Rust безопасностью и строгой проверкой типов.

1.4. Требуемые инструменты

Коллекция примеров кода, включенная в книгу, доступна под лицензией MIT совершенно свободно. Для получения копии кода понадобится подключенный к Интернету компьютер с установленными поддерживаемой операционной системой и

набором инструментов, рассмотренным в табл. 1.2. Подробности установки этого набора можно найти в *приложении*.

Таблица 1.2. Требуемый набор инструментов

Название	Описание
git	Исходный код для этой книги хранится в открытом Git-хранилище, размещенном в GitHub ³
rustup	Rust-инструмент для управления компонентами языка rustup будет управлять вашей установкой rustc и других Rust-компонентов
gcc или clang	Небольшому числу из общего количества примеров кода для компиляции потребуется установка копии коллекции компиляторов GNU Compiler Collection (GCC) или Clang. Наверное, лучшим выбором для большинства читателей будет все же Clang, поэтому именно эта коллекция и упоминается изначально. В тех случаях, когда будет указана команда clang, вы, если захотите, можете спокойно заменить ее на gcc

Резюме

- Rust представляет собой современный язык программирования системного уровня с весьма эффективными функциями безопасности и с абсолютно не требующими никакого расхода ресурсов абстракциями.
- Первоначальным сдерживающим фактором может стать довольно сложный процесс обучения программированию на языке Rust, но эта книга поможет преодолеть все препятствия.
- У Rust много общего с другими языками, и всё же он уникален, подтверждением чему может послужить весь материал этой книги.
- Весьма активное сообщество приверженцев Rust и полноценное хранилище пакетов образуют богатую надстроенную над языком экосистему.
- Для полноценного усвоения материалов книги, воспользуйтесь доступными примерами кода⁴.

³ См. <http://mng.bz/BA70>.

⁴ См. <http://mng.bz/BA70>.

Часть I

Работа с Rust на профессиональном уровне

Особая ценность Rust определяется его скоростью, безопасностью и широким выбором включенных в него инструментальных средств. Изучить язык, конечно, нужно, но быстро достичь высокого уровня мастерства в работе с ним поможет овладение инструментами, как входящими в основной проект Rust, так и их расширенным набором, предоставляемым сообществом приверженцев Rust.

Богатая инструментальная оснащенность, как и достижение с ее помощью высокой эффективности в работе, отличает хорошие языки программирования от плохих. Сложившееся у вас представление о составе доступного инструментария и о реализуемых с его помощью возможностях позволит вам стать на голову выше разработчиков, не пожелавших потратить свое время на изучение инструментов.

В *первой части* книги мы уделим время знакомству с основами языка (или их рассмотрению — в зависимости от вашего уровня знаний), а также, в особенности, с инструментами, необходимыми для работы с ним. Чтобы использовать эти инструменты, можно не быть крутым Rust-профессионалом, но для эффективного взаимодействия с языком надо как следует разобраться с порядком их применения.

Управление проектами с помощью *Cargo*

В этой главе:

- представление Cargo и рассмотрение способов его применения для управления Rust-проектами;
- управление зависимостями в Rust-проектах;
- сравнение Rust с другими языками программирования;
- порядок привязки к не-Rust библиотекам;
- порядок выпуска Rust-приложений и библиотек;
- порядок документирования кода на языке Rust;
- порядок следования всем передовым начинаниям Rust-сообщества по управлению Rust-проектами и выпуску приложений на их основе;
- порядок структурирования Rust-проектов с использованием модулей и рабочих пространств;
- соображения по применению Rust для разработки программ, работающих во встроенных средах.

До перехода непосредственно к созданию приложений на языке Rust нужно прежде ознакомиться с его основными инструментами. Спешу заверить вас, что овладение инструментами, каким бы скучным оно ни казалось, имеет решающее значение для успешного освоения языка. Инструменты создавались разработчиками языка для облегчения жизни его пользователям, и четкое понимание их предназначения поможет вам достичь успеха в работе с ним.

В Rust пакетами управляет инструментальное средство под названием *Cargo*, представляющее собой интерфейс к Rust-компилятору `rustc`, реестру <https://crates.io> и ко многим другим имеющимся в Rust инструментам (которые более подробно будут рассматриваться в главе 3). По правде говоря, Rust и `rustc` можно использовать и без применения Cargo, но делать так я бы никому не посоветовал.

Работая с Rust, вы, скорее всего, будете постоянно обращаться к Cargo и к инструментам, применяемым в Cargo. Поэтому вам весьма важно как можно быстрее

ознакомиться с этим средством и с самыми ходовыми приемами работы с ним. В главе 3 я приведу рекомендации и подробную информацию о том, как еще больше повысить полезность Cargo с помощью *крейтов*¹, разработанных сообществом.

2.1. Ознакомительный тур по Cargo

Чтобы продемонстрировать все основные особенности Cargo, давайте проведем ознакомительный тур по этому средству и по наиболее распространенным приемам его применения.

2.1.1. Основное применение

Для начала запустим на выполнение команду cargo help, чтобы вывести на экран перечень доступных команд:

```
$ cargo help
Rust's package manager
```

USAGE:

```
cargo [+toolchain] [OPTIONS] [SUBCOMMAND]
```

OPTIONS:

-V, --version	Print version info and exit
--list	List installed commands
--explain <CODE>	Run `rustc --explain CODE`
-v, --verbose	Use verbose output (-vv very verbose/build.rs output)
-q, --quiet	No output printed to stdout
--color <WHEN>	Coloring: auto, always, never
--frozen	Require Cargo.lock and cache are up to date
--locked	Require Cargo.lock is up to date
--offline	Run without accessing the network
-Z <FLAG>...	Unstable (nightly-only) flags to Cargo, see 'cargo -Z help' for details
-h, --help	Prints help information

Some common cargo commands are (see all commands with --list):

build, b	Compile the current package
check, c	Analyze the current package and report errors, but don't build object files
clean	Remove the target directory
doc	Build this package's and its dependencies' documentation
new	Create a new cargo package
init	Create a new cargo package in an existing directory
run, r	Run a binary or example of the local package

¹ Крейт — это наименьший объем кода, который компилятор Rust рассматривает за раз.

```
test, t      Run the tests
bench        Run the benchmarks
update       Update dependencies listed in Cargo.lock
search        Search registry for crates
publish      Package and upload this package to the registry
install      Install a Rust binary. Default location is $HOME/.cargo/bin
uninstall    Uninstall a Rust binary
```

See 'cargo help <command>' for more information on a specific command.

Когда вы сами запустите эту команду, вывод на экране может немного отличаться от показанного здесь, поскольку всё зависит от той версии Cargo, которая установлена на вашей машине. Если же ничего похожего на приведенный здесь вывод вы не увидите, вам лучше проверить установку Cargo на работоспособность (установка Cargo подробно рассмотрена в *приложении* к книге).

2.1.2. Создание нового приложения или библиотеки

В Cargo имеется встроенный генератор шаблонов, способный создать приложение «Hello, world!» или только лишь библиотеку, что позволит сэкономить поначалу ваше время. Начните с запуска в вашей рабочей среде из каталога разработки (лично я предпочитаю использовать каталог `~/dev`) следующей команды:

```
$ cargo new dolphins-are-cool
Created binary (application) `dolphins-are-cool` package
```

Эта команда создает новое шаблонное приложение по имени `dolphins-are-cool` (имя можно заменить на что угодно). Давайте коротко рассмотрим то, что будет выведено на экран:

```
$ cd dolphins-are-cool/
$ tree
```

```
.
├── Cargo.toml
└── src
    └── main.rs
```

1 directory, 2 files

Как можно видеть, средством Cargo создано два файла:

- `Cargo.toml` — файл конфигурации Cargo для нового приложения в формате TOML;
- `main.rs` — точка входа для нашего нового приложения, находящаяся внутри каталога `src`.

Подсказка

TOML (Tom's Obvious Minimal Language) — минимальный формат файла конфигурации, легко читаемый благодаря очевидной семантике и используемый многими инструментами, имеющими прямое отношение к Rust. Подробные сведения по TOML можно найти по адресу <https://toml.io>.

А теперь воспользуемся командой `cargo run`, чтобы провести компиляцию вновь созданного приложения и запустить его на выполнение:

```
$ cargo run
Compiling dolphins-are-cool v0.1.0 (/Users/brenden/dev/dolphins-are-cool)
  Finished dev [unoptimized + debuginfo] target(s) in 0.59s
    Running `target/debug/dolphins-are-cool`
Hello, world!
```

Здесь `Hello, world!` — это то, что выводится программой на языке Rust на экран.

Запуск точно такой же команды `cargo new`, но уже с аргументом `--lib`, приведет к созданию новой библиотеки:

```
$ cargo new narwhals-are-real --lib
  Created library `narwhals-are-real` package
$ cd narwhals-are-real/
$ tree
.
├── Cargo.toml
└── src
    └── lib.rs
```

1 directory, 2 files

Код, сгенерированный командой `cargo new --lib`, несколько отличается от предыдущего тем, что содержит не функцию `main`, а одиночный тест, находящийся в файле `src/lib.rs`. Этот тест можно запустить на выполнение командой `cargo test`:

```
$ cargo test
  Finished test [unoptimized + debuginfo] target(s) in 0.00s
    Running target/debug/deps/narwhals_are_real-3265ca33d2780ea2

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests narwhals-are-real

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Подсказка

Приложениями в качестве точки входа используется файл `src/main.rs`, а библиотеками — файл `src/lib.rs`.

При выполнении команды `cargo new` Cargo производит автоматическую инициализацию нового каталога в виде Git-репозитория (за исключением случаев, когда он уже находится внутри репозитория), включая файл `.gitignore`. Cargo также поддерживает системы контроля версий hg, Pijul и Fossil с ключом `--vcs`.

2.1.3. Компиляция, запуск и тестирование

Основные команды, с которыми вам, наверное, придется постоянно работать, — это `build`, `check`, `test` и `run` (табл. 2.1).

Таблица 2.1. Краткое описание основных команд Cargo

Команда Cargo	Краткое описание
Build	Выполняет компиляцию и сборку вашего пакета, создавая все конечные цели
Check	Выполняет то же, что и <code>build</code> , но не генерирует никаких целей или объектов, — просто проверяет приемлемость кода
Test	Выполняет компиляцию и запуск всех тестов
Run	Выполняет компиляцию и запуск целевого двоичного файла

При этом чаще всего вам, вероятно, понадобится выполнять команды `cargo check` и `cargo test`. Использование команды `check` позволяет сэкономить время и быстро производить последовательное улучшение создаваемого кода, поскольку проверка синтаксиса выполняется быстрее, чем компиляция (при запуске команды `cargo build`). Давайте в качестве иллюстрации посмотрим, сколько времени займет компиляция крейта `dryoc`², используемого в примерах этой книги, при выполнении команд `cargo build` и `cargo check`:

```
$ cargo clean
$ time cargo build
...
Finished dev [unoptimized + debuginfo] target(s) in 9.26s
cargo build
$ cargo clean26.95s user 5.18s system 342% cpu 9.374 total
$ time cargo check
...
Finished dev [unoptimized + debuginfo] target(s) in 7.97s
cargo check
23.24s user 3.80s system 334% cpu 8.077 total
```

Разница здесь несущественна: примерно 9,374 секунды для команды `build` против 8,077 секунды для команды `check` (в соответствии с физическим временем, указанным командой `time`). Но при работе с крупными крейтами экономия времени может быть весьма значительной. Кроме того, следует учесть и мультиплатформенный

² См. <https://github.com/brndnmthws/dryoc>.

эффект, достигаемый при частой и многократной перекомпиляции (или перепроверке) кода при неоднократном внесении в него изменений.

2.1.4. Переключения между наборами инструментов

Набор инструментов (*toolchain*) подбирается под сочетание архитектуры, платформы и канала. Вот один из примеров: `stable-x86_64-apple-darwin` — стабильный канал для *x64-64 Darwin* (эквивалента macOS от Apple на центральных процессорах Intel). Rust выпускается в трех разных *каналах*: `stable`, `beta` и `nightly` (стабильном, бета и ночном). Стабильный канал — наименее часто обновляемый и наиболее проверенный. Бета-версия канала содержит функции, готовые к стабилизации, но требующие дальнейшего тестирования и возможного изменения. Ночная версия канала содержит еще не выпускавшиеся функции языка, считающиеся работающими.

При работе с Rust часто приходится переключаться между различными наборами инструментов — в особенности, между каналами `stable` и `nightly`. Простой способ переключиться с помощью *Cargo* — воспользоваться опцией `+channel`:

```
# Runs tests with stable channel:  
...  
$ cargo +stable test  
  
# Runs tests with nightly channel:  
$ cargo +nightly test  
...
```

ПРИМЕЧАНИЕ

Возможно, перед запуском команды `cargo +nightly` потребуется установить ночной набор инструментов, выполнив команду `rust up toolchain install nightly`, если, конечно, этого еще не сделано ранее. Если *Cargo* устанавливается через диспетчер пакетов системного уровня (например, `apt` от *Debian*), эта команда может сработать неожиданным образом.

Применение этой опции — самый быстрый способ переключения между наборами инструментов, работающий со всеми командами *Cargo*. Альтернативой ему является переключение исходного набора инструментов с помощью команды `rustup`, описание которой дается в *приложении*.

Зачастую возникает желание перед выпуском своего кода протестировать его в обоих каналах — стабильном и ночном, особенно в проектах с открытым исходным кодом, поскольку многими пользователями задействуются оба этих набора инструментов. Кроме того, многие Rust-проекты работают только в ночном канале, о чем более подробный разговор пойдет в *главе 3*.

В `rustup` можно также воспользоваться опцией `override`, позволяющей установить набор инструментов для конкретного проекта или каталога. Инструмент `rustup` сохраняет эту конфигурацию в своем файле `settings.toml`, который в UNIX-подобных системах находится в каталоге `$HOME/.rustup`. Например, с помощью следующего кода для текущего рабочего каталога можно установить значение ночного канала:

```
# Only applies to the current directory and its children
$ rustup override set nightly
```

Это очень удобно, поскольку позволяет сохранять по умолчанию стабильный канал, а для отдельных проектов переключаться на ночной.

2.2. Управление зависимостями

Одним из факторов повышения эффективности работы с Rust является использование в его среде реестра пакетов (или крейтов) [crates.io](#). Пакеты в сообществе Rust называются *крейтами* (*crates*) и включают в себя как приложения, так и библиотеки. На момент подготовки книги было доступно уже более 92 тыс. различных крейтов.

Под крейтами в этой книге чаще всего подразумеваются не приложения, а библиотеки. В главе 3 мы рассмотрим дополнительные инструментальные средства Rust, устанавливаемые с помощью крейтов, но в основном всё же в книге будут применяться библиотеки.

В Rust по сравнению с некоторыми другими языками программирования используется свой уникальный подход, при котором в сам язык как таковой включены весьма скромные возможности. Если сравнивать Rust с такими языками, как Java, C# и даже C++, то в них важные компоненты составляют в определенной степени основную часть языка (они либо задействуются в процессе выполнения программы, либо являются частью компилятора). К примеру, если сравнивать число используемых в Rust основных структур данных со структурами в других языках, то можно отметить их малочисленность и принадлежность многих из них к оболочкам основной изменяемой структуры данных `Vec`. В предоставлении функций предпочтение в Rust отдается не созданию обширных стандартных библиотек, а использованию крейтов.

В сам язык Rust не включен даже генератор случайных чисел, играющий важную роль в решении многих задач программирования. Для них приходится использовать крейт `rand`, который на момент подготовки книги был самым загружаемым, или же создавать собственный генератор случайных чисел.

Если ранее вам приходилось работать с такими языками, как JavaScript, Ruby или Python, то крейты, имеющиеся в Rust, покажутся вам в чем-то похожими на используемые в них соответствующие инструменты управления пакетами. А если до этого работу вы вели с языками С или C++, то крейты станут новым словом в вашей практике программирования. В общем, прошли те времена, когда вручную писались сложные средства управления сборкой сторонних библиотек или выполнялась интеграция стороннего кода и встраивание систем в собственный репозиторий исходных текстов.

Конкретизация зависимостей в Rust происходит путем их перечисления в файле `Cargo.toml`. В листинге 2.1 показан простой пример использования крейта `rand`.

Листинг 2.1. Небольшой Cargo.toml

```
[package]
name = "simple-project"
version = "0.1.0"
authors = ["Brenden Matthews <brenden@brndn.io>"]
edition = "2018"

[dependencies]
rand = "0.8"
```

В приведенном коде крейт `rand` включен с использованием последней — 0.8 — версии библиотеки. При указании версий зависимостей следует руководствоваться семантическим версионированием SemVer³, в котором используется шаблон `major.minor.patch` (мажорная.минорная.патч). Если оператор не указан, то Cargo будет по умолчанию руководствоваться *каретными* (caret) требованиями, разрешающими обновления до наименьшей указанной версии.

Добавить зависимость в проект можно также с помощью команды `cargo add`:

```
# Добавляет крейт rand в качестве зависимости к текущему проекту
$ cargo add rand
```

В Cargo поддерживаются следующие разновидности требований:

- каретные:** ^x.y.z;
- тильдовые:** ~x.y.z;
- джокерные:** *, x.*;
- сравнительные:** >=x, <x.y, =x.y.z;
- и их сочетания** (см. табл. 2.2).

По сути, вам нужно указать версию библиотеки в виде `major.minor` (разрешая при этом применение совместимых обновлений в соответствии с каретными требованиями) или `=major.minor.patch` (привязывая к конкретной версии). Дополнительную информацию об указаниях зависимостей можно найти по адресу <http://mng.bz/rjAB>.

Таблица 2.2. Примеры задания зависимостей в соответствии с SemVer

Оператор	Пример	Минимальная версия	Максимальная версия	Обновление
Карет	<code>^2.3.4</code>	<code>>=2.3.4</code>	<code><3.0.0</code>	Разрешено
Карет	<code>^2.3</code>	<code>>=2.3.0</code>	<code><3.0.0</code>	Разрешено
Карет	<code>^0.2.3</code>	<code>>=0.2.3</code>	<code><0.3.0</code>	Разрешено
Карет	<code>^2</code>	<code>>=2.0.0</code>	<code><3.0.0</code>	Разрешено

³ См. <https://semver.org>.

Таблица 2.2 (окончание)

Оператор	Пример	Минимальная версия	Максимальная версия	Обновление
Тильда	<code>~2.3.4</code>	<code>>=2.3.4</code>	<code><2.4.0</code>	Разрешено
Тильда	<code>~2.3</code>	<code>>=2.3.0</code>	<code><2.4.0</code>	Разрешено
Тильда	<code>~0.2</code>	<code>>=0.2.0</code>	<code><0.3</code>	Разрешено
Джокер	<code>2.3.*</code>	<code>>=2.3.0</code>	<code><2.4.0</code>	Разрешено
Джокер	<code>2.*</code>	<code>>=2.0.0</code>	<code><3.0.0</code>	Разрешено
Джокер	<code>*</code>	Не определена	Не определена	Разрешено
Сравнение	<code>=2.3.4</code>	<code>=2.3.4</code>	<code>=2.3.4</code>	Не разрешено
Сравнение	<code>>=2.3.4</code>	<code>>=2.3.4</code>	Не определена	Разрешено
Сравнение	<code>>=2.3.4,<3.0.0</code>	<code>>=2.3.4</code>	<code><3.0.0</code>	Разрешено

Для разбора указанных версий внутри Cargo используется крейт `semver`⁴. Когда в проекте запускается команда `cargo update`, Cargo, согласно заданному указанию зависимостей, обновляет файл `Cargo.lock` перечнем самых новых доступных крейтов.

СОВЕТ

Я по возможности стараюсь избегать привязки версий зависимостей, особенно относительно библиотек. Впоследствии, когда вновь создаваемые конкурирующие пакеты потребуют разных версий общих библиотек, это вызовет явные неудобства. Хотя многие и ратуют за привязку версий, лучше все-таки по мере необходимости допускать некоторую гибкость.

Конкретные способы задания зависимостей — тема, требующая более пространного обсуждения. Каких-то жестких правил на этот счет не существует, но обычно предполагается, что и в других проектах будут придерживаться шаблона SemVer. В каких-то проектах ему строго следуют, а в каких-то нет. В большинстве случаев приходится подстраиваться под конкретную ситуацию. Разумнее всего разрешить обновление до минорных версий и патчей, указав с помощью каретного оператора минимальную требуемую версию, используемую в Rust по умолчанию (что и произойдет, если вообще не указывать никакого оператора). А для своих собственных публикуемых крейтов следуйте, пожалуйста, шаблону SemVer, помогая тем самым другим разработчикам встраивать то, что вами сделано, сохраняя при этом совместимость с прежними продуктами.

Работа с файлом `Cargo.lock`

Работа с файлом `Cargo.lock` требует внимания — по крайней мере в части того, что относится к системам управления версионированием. В этом файле содержится перечень пакетных зависимостей (как явных, так и неявных), их версий и контрольных сумм, слу-

⁴ См. <https://crates.io/crates/semver>.

жащих для проверки достоверности. Специалисты, работавшие с языками со сходными системами управления зависимостями, возможно, ранее уже сталкивались с подобными файлами (`package-lock.json` в npm, `Gemfile.lock` в gem-системе Ruby и `poetry.lock` в Poetry-системе Python).

Для библиотек этот файл в свою систему контроля версий лучше не включать. Если используется Git, это можно сделать путем добавления `Cargo.lock` в файл `.gitignore`. Игнорирование `lock`-файла позволит нижестоящим пакетам обновлять косвенные зависимости по мере надобности.

Для приложений же рекомендуется наряду с `Cargo.toml` всегда включать и `Cargo.lock`. Это помогает обеспечить согласованное поведение системы в опубликованных релизах при последующих возможных изменениях сторонних библиотек. Впрочем, это устоявшиеся соглашения, не являющиеся чем-то уникальным только для Rust.

В завершение Cargo автоматически создаст для вас соответствующий файл `.gitignore` и инициализирует репозиторий Git.

2.3. Фича-флаги

При публикации программных средств, особенно библиотек, принято использовать дополнительные зависимости. Обычно это делается для сокращения времени компиляции и уменьшения размеров двоичных файлов, а также, возможно, для повышения производительности, расплачиваясь за это усложнением процесса компиляции.

Бывает, что дополнительные зависимости нужно включить в состав вашего крейта. Они могут быть представлены в Cargo в виде флагов функциональных особенностей (фича-флагов) — их примеры приведены в табл. 2.3. На фича-флаги накладываются ограничения — в частности, в них допускаются только логические выражения (т. е. они включены или отключены). Фича-флаги также передаются в крейты в вашем списке зависимостей, поэтому с помощью фича-флагов верхнего уровня можно включать соответствующие функциональные особенности для тех крейтов, что указаны ниже их.

Таблица 2.3. Примеры фича-флагов из крейта `dryoc`

Флаг	Описание	Включен по умолчанию
<code>serde</code>	Включает дополнительную зависимость <code>serde</code>	Нет
<code>base64</code>	Включает зависимость <code>base64</code> , активируемую только при разрешенной зависимости <code>serge</code>	Нет
<code>simd_backend</code>	Включает SIMD и функциональные особенности ассемблирования для крейтов <code>curve25519-dalek</code> и <code>sha2</code>	Нет
<code>u64_backend</code>	Включает взаимоисключающий с <code>u32_backend</code> внутренний компонент U64 для крейта <code>x25519-dalek</code>	Да
<code>u32_backend</code>	Включает взаимоисключающий с <code>u64_backend</code> внутренний компонент U32 для крейта <code>x25519-dalek</code>	Нет

Однако я не рекомендую излишне полагаться на фича-флаги. Можно увлечься созданием суперкрайтов с множеством фича-флагов, но, поймав себя за этим занятием, будет, наверное, лучше разбить свой крейт на более мелкие отдельные подкрайты. Этот прием применяется довольно часто — в качестве подходящих примеров можно привести крейты `serde`, `rand` и `rocket`. Впрочем бывает, что для использования конкретных дополнительных функциональных особенностей — например, при предоставлении реализаций дополнительных свойств в крейте верхнего уровня, без фича-флагов *просто не обойтись*.

Чтобы понять, как фича-флаги используются на практике, давайте рассмотрим крейт `dryok` (листинг 2.2). В этом крейте используется целый ряд флагов, указывающих на несколько функциональных особенностей: на `serde`, на `base64`, для двоичного кодирования (с `serde`) и на SIMD-оптимизацию.

Листинг 2.2. Cargo.toml из крейта dryoc

```
[dependencies]
base64 = {version = "0.13", optional = true}                                (1)
curve25519-dalek = "3.0"
generic-array = "0.14"
poly1305 = "0.6"
rand_core = {version = "0.5", features = ["getrandom"]}
salsa20 = {version = "0.7", features = ["hsalsa20"]}
serde = {version = "1.0", optional = true, features = ["derive"]}           (2)
sha2 = "0.9"
subtle = "2.4"
x25519-dalek = "1.1"
zeroize = "1.2"

[dev-dependencies]
base64 = "0.13"
serde_json = "1.0"
sodiumoxide = "0.2"

[features]
default = [                                                               (3)
    "u64_backend",
]
simd_backend = ["curve25519-dalek/simd_backend", "sha2/asm"]                (5)
u32_backend = ["x25519-dalek/u32_backend"]
u64_backend = ["x25519-dalek/u64_backend"]
```

Здесь:

- в поз. (1) — необязательная зависимость `base64`, не включаемая по умолчанию;
- в поз. (2) — необязательная зависимость `serde`, не включаемая по умолчанию;
- в поз. (3) — раздел используемых по умолчанию и дополнительных функциональных особенностей;

- в поз. (4) — список стандартных функциональностей;
- в поз. (5) — дополнительные функциональности и функциональности, включаемые ими для зависимостей.

А теперь давайте исследуем часть кода крейта, чтобы понять механизм применения выражений `cfg` и `cfg_attr`, предписывающих Rust-компилятору `rustc` порядок использования этих флагов, для чего рассмотрим содержимое файла `src/message.rs` (листинг 2.3).

Листинг 2.3. Часть кода файла `src/message.rs` из крейта `dryoc`

```
#[cfg(feature = "serde")]
use serde::{Deserialize, Serialize}; (1)

use zeroize::Zeroize;

#[cfg_attr(
    feature = "serde",
    derive(Deserialize, Serialize, Zeroize, Debug, PartialEq)
)] (2)
    (2)
    (2)
}

#[cfg_attr(not(feature = "serde"), derive(Zeroize, Debug, PartialEq))] (3)
#[zeroize(drop)]
/// Контейнер для незашифрованных сообщений
pub struct Message(pub Box<InputBase>);
```

Здесь:

- в поз. (1) — включение инструкции `use` только при включенном `serde`;
- в поз. (2) — включение инструкции `derive()` только при включенном `serde`;
- в поз. (3) — включение инструкции `derive()` только при выключенном `serde`.

В коде этого листинга используются несколько условных атрибутов компиляции:

- `cfg(предикат)` — предписывает компилятору выполнять компиляцию прикрепленного кода, только если предикат (первый аргумент) является истинным;
- `cfg_attr(предикат, атрибут)` — предписывает компилятору включить указанный атрибут (второй аргумент) только в том случае, если предикат (первый аргумент) является истинным;
- `not(предикат)` — возвращает `true`, если предикат ложный, и наоборот.

Кроме них можно также использовать выражения `all(предикат)` и `any(предикат)`, возвращающие `true`, когда все или же какой-нибудь из предикатов вычисляется в `true`. Еще больше примеров вы можете увидеть в файлах `src/lib.rs`, `src/b64.rs`, `src/dryocbox.rs` и `src/dryocsecretbox.rs` из крейта `dryoc`.

СОВЕТ

При документировании проекта с помощью команды `rustdoc` листинг фича-флага предоставляется вам автоматически (эта команда будет подробно рассмотрена чуть позже).

2.4. Корректировка зависимостей

Периодически может возникать проблема, связанная с необходимостью внесения изменений в код вышестоящего крейта (т. е. внешнего крейта, входящего в зависимости вашего проекта). У меня нередко возникала потребность в обновлении внешнего крейта, входящего в зависимости, что обычно вызывалось проявлением какой-то небольшой проблемы. Потребности в замене функциональности вышестоящих крейтов всего лишь для исправления одной-двух незначительных ошибок возникают довольно редко. Порой можно просто переключиться на предыдущую версию крейта — в противном случае вам придется править его код самостоятельно.

Процесс внесения изменений в код вышестоящего крейта происходит примерно следующим образом:

1. Создается разветвление (форк-проект) на GitHub.
2. В крейт в вашей ветви (в форк-проект) вносятся изменения.
3. Отправляется запрос на включение изменений в вышестоящий проект.
4. Вносятся изменения в файл `Cargo.toml`, указывающие на вашу ветвь в ожидании запроса на включение ее для объединения и выпуска.

Этот процесс не проходит гладко. Ему препятствует отслеживание изменений в вышестоящем крейте и их включение по мере надобности. Другой проблемой может стать грозящий подвисанием на разветвлении отказ в принятии ваших изменений вышестоящим крейтом. По возможности разветвлений при работе с вышестоящими крейтами следует избегать.

Cargo облегчает внесение изменений в крейты с использованием рассмотренного здесь метода разветвления, но тут уже не обойтись без целого ряда предостережений. Давайте в качестве примера рассмотрим типичный процесс внесения исправлений в крейт, просто сделав вместо форк-проекта на GitHub локальную копию исходного кода.

Итак, мы модифицируем крейт `num_cpus`, заменив его собственной измененной версией. Этот крейт был выбран из-за его простоты — он всего лишь возвращает число логических ядер ЦПУ. Начнем с создания пустого проекта:

```
$ cargo new patch-num-cpus
$ cd patch-num-cpus
$ cargo run
...
Hello, world!
```

Затем добавим в `Cargo.toml` зависимость от `num_cpus`:

```
[dependencies]
num_cpus = "0.0"
```

Обновим файл `src/main.rs` для вывода на экран числа ЦПУ:

```
fn main() {
    println!("There are {} CPUs", num_cpus::get());
}
```

И в завершение запустим новый крейт:

```
$ cargo run
   Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/patch-num-cpus`
There are 4 CPUs
```

Пока здесь нет никаких изменений или модификаций. Давайте в том же самом рабочем каталоге `num_cpus`, где будет пересоздаваться наш API, создадим новую библиотеку:

```
$ cargo new num_cpus --lib
...
```

А теперь внесем изменения в код исходного файла `src/lib.rs`, чтобы реализовать функцию `num_cpus::get()`. Для этого приведите код файла `src/lib.rs` из каталога `num_cpus` к следующему виду:

```
pub fn get() -> usize {
    100           ← это произвольное значение возвращается в целях тестирования
```

Теперь у нас имеется крейт `num_cpus` с нашей собственной реализацией, возвращающей не имеющее никакого смысла жестко заданное значение (в нашем случае 100). Вернитесь в каталог с исходным проектом `patch-num-cpus` и внесите изменения в `Cargo.toml`, позволяющие использовать новый крейт:

```
[dependencies]
num_cpus = { path = "num_cpus" }
```

Запустите тот же самый код с измененным крейтом:

```
$ cargo run
Compiling patch-num-cpus v0.1.0
   Finished dev [unoptimized + debuginfo] target(s) in 0.33s
     Running `target/debug/patch-num-cpus`
There are 100 CPUs
```

Этот пример при всей своей бессмысленности вполне понятно иллюстрирует процесс. К примеру, если нужно внести изменения в зависимость, воспользовавшись форк-проектом из GitHub, указание на зависимость — примерно следующего вида (в `Cargo.toml`) — должно быть направлено непосредственно в репозиторий GitHub:

```
[dependencies]
num_cpus = { git = "https://github.com/brndnmthws/num_cpus",
             rev = "b423db0a698b035914ae1fd6b7ce5d2a4e727b46" }
```

Если теперь запустить на выполнение команду `cargo run`, вы опять увидите отчет с правильным числом центральных процессоров (поскольку мною ранее был создан

форк-проект без внесения каких-либо изменений). При этом `rev` в приведенном примере ссылается на последний на момент написания кода Git-хеш. При компиляции проекта Cargo извлечет исходный код из репозитория GitHub, проверит конкретную указанную редакцию (которая может быть фиксацией изменений, разветвлением или тегом) и скомпилирует эту версию в качестве зависимости.

2.4.1. Косвенные зависимости

Иногда нужно внести изменения в зависимости зависимостей. Иначе говоря, у вас может быть зависимость от крейта, который сам зависит от другого крейта, требующего внесения изменений. Если взять в качестве примера крейт `num_cpus`, то он в настоящий момент зависит от `libc = "0.2.26"` (но только на платформах, отличных от Windows). Для примера мы можем внести изменения в эту зависимость, указав более свежую версию и обновив `Cargo.toml` следующим образом:

```
[patch.crates-io]
libc = { git = "https://github.com/rust-lang/libc", tag = "0.2.88" }
```

В этом примере мы собираемся обратиться к Git-репозиторию для `libc` и явным образом указать тег `0.2.88`. Имеющийся в файле `Cargo.toml` раздел `patch` обеспечивает исправление именно реестра `crates.io`, а не самого пакета. По сути, все вышестоящие зависимости для `libc` заменяются при этом вашей собственной версией.

Использовать такую функциональную возможность нужно осмотрительно и только при вполне конкретных обстоятельствах. Она не оказывает никакого влияния на нижестоящие зависимости, так что никакие крейты, зависящие от вашего крейта, не смогут унаследовать сделанное исправление. Пока разумного обходного решения для этого Cargo-ограничения не существует. В тех случаях, когда вам потребуется получить более серьезный контроль над зависимостями второго и третьего порядка, надо будет либо создать ответвления всех задействованных проектов, либо включить их непосредственно в свой собственный проект в виде подпроектов, используя для этого *рабочие пространства* (рассматриваемые далее в этой главе).

2.4.2. Лучшие методы корректировки зависимостей

Существует ряд правил, которых следует придерживаться при корректировке зависимостей:

- исправление зависимостей должно быть крайней мерой, поскольку со временем такие исправления будет трудно сопровождать;
- для проектов с открытым исходным кодом, особенно если этого требуют лицензии (например, код под лицензией GPL), исправления с требуемыми изменениями нужно при необходимости отправлять в исходное расположение;
- разветвления исходных крейтов следует избегать, а в тех случаях, когда без этого просто не обойтись, надо постараться вернуться на главную ветвь как можно быстрее. Долговременные разветвления будут иметь существенные различия и в конечном итоге могут стать кошмаром при сопровождении.

2.5. Публикация крейтов

Для проектов, готовых к публикации на [crates.io](#), процесс прост. Как только крейт будет готов к запуску, можно выполнить команду `cargo publish`, а о деталях уже позаботится сам инструмент Cargo. При публикации крейтов следует придерживаться ряда требований, среди которых: указание лицензии, предоставление конкретных данных проекта в виде документации и URL-адреса репозитория, а также обеспечение для [crates.io](#) доступности всех зависимостей.

Можно публиковать крейты в закрытом реестре, но на момент подготовки книги поддержка Cargo закрытых реестров носила весьма ограниченный характер. Поэтому, чтобы полагаться для закрытых крейтов на [crates.io](#), рекомендуется воспользоваться закрытыми Git-репозиториями и тегами.

2.5.1. CI/CD-интеграция

Для большинства крейтов хочется выстроить систему для автоматической публикации выпусков в [crates.io](#). Системы непрерывной интеграции и непрерывного развертывания (Continuous integration/continuous deployment, CI/CD) — обычная составляющая современных циклов разработки. Как правило, они представлены двумя самостоятельными этапами:

- непрерывной интеграции (CI) — система, выполняющая компиляцию, управление и проверку каждой отправки (коммита) в VCS-репозиторий;
- непрерывного развертывания (CD) — система, проводящая автоматическое развертывание каждой отправки (коммита) или выпуска (релиза), при условии прохождения им всех необходимых проверок со стороны CI.

Давайте в целях демонстрации пройдемся по проекту `dryok`, использующему сервис GitHub Actions⁵, находящийся в свободном доступе для проектов с открытым кодом.

Перед просмотром кода обратим внимание на описание процесса выпуска с технологическим потоком Git, соответствующее моменту принятия решения по публикации релиза:

1. Обновление при необходимости атрибута `version` в `Cargo.toml` с указанием выпускаемой версии.
2. В результате запустится система CI, контролирующая прохождение всех тестов и проверок.
3. Создание и внедрение вами тега для релиза (с использованием префикса версии — например, с помощью команды `git tag -s vX.Y.Z`).
4. В результате запустится система CD, создающая намеченный релиз и выполняющая его публикацию в [crates.io](#) с помощью команды `cargo publish`.

⁵ См. <https://github.com/features/actions>.

5. Обновление атрибута `version` в `Cargo.toml` для следующего цикла релиза в новой отправке.

ПРИМЕЧАНИЕ

После публикации в `crates.io` выполнить откат или внести изменения будет уже нельзя. Опубликованные крейты не подлежат изменению, поэтому для внесения в них любых изменений придется повторить весь процесс заново.

Давайте рассмотрим крейт `dryok`, в котором описанная схема реализуется с использованием GitHub Actions.

Необходимо обратить внимание на два действия из состава Action:

- `.github/workflows/build-and-test.yml` — создающее и запускающее тесты для сочетания функциональностей, платформ и инструментария⁶;
- `.github/workflows/publish.yml` — создающее и запускающее тесты для намеченного релиза в соответствии со схемой `v*` и публикующее крейт в `crates.io`⁷.

В листинге 2.4 показаны параметры задания сборки, включая матрицу функциональности, канала и платформы. Для настройки среды Rust этими заданиями используется действие «все в одном» `brndnmthws/rust-action` из GitHub Action⁸.

Листинг 2.4. Фрагмент кода для `.github/workflows/build-and-test.yml`

```
name: Build & test
on:
  push:
    branches: [main]
  pull_request:
    branches: [main]
env:
  CARGO_TERM_COLOR: always
concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true
jobs:
  build:
    strategy:
      matrix:
        rust:
          - stable
          - beta
          - nightly
(1)
(2)
```

⁶ См. <http://mng.bz/VRmP>.

⁷ См. <http://mng.bz/xjAW>.

⁸ См. <http://mng.bz/A87z>.

```

features:                                     (3)
  - serde
  - base64
  - simd_backend
  - default

os:                                         (4)
  - ubuntu-latest
  - macos-latest
  - windows-latest

exclude:                                      (5)
  - rust: stable
    features: simd_backend
  - rust: beta
    features: simd_backend
  - os: windows-latest
    features: simd_backend

```

Здесь:

- в поз. (1) — сборки будут запускаться только при отправке изменений в Git и запросах на извлечение для основной ветви;
- в поз. (2) — запуск со стабильного, бета и ночного канала;
- в поз. (3) — запуск тестов, имеющих разные функции, включаемые по отдельности;
- в поз. (4) — запуск под Linux, macOS и Windows;
- в поз. (5) — некоторые комбинации сборок не работают, поэтому здесь они отключены.

В листинге 2.5 показаны отдельные этапы по сборке, тестированию, форматированию и запуску инструмента Clippy (рассматривается в главе 3).

Листинг 2.5. Фрагмент кода для .github/workflows/build-and-test.yml

```

runs-on: ${{ matrix.os }}

env:
  FEATURES: >
    ${{ matrix.rust != 'nightly' && matrix.features
      || format('{0},nightly', matrix.features) }}

steps:
  - uses: actions/checkout@v3
  - name: Setup ${{ matrix.rust }} Rust toolchain with caching.
    uses: brndnmthws/rust-action@v1
    with:
      toolchain: ${{ matrix.rust }}                               (1)
  - run: cargo build --features ${{ env.FEATURES }}           (2)
  - run: cargo test --features ${{ env.FEATURES }}            (3)

```

```
env:  
  RUST_BACKTRACE: 1  
- run: cargo fmt --all -- --check  
  if: ${{ matrix.rust == 'nightly' && matrix.os == 'ubuntu-latest' }}  
- run: cargo clippy --features ${{ env.FEATURES }} -- -D warnings      (5)
```

Здесь:

- в поз. (1) — на этом этапе устанавливается нужный набор инструментов;
- в поз. (2) — запуск сборки с указанной функциональностью;
- в поз. (3) — запуск всех тестов с указанными характеристиками;
- в поз. (4) — проверка форматирования кода;
- в поз. (5) — запуск Clippy с проверкой указанных функций.

В листинге 2.6 показаны этапы, необходимые для публикации нашего крейта.

Листинг 2.6. Код для .github/workflows/publish.yml

```
name: Publish to crates.io  
on:  
  push:  
    tags:  
      - v*                                         (1)  
env:  
  CARGO_TERM_COLOR: always  
jobs:  
  build-test-publish:  
    runs-on: ubuntu-latest  
    steps:  
      - uses: actions/checkout@v3  
      - uses: brndnmtthws/rust-action@v1  
        with:  
          toolchain: stable  
      - run: cargo build  
      - run: cargo test  
      - run: cargo login -- ${{ secrets.CRATES_IO_TOKEN }}      (2)  
      - run: cargo publish                                (3)  
      - name: Create Release                            (4)  
        id: create_release  
        uses: softprops/action-gh-release@v1  
        if: startsWith(github.ref, 'refs/tags/')  
        with:  
          draft: false  
          prerelease: false  
          discussion_category_name: General  
          generate_release_notes: true
```

Здесь:

- в поз. (1) — работает, только если тег соответствует `v*`;
- в поз. (2) — вход в `crates.io` с применением секрета (токена), сохраненного в конфигурации секретов репозитория. Этот токен хранится с использованием функции хранения секретов GitHub, которая должна быть предоставлена заранее;
- в поз. (3) — публикация крейта на <https://crates.io>;
- в поз. (4) — создание релиза на GitHub.

ПРИМЕЧАНИЕ

Действия GitHub Actions пока что не поддерживают никаких способов управления пропуском релиза при использовании отдельных этапов (т. е. ожидания успешного завершения этапа сборки перед переходом к этапу развертывания). Чтобы управлять этим процессом, перед отправкой каких-либо тегов нужно убедиться, что этап сборки был пройден успешно.

На последнем этапе публикации следует предоставить токен для <https://crates.io>. Сделать это можно путем создания учетной записи `crates.io`, сгенерировав токен из настроек учетной записи `crates.io` и добавив его после этого в секретное хранилище GitHub в настройках вашего репозитория GitHub.

2.6. Ссылки на библиотеки C

Порой бывает необходимо воспользоваться внешними библиотеками кода, не имеющего отношения к Rust. Обычно это достигается применением интерфейса внешней функции (Foreign Function Interface, FFI). Этот интерфейс считается стандартным способом обеспечения межъязыковой совместимости (более подробно FFI-интерфейс будет описан в главе 4).

Давайте рассмотрим простой пример вызова функций из одной из самых популярных библиотек языка C — zlib. Выбор на эту библиотеку пал по причине ее повсеместной распространенности, и приводимый здесь пример должен без проблем работать «из коробки» на любой платформе, где доступна zlib. В Rust мы реализуем две ее функции: `compress()` и `uncompress()`. Определения из zlib-библиотеки выглядят, как показано в листинге 2.7 (они здесь упрощены в целях использования в нашем примере).

Листинг 2.7. Упрощенный код из файла `zlib.h`

```
int compress(void *dest, unsigned long *destLen,
            const void *source, unsigned long sourceLen);

unsigned long compressBound(unsigned long sourceLen);

int uncompress(void *dest, unsigned long *destLen,
              const void *source, unsigned long sourceLen);
```

Сначала определим в Rust C-интерфейс, воспользовавшись ключевым словом `extern` (листинг 2.8).

Листинг 2.8. Код для полезных функций из zlib

```
use libc::(c_int, c_ulong);

#[link(name = "z")]
extern "C" {
    fn compress(
        dest: *mut u8,
        dest_len: *mut c_ulong,
        source: *const u8,
        source_len: c_ulong,
    ) -> c_int;
    fn compressBound(source_len: c_ulong) -> c_ulong;
    fn uncompress(
        dest: *mut u8,
        dest_len: *mut c_ulong,
        source: *const u8,
        source_len: c_ulong,
    ) -> c_int;
}
```

В качестве зависимости здесь была включена библиотека `libc`, предоставляющая С-совместимые типы в Rust. При каждом случае ссылок на библиотеки С надо для поддержки совместимости использовать типы из `libc`. Невыполнение этого требования может привести к неопределенному поведению. Из `zlib` нами здесь определены три полезные функции: `compress`, `compressBound` и `uncompress`.

Атрибут `link` сообщает `rustc` о необходимости связать указанные функции с `zlib`, что равносильно добавлению при компоновке флага `-lz`. Под macOS это, как показано в следующем коде, можно проверить с помощью команды `otool -L` (под Linux используется `ldd`, а под Windows — `dumpbin`):

```
$ otool -L target/debug/zlib-wrapper
target/debug/zlib-wrapper:
/usr/lib/libz.1.dylib (compatibility version 1.0.0, current version 1.2.11)
/usr/lib/libiconv.2.dylib (compatibility version 7.0.0, current version 7.0.0)
/usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 1292.60.1)
/usr/lib/libresolv.9.dylib (compatibility version 1.0.0, current version 1.0.0)
```

Затем нужно написать функции на языке Rust, которые послужат обертками для С-функций и смогут быть вызванными из кода Rust. Вызов С-функций напрямую считается в Rust небезопасным, поэтому такой вызов нужно заключать в блок `unsafe {}` (листинг 2.9).

Листинг 2.9. Код для zlib_compress

```
pub fn zlib_compress(source: &[u8]) -> Vec<u8> {
    unsafe {
        let source_len = source.len() as c_ulong;

        let mut dest_len = compressBound(source_len);           (1)
        let mut dest = Vec::with_capacity(dest_len as usize);   (2)

        compress(                                              (3)
            dest.as_mut_ptr(),
            &mut dest_len,
            source.as_ptr(),
            source_len,
        );
        dest.set_len(dest_len as usize);
        dest                                         (4)
    }
}
```

Здесь:

- в поз. (1) — возвращение верхней границы длины сжатого вывода;
- в поз. (2) — выделение в куче dest_len байта с помощью Vec;
- в поз. (3) — вызов C-функции из zlib;
- в поз. (4) — возвращение результата в виде Vec.

Версия zlib_uncompress практически идентична приведенной в предыдущем примере функции, за исключением того, что для целевого буфера следует указывать нашу собственную длину. А теперь можно продемонстрировать их использование, показанное в листинге 2.10.

Листинг 2.10. Код для main()

```
fn main() {
    let hello_world = "Hello, world!".as_bytes();
    let hello_world_compressed = zlib_compress(&hello_world);
    let hello_world_uncompressed =
        zlib_uncompress(&hello_world_compressed, 100);
    assert_eq!(hello_world, hello_world_uncompressed);
    println!(
        "{}",
        String::from_utf8(hello_world_uncompressed)
            .expect("Invalid characters")
    );
}
```

Самые большие трудности при работе с FFI возникают из-за сложности некоторых API языка C и из-за сопоставления различных типов и функций. Для их преодоления можно воспользоваться rust-инструментом `bindgen`, более подробно рассматриваемым в главе 4.

2.7. Бинарный дистрибутив

Бинарные файлы Rust состоят из всех зависимостей Rust для текущей платформы, собранных (за исключением среды выполнения C) в виде одного файла в двоичном формате, включающего любые не-Rust библиотеки, которые могут быть динамически скомпонованы. Впрочем, можно создавать бинарные файлы, которые будут статически связаны со средой выполнения C, но по умолчанию это опционально. Таким образом, при дистрибуции относящихся к Rust бинарных файлов следует рассмотреть вопрос — что вам лучше: статически связать среду выполнения с C или же положиться на среду выполнения системы.

Сами по себе бинарные файлы зависят от используемой платформы. Они могут пройти кросс-компиляцию для различных платформ, но смешивать различные архитектуры или платформы в одном и том же бинарном Rust-файле невозможно. Бинарный файл, скомпилированный для центральных процессоров на основе Intel x64-64, не будет запускаться на платформах на основе ARM вроде AArch64 (известной также как ARMv8) без тех или иных вариантов эмуляции. Бинарный файл, скомпилированный для работы под macOS, не станет запускаться под Linux.

Некоторые поставщики операционных систем, в частности macOS от Apple, предоставляют средства для эмуляции других платформ центральных процессоров. При помощи инструментального средства Rosetta из арсенала Apple можно добиться автоматического запуска бинарных файлов, скомпилированных для работы на платформе x86-64. Подробности дистрибуции бинарных файлов под macOS можно узнать из документации разработчиков компании Apple⁹. В большинстве случаев вам потребуются исходные настройки для текущей используемой платформы, но из этого правила всё же есть исключения.

Если вы перешли на Rust с такого языка, как Go, то, возможно, вам привычнее будет выполнять дистрибуцию предварительно скомпилированных бинарных файлов, не беспокоясь о среде выполнения C. В отличие от Go, языку Rust требуется среда выполнения C, и по умолчанию используется динамическая компоновка.

2.7.1. Кросс-компиляция

Для кросс-компиляции бинарных файлов для различных целевых платформ можно воспользоваться Cargo, но только там, где эти платформы поддерживаются компилятором.

⁹ См. <http://mng.bz/ZRvP>.

Например, в Windows можно легко скомпилировать бинарные файлы для Linux, но скомпилировать бинарные файлы для Windows в Linux не так-то просто (но всё же возможно).

Список доступных на вашей платформе целевых платформ можно получить с помощью `rustup`:

```
$ rustup target list
rustup target list
aarch64-apple-darwin
aarch64-apple-ios
aarch64-fuchsia
aarch64-linux-android
aarch64-pc-windows-msvc
..
..
```

Команда `rustup target add <target>` позволяет устанавливать различные цели, после чего сборку для конкретной целевой платформы можно провести с помощью команды `cargo build --target <target>`. Например, на моем компьютере с macOS на базе процессора Intel для компиляции бинарных файлов под AArch64 (используется чип M1) можно запустить следующую команду:

```
$ rustup target add aarch64-apple-darwin
info: downloading component 'rust-std' for 'aarch64-apple-darwin'
info: installing component 'rust-std' for 'aarch64-apple-darwin'
info: using up to 500.0 MiB of RAM to unpack components
18.3 MiB / 18.3 MiB (100 %) 14.7 MiB/s in 1s ETA: 0s
$ cargo build --target aarch64-apple-darwin
...
Finished dev [unoptimized + debuginfo] target(s) in 3.74s
```

Но если попытаться запустить полученный бинарный файл на выполнение, ничего не получится:

```
$ ./target/aarch64-apple-darwin/debug/simple-project
-bash: ./target/aarch64-apple-darwin/debug/simple-project: Bad CPU type in executable
```

Однако при наличии доступа к компьютеру на macOS с процессором AArch64 этот бинарный файл можно было бы скопировать на него и успешно запустить там на выполнение.

2.7.2. Создание статически связанных бинарных файлов

Обычные бинарные файлы Rust включают в себя все скомпилированные зависимости, за исключением, как уже отмечалось, библиотеки среды выполнения C. В Windows и macOS считается нормой распространять предварительно скомпилированные бинарные файлы и ссылаться на библиотеки среды выполнения C, работающей под той или иной конкретной операционной системой. А в Linux большинство пакетов компилируются из исходных кодов разработчиками дистрибутивов, и ответственность за управление средой выполнения C возлагается на дистрибутивы.

При распространении бинарных файлов Rust под Linux можно, в зависимости от личных предпочтений, воспользоваться библиотеками glibc или musl. Библиотека glibc на языке C используется в большинстве дистрибутивов Linux по умолчанию. Но когда при распространении бинарных файлов под Linux желательно обеспечить максимальную переносимость, я рекомендую создавать их статическую связь с помощью musl. Дело в том, что при попытке статической компоновки на известных целевых объектах Rust по умолчанию предполагает, что вы воспользуетесь musl.

ПРИМЕЧАНИЕ

В некоторых случаях musl ведет себя иначе, чем glibs. Эти отличия задокументированы в [musl-вики¹⁰](#).

С помощью флага `target-feature` можно указать `rustc` на необходимость использования статической среды выполнения С:

```
$ RUSTFLAGS="-C target-feature=+crt-static" cargo build
   Finished dev [unoptimized + debuginfo] target(s) in 0.01s
```

В этом коде `-C target-feature=+crt-static` передается `rustc` через переменную среды окружения `RUSTFLAGS`, которая интерпретируется `Cargo` и передается `rustc`.

А следующий код служит для статического связывания с `musl` на ОС Linux с процессором `x86-64`:

```
$ rustup target add x86_64-unknown-linux-musl          (1)
...
$ RUSTFLAGS="-C target-feature=+crt-static" cargo build --target
x86_64-unknown-linux-musl                           (2)
```

...

Здесь:

- в поз. (1) — убедиться, что `musl` target установлен;
- в поз. (2) — компилировать с использованием `musl` target и принудительно использовать статическую среду выполнения С.

Для явного отключения статического связывания воспользуйтесь в приведенном коде следующим значением переменной среды окружения:

```
RUSTFLAGS="-C target-feature=-crt-static"
```

заменив плюс + минусом -.

Это может пригодиться для целевых сред, изначально использующих статическое связывание, — но если вы в этом не уверены, задействуйте параметры по умолчанию.

Кроме того, `rustc`-флаги для `Cargo` можно указать с использованием `~/.cargo/config`:

```
[target.x86_64-pc-windows-msvc]
rustflags = ["-Ctarget-feature=+crt-static"]
```

¹⁰ См. <http://mng.bz/Rm7K>.

После добавления приведенного кода в `~/.cargo/config` он даст `rustc` команду выполнить статическое связывание при использовании целевой среды `x86_64-pc-windows-msvc`.

2.8. Документирование Rust-проектов

Изначально поставляемое с Rust средство документирования кода называется `rustdoc`. Если вам уже приходилось применять средства документирования кода из других проектов (такие, например, как Javadoc, `docstring` или RDoc), разобраться в `rustdoc` будет нетрудно.

Использование `rustdoc` сводится к добавлению в код комментариев и генерированию документации. Рассмотрим простой пример и начнем с создания библиотеки:

```
$ cargo new rustdoc-example --lib
   Created library `rustdoc-example` package
```

После чего отредактируем файл `src/lib.rs`, добавив в него функцию под названием `mult`, принимающую два целочисленных значения (`a` и `b`) и перемножающую их. Добавим также в наш пример и тест:

```
pub fn mult(a: i32, b: i32) -> i32 {
    a * b
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        assert_eq!(2 * 2, mult(2, 2));
    }
}
```

Пока сюда не добавлено никакой документации. Перед тем как это сделать, давайте сгенерируем с помощью Cargo пустую документацию:

```
$ cargo doc
Documenting rustdoc-example v0.1.0
(/Users/brenden/dev/code-like-a-pro-in-rust/code/c2/2.8/rustdoc-example)
Finished dev [unoptimized + debuginfo] target(s) in 0.89s
```

Теперь в `target/` можно увидеть сгенерированную документацию в формате HTML. При желании открыть документацию в браузере вы можете запустить файл `target/doc/src/rustdoc_example/lib.rs.html` и просмотреть его содержимое. Оно должно быть таким, как показано на рис. 2.1. Исходная документация ничего не содержит, но в ней в перечислении можно увидеть открытую функцию `mult`.

А теперь давайте добавим к нашему проекту атрибут компилятора и немного документации. Обновите содержимое файла `src/lib.rs`, придав ему следующий вид:

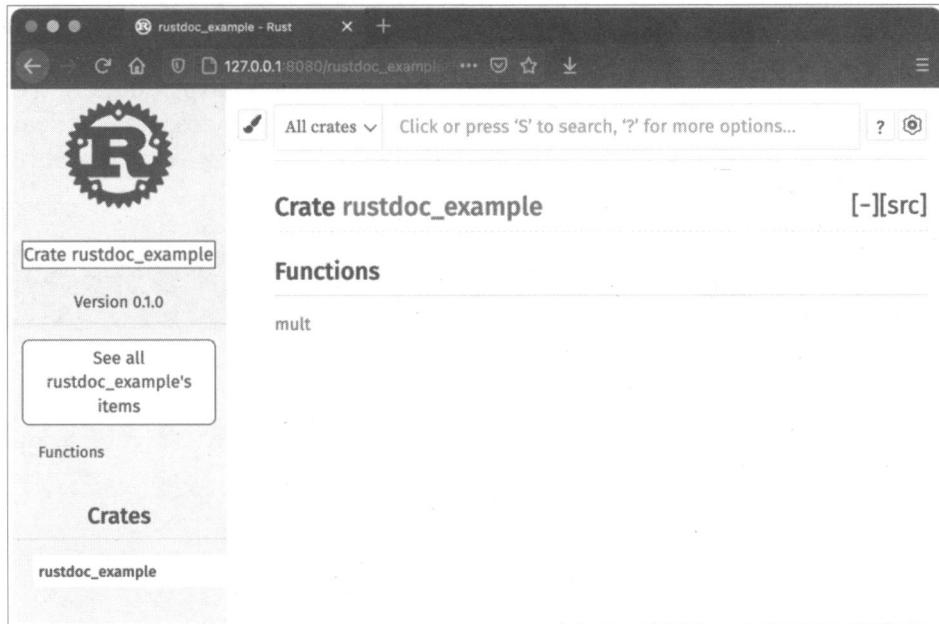


Рис. 2.1. Страница с пустым HTML-выводом rustdoc

```
//! # rustdoc-example          (1)
//!
//! A simple project demonstrating the use of rustdoc with the function (1)
//! ['mult'].

#![warn(missing_docs)]          (2)

/// Returns the product of `a` and `b`.          (3)
pub fn mult(a: i32, b: i32) -> i32 {
    a * b
}
```

Здесь:

- в поз. (1) — строки документа крейт-уровня, отображаемые на первой странице документов крейта;
- в поз. (2) — этот атрибут компилятора предписывает rustc выдавать предупреждение, если у открытых функций, модулей или типов отсутствует документация;
- в поз. (3) — этот комментарий содержит документацию по функции `mult`

СОВЕТ

При форматировании Rust-документации используется язык CommonMark¹¹, являющийся разновидностью языка разметки Markdown.

¹¹ См. <https://commonmark.org/help>.

Если перезапустить команду cargo doc с только что созданной кодовой документацией и открыть эту документацию в браузере, отобразится страница, показанная на рис. 2.2. Для крейтов, опубликованных в crates.io, на сайте <https://docs.rs> имеется сопутствующий rustdoc-узел, где происходит автоматическое создание и хранение документации по крейтам. Например, документация по крейту dryoc находится по адресу <https://docs.rs/dryoc>.

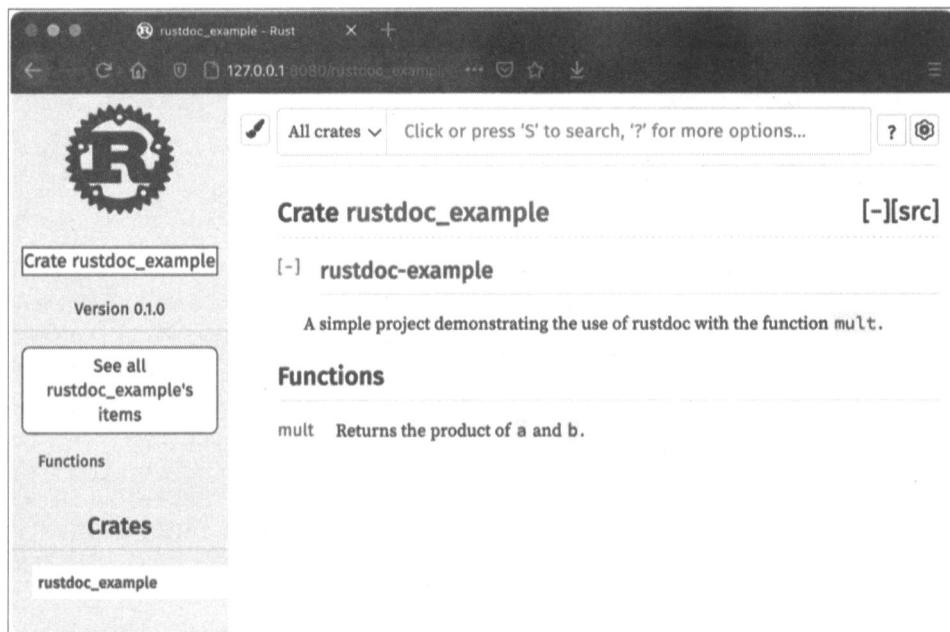


Рис. 2.2. Страница HTML-вывода rustdoc с комментариями

Для задокументированных крейтов можно обновить файл Cargo.toml, включив в него свойство documentation, которое станет ссылкой на документацию проекта. Это поможет тем, кто ищет свой крейт в источниках наподобие crates.io. Например, для крейта dryoc в Cargo.toml есть следующий раздел:

```
[package]
name = "dryoc"
documentation = "https://docs.rs/dryoc"
```

Чтобы воспользоваться doc.rs не нужно больше ничего другого. При публикации на crates.io обновленная документация на веб-сайте создается автоматически. Краткий справочник по синтаксису rustdoc приведен в табл. 2.4.

Таблица 2.4. Краткий справочник по использованию rustdoc

Синтаксис	Тип	Описание
//!	Строка документа	Документация на уровне крейта или модуля, находящаяся в их начале. В ней используется CommonMark

Таблица 2.4 (окончание)

Синтаксис	Тип	Описание
///	Строка документа	Документация модуля, функции, типажа или типа, следующая за обозначением комментария. В ней используется CommonMark
[func], [`func`], [Foo](Bar)	Ссылка	Ссылки в документации на функцию, модуль или другой тип. Для надлежащей ссылки ключевое слово должно быть в области видимости rustdoc. Для ссылки доступно множество вариантов, узнать о них можно из документации по rustdoc

2.8.1. Примеры кода в документации

Одним из полезных свойств rustdoc является то, что код, включенный в документацию в качестве примеров, проходит компиляцию и выполняется в виде комплексных тестов. То есть у нас имеется возможность включить примеры кода с проходящими тестированием утверждениями при запуске команды cargo test. Это способствует созданию высококачественной документации с рабочими образцами кода.

Один из примеров такого кода (добавленный к документации уровня крейта в файл `src/lib.rs` из предыдущего примера) может иметь следующий вид:

```
///! # Example
///
///! -----
///! use rustdoc_example::mult;
///! assert_eq!(mult(10, 10), 100);
///! -----
```

При запуске тестов с помощью команды cargo test будет получен следующий результат:

```
cargo test
Compiling rustdoc-example v0.1.0
(/Users/brenden/dev/code-like-a-pro-in-rust/code/c2/2.8/rustdoc-example)
Finished test [unoptimized + debuginfo] target(s) in 0.42s
    Running target/debug/deps/rustdoc_example-bec4912aee60500b
```

```
running 1 test

test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests rustdoc-example

running 1 test
test src/lib.rs - (line 7) ... ok
```

```
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.23s
```

Дополнительные сведения об использовании средства `rustdoc` можно получить из его официальной документации¹², а узнать подробности о применении CommonMark позволит обращение к странице помощи этого языка¹³.

2.9. Модули

Модули Rust предоставляют возможность иерархической организации кода в виде набора отдельных частей, которые при желании можно разбить на самостоятельные исходные файлы. В модулях Rust два свойства объединяют в одно целое: включение кода из других файлов-источников и указание пространства имен повсеместно видимых обозначений. По умолчанию все обозначения в Rust объявляются закрытыми, но они могут экспортirоваться (или превращаться в повсеместно видимые) с помощью ключевого слова `pub`. Когда приходится экспортirовать большое число обозначений, со временем может возникнуть коллизия имен. Поэтому, чтобы не захламлять пространство имен, можно распределить код по модулям.

В блоке объявления модуля используется ключевое слово `mod` с необязательным указателем видимости `pub`, за которым тут же следует блок кода в фигурных скобках:

```
mod private_mod {
    /...           ← сюда будет помещен закрытый код
}
pub mod public_mod {
    /...           ← сюда будет помещен открытый код для экспорта
}
```

Иногда при упоминании кода на Rust термины `module` и `mod` используются как взаимозаменяемые. По соглашению в именах модулей обычно используется «змеиный» регистр, а в большинстве других имен (структур, перечислений, типажей) — «верблюжий». Элементарные типы (`i32`, `str`, `u64` и т. д.) обычно обозначаются короткими односложными словами и иногда пишутся с использованием «змеиного» регистра. Константы, как правило, обозначаются символами в верхнем регистре, что соответствует соглашениям, принятым в большинстве других языков программирования. Следование этим установкам упрощает опознавание импортируемых объектов, для чего бывает вполне достаточно взглянуть на содержимое инструкции `use`.

Модуль можно включить с помощью того же ключевого слова `mod`, закончив объявление не блоком кода, а точкой с запятой:

```
mod private_mod;
pub mod public_mod;
```

¹² См. <https://doc.rust-lang.org/rustdoc>.

¹³ См. <https://commonmark.org/help>.

Модули могут иметь глубокую вложенность:

```
mod outer_mod {
    mod inner_mod {
        mod super_inner_mod {
            ...
        }
    }
}
```

Спецификаторы видимости

По умолчанию в Rust с точки зрения видимости закрыто всё, за исключением открытых типажей и открытых перечислений, где связанные элементы по умолчанию являются открытыми. Объявления с закрытой областью видимости привязаны к модулю, а это означает, что к ним можно получить доступ из модуля (и подмодулей), в котором они объявлены.

Применение ключевого слова `pub` меняет видимость на открытую за счет необязательного модификатора: `pub` (модификатор), который можно использовать с `crate`, `self`, `super` или же `in path` с указанием пути к другому модулю. Проще говоря, `pub(crate)` указывает, что элемент является открытым в крейте, но недоступен за его пределами.

Если сам модуль так же не является открытым, то всё объявленное внутри него не экспортируется за пределы области видимости крейта. Например, в показанном далее фрагменте кода имеются две открытые функции, но в рассматриваемом случае за пределами крейта будет видна только `public_mod_fn()`:

```
mod private_mod {
    pub fn private_mod_fn() {}
}

pub mod public_mod {
    pub fn public_mod_fn() {}
}
```

В добавок к этому находящийся в открытом модуле элемент с закрытой областью видимости останется закрытым и недоступным за пределами своего крейта.

Видимость в Rust вполне понятна на интуитивном уровне и, кроме этого, помогает предотвращать случайное появление «дырявых» абстракций. За более подробными сведениями о видимости в Rust следует обратиться к справочнику по этому языку¹⁴.

Для включения обозначения или модуля из другого крейта используется инструкция `use`:

`use serde::ser::{Serialize, Serializer};`

включающая обозначения `Serialize` и `Serializer` из модуля `ser` в крейт `serge`.

При включении кода с помощью инструкции `use` первым именем обычно указывается крейт, из которого его нужно включить, а затем следует имя модуля, конкрет-

¹⁴ См. <http://mng.bz/2710>.

ные обозначения или групповой символ (*), чтобы включить из модуля все обозначения.

Модули также могут быть упорядочены с использованием файловой системы. Точно такую же иерархию, как в предыдущем примере, можно создать, воспользовавшись путями в исходном каталоге нашего крейта, но Cargo всё же следует уведомить, какие именно файлы нужно включить в крейт. Для этого используется не блок, а инструкция `mod`. Рассмотрим крейт со следующей структурой:

```
$ tree .
.
├── Cargo.lock
└── Cargo.toml
└── src
    ├── lib.rs
    ├── outer_module
    │   └── inner_module
    │       ├── mod.rs
    │       └── super_inner_module.rs
    └── outer_module.rs
```

3 directories, 6 files

В этом коде показан крейт с тремя вложенными внутренними модулями. В наш файл верхнего уровня `lib.rs` будет включен внешний модуль, определение которого находится в файле `outer_module.rs`:

```
mod outer_module;
```

Компилятор будет искать объявление `mod` либо в `outer_module.rs`, либо в `outer_module/mod.rs`. В нашем случае `outer_module.rs` представлен на том же уровне, что и `lib.rs`. Для включения внутреннего модуля в `outer_module.rs` имеется следующая инструкция:

```
mod inner_module;
```

Затем во внешнем модуле компилятор станет искать `inner_module.rs` или `inner_module/mod.rs`. В нашем случае он находит файл `inner_module/mod.rs`, содержащий следующую строку:

```
mod super_inner_module;
```

в результате чего `super_inner_module.rs` включается в каталог `inner_module`. Может показаться, что это несколько сложнее того примера, который был показан в этом разделе ранее, но для более крупных проектов намного полезнее будет использование модулей, чем включение всего исходного кода для крейта либо в `lib.rs`, либо в `main.rs`.

Если разобраться с модулями вам пока трудно, то для понимания того, как части соединяются друг с другом, попробуйте воссоздать аналогичные структуры с нуля. Начать можно с примера, содержащегося в каталоге `c02/modules` исходного кода этой книги. Модульная структура будет дополнительно рассмотрена в главе 9.

ПРИМЕЧАНИЕ

Напомню, что код примеров, включенных в книгу, доступен для загрузки с веб-сайта издательства Manning¹⁵, а также с GitHub¹⁶. Копия этого кода может быть также скачана вами на свой компьютер с помощью следующей Git-команды:

```
$ git clone https://github.com/brndnmthws/code-like-a-pro-in-rust-book
```

2.10. Рабочие пространства

Свойство Cargo `workspace` позволяет разбить большой крейт на несколько обособленных крейтов и групп таких крейтов в рабочем пространстве, совместно использующем один и тот же lock-файл `Cargo.lock`. У рабочих пространств имеется ряд важных свойств, рассматриваемых в этом разделе, но их главным свойством является разрешение использования общих параметров из `Cargo.toml` и разрешение дерева зависимостей из одного и того же файла `Cargo.lock`. Каждый из проектов, относящихся к рабочему пространству, использует совместно с другими проектами следующее:

- файл верхнего уровня `Cargo.lock`;
- выходной каталог `output/`, в котором содержатся цели проекта из всех рабочих пространств;
- разделы `[patch]`, `[replace]` и `[profile.*]` из `Cargo.toml` верхнего уровня.

Для использования рабочих пространств проекты с помощью Cargo создаются как обычно в подкаталогах, которые не пересекаются с каталогами крейта верхнего уровня (т. е. они не должны находиться в каталогах `src/`, `target/`, `tests/`, `examples/`, `benches/` и т. д.).

Затем вы можете добавить эти зависимости, как делается обычно, — за исключением того, что вместо указания версии или репозитория вы просто указываете путь или добавляете каждый проект в список `workspace.members` в `Cargo.toml`.

А теперь давайте рассмотрим пример проекта с использованием рабочих пространств. Начнем с создания приложения верхнего уровня и перейдем в только что созданный каталог:

```
$ cargo new workspaces-example  
Created binary (application) `workspaces-example` package  
$ cd workspaces-example
```

Теперь создадим подпроект в виде простой библиотеки:

```
$ cargo new subplot --lib
```

¹⁵ См. <https://www.manning.com/books/code-like-a-pro-in-rust>.

¹⁶ См. <https://github.com/brndnmthws/code-like-a-pro-in-rust-book>.

Вновь созданная структура каталогов должна выглядеть следующим образом:

```
$ tree
.
├── Cargo.toml
├── src
│   └── main.rs
└── subproject
    ├── Cargo.toml
    └── src
        └── lib.rs
3 directories, 4 files
```

Далее давайте обновим Cargo.toml верхнего уровня, включив в него подпроект путем добавления его в качестве зависимости (в рабочих пространствах всё же нужно определять зависимости):

```
[dependencies]
subproject = { path = "./subproject" }
```

В этом коде подпроект добавляется путем указания в виде зависимости и использования для него свойства path. Чтобы включить проект в рабочее пространство, нужно также добавить его в workspace.members, где для компонентов рабочего пространства содержится список путей или шаблон glob. Для более крупных проектов вместо явного перечисления каждого пути при условии, что используется согласованная иерархия путей, проще может оказаться использование glob. Для нашего примера код рабочего пространства в Cargo.toml будет выглядеть следующим образом:

```
[workspace]
members = ["subproject"]
```

Теперь, чтобы убедиться, что всё компилируется без ошибок, можно запустить cargo check. Пока что в нашем проекте верхнего уровня код из подпроекта не используется, поэтому давайте добавим функцию, которая возвращает "Hello, world!", и вызовем ее из нашего приложения. Сначала обновим файл subproject/src/lib.rs, включив в него нашу функцию hello_world:

```
pub fn hello_world() -> String {
    String::from("Hello, world!")
}
```

Теперь, чтобы вызвать эту функцию, обновим файл src/main.rs в приложении верхнего уровня:

```
fn main() {
    println!("()", subproject::hello_world());
}
```

И, наконец, запустим наш новый код:

```
$ cargo run
Compiling subproject v0.1.0
(/Users/brenden/dev/code-like-a-pro-in-rust/code/c2/2.9/workspaces-example/subproject)
```

```
Compiling workspaces-example v0.1.0 (/Users/brenden/dev/
  ➔ code-like-a-pro-in-rust/code/c2/2.9/workspaces-example)
  Finished dev [unoptimized + debuginfo] target(s) in 0.85s
    Running `target/debug/workspaces-example`
Hello, world!
```

Эти шаги можно повторить с любым количеством подпроектов, подставляя для каждого вхождения подпроекта в предыдущем коде другое имя. Полный код для этого примера можно найти в каталоге c02/workspaces-example.

Совет

Cargo также поддерживает *виртуальные манифести*, являющиеся крейтами верхнего уровня, которые не определяют раздел `package` в `Cargo.toml` и содержат только подпроекты. Это пригодится, когда потребуется опубликовать коллекцию пакетов в одном крейте верхнего уровня.

Рабочие пространства используются многими крейтами для разбивки проектов. Дополнительной особенностью рабочих пространств является то, что каждый подпроект может быть опубликован как его собственный отдельный крейт для использования другими компонентами.

В качестве пары примечательных примеров проектов, использующих функциональность рабочих пространств, можно привести крейты `rand`¹⁷ и `Rocket`¹⁸, в последнем из которых задействован виртуальный манифест. Доступен для изучения и полный справочник по рабочим пространствам Cargo¹⁹.

2.11. Пользовательские сценарии сборки

В Cargo предоставляется функциональность, проявляющаяся в ходе проведения сборки и позволяющая указывать операции, проводимые в это время в Rust-сценарии. В сценарии содержится одна-единственная функция `main` плюс любой другой требующий включения код, включая зависимости сборки, указываемые в специальном разделе `build-dependencies` файла `Cargo.toml`. Сценарий обменивается данными с Cargo, выводя отформатированные команды в `stdout`, а Cargo будет их интерпретировать и выполнять.

ПРИМЕЧАНИЕ

Стоит заметить, что так называемый *сценарий* не является таковым в смысле интерпретации содержащегося в нем кода. То есть код все равно компилируется `rustc` и выполняется из двоичного файла.

К числу распространенных вариантов использования сценариев сборки можно отнести следующие:

¹⁷ См. <https://crates.io/crates/rand>.

¹⁸ См. <https://rocket.rs/>.

¹⁹ См. <http://mng.bz/1JRj>.

- компиляция кода C или C++;
- запуск пользовательских препроцессоров в коде Rust перед его компиляцией;
- генерация кода Rust protobuf с помощью protoc-rust²⁰;
- генерация кода Rust из шаблонов;
- запуск проверок платформы — таких как проверка ее наличия и поиск библиотек.

Cargo обычно перезапускает сценарий при каждом запуске сборки, но это поведение можно изменить с помощью `cargo:rerun-if-changed`.

Давайте рассмотрим простой пример "Hello, world!" с использованием небольшой библиотеки C. Сначала создадим новое приложение Rust и перейдем в его каталог:

```
$ cargo new build-script-example
$ cd build-script-example
```

Затем создадим небольшую библиотеку C с функцией, возвращающей строку "Hello, world!", для чего создадим файл с именем `src/hello_world.c`:

```
const char *hello_world(void) {
    return "Hello, world!";
}
```

Теперь обновим `Cargo.toml`, включив в него крейт cc в виде зависимости сборки и крейт libc для типов C:

```
[dependencies]
libc = "0.2"
[build-dependencies]
cc = "1.0"
```

После этого приступим к самому сценарию сборки, создав файл `build.rs` в каталоге верхнего уровня (но не внутри `src/`, где находятся другие исходные файлы):

```
fn main() {
    println!("cargo:rerun-if-changed=src/hello_world.c");           (1)
    cc::Build::new()                                                 (2)
        .file("src/hello_world.c")
        .compile("hello_world");
}
```

Здесь:

- в поз. (1) — предписание Cargo перезапускать сценарий сборки только при изменении файла `src/hello_world.c`;
- в поз. (2) — компиляция кода C в библиотеку с использованием крейта cc.

И, наконец, обновим файл `src/main.rs` для вызова функции C из нашей крошечной библиотеки:

²⁰ См. <https://crates.io/crates/protoc-rust>.

```
use libc::c_char;
use std::ffi::CStr;

extern "C" {
    fn hello_world() -> *const c_char; (1)
}

fn call_hello_world() -> &'static str { (2)
    unsafe {
        CStr::from_ptr(hello_world())
            .to_str()
            .expect("String conversion failure")
    }
}

fn main() {
    println!("{}", call_hello_world());
}
```

Здесь:

- в поз. (1) — определение внешнего интерфейса библиотеки С;
- в поз. (2) — оболочка внешней библиотеки, извлекающая статическую строку С.

И в завершение скомпилируем и запустим код:

```
$ cargo run
Compiling cc v1.0.67
Compiling libc v0.2.91
Compiling build-script-example v0.1.0 (/Users/brenden/dev/
code-like-a-pro-in-rust/code/c2/2.10/build-script-example)
    Finished dev [unoptimized + debuginfo] target(s) in 2.26s
        Running `target/debug/build-script-example`
Hello, world!
```

Полный код этого примера можно найти в каталоге `c02/build-script-example`.

2.12. Проекты Rust во встраиваемых средах

Будучи языком программирования системного уровня, Rust является отличным кандидатом для программирования встраиваемых систем. Польза от него особенно ярко проявляется в случаях, когда первостепенное значение имеет безопасность и явное выделение памяти. В этой книге программирование встраиваемых систем подробно не рассматривается, поскольку это тема, заслуживающая отдельной книги, но о нем всё же хотя бы вскользь стоит упомянуть, если вы предполагаете использовать Rust в качестве средства разработки проектов для встраиваемых систем.

Инструментарий статического анализа Rust особенно эффективен во встраиваемых доменах, где может быть сложнее отлаживать и проверять код во время выполне-

ния. Гарантии, выдаваемые в ходе компиляции, могут облегчить проверку состояний ресурсов, выбор пинов и безопасное выполнение конкурентных операций с совместно используемым состоянием.

При желании поэкспериментировать с программированием встроенных систем на языке Rust можно воспользоваться отличной поддержкой эмуляции устройств Cortex-M с использованием популярного проекта QEMU²¹. Соответствующий пример кода доступен на GitHub²².

На момент подготовки книги имеющиеся в Rust ресурсы программирования встроенных систем, отличных от ARM, не были широко представлены, но одним заметным исключением является платформа Arduino Uno. Крейт `ruduino`²³ содержит многократно используемые компоненты специально для системы Arduino Uno, являющейся доступной, маломощной, встроенной платформой, которую можно приобрести по цене ужина на двоих. Более подробную информацию о платформе Arduino можно найти на ее сайте²⁴.

Компилятор Rust (`rustc`) основан на проекте LLVM²⁵, поэтому техническая поддержка в нем существует для любой платформы, для которой LLVM имеет соответствующий бэкенд, хотя периферийные устройства при этом могут не работать. Например, даже при наличии ранней поддержки RISC-V, обеспечиваемой LLVM, аппаратные возможности для RISC-V всё равно ограничены. Более подробные сведения о программировании встроенных систем на Rust можно найти в доступном онлайн-источнике «The Embedded Rust Book»²⁶.

Выделение памяти

Когда необходимости в динамическом выделении памяти нет, для предоставления структур данных с фиксированными размерами без динамического распределения памяти можно воспользоваться крейтом `heapless`. Если же потребуется динамическое выделение памяти, относительно легко можно создать свой собственный выделитель, реализовав типаж `GlobalAlloc`²⁷. Для ряда встраиваемых платформ, в частности для популярных процессоров Cortex-M, средство выделения кучи в виде крейта `alloc-cortex-m` уже существует.

²¹ См. <https://www.qemu.org>.

²² См. <https://github.com/rust-embedded/cortex-m-quickstart>.

²³ См. <https://crates.io/crates/ruduino>.

²⁴ См. <https://www.arduino.cc>.

²⁵ LLVM (ранее Low Level Virtual Machine) — проект программной инфраструктуры для создания компиляторов и сопутствующих им утилит (см. <https://llvm.org>).

²⁶ См. <https://docs.rust-embedded.org/book>.

²⁷ См. <http://mng.bz/PR7n>.

Резюме

- Cargo является основным инструментом, используемым для создания, управления и публикации проектов на языке Rust.
- Пакеты в Rust называются *крайтами*, которые могут быть опубликованы в виде библиотек или приложений в реестре <https://crates.io>.
- Cargo используется для установки крайтов из crates.io.
- Cargo может применяться для автоматизации этапов сборки, тестирования и публикации системы, предназначеннной для непрерывной интеграции и развертывания.
- Команда `cargo doc` способна с помощью `rustdoc` автоматически создавать документацию для Rust-проекта. Документация может быть отформатирована с использованием формата CommonMark (спецификация Markdown).
- Как и crates.io, <https://docs.rs> автоматически предоставляет бесплатный хостинг документации для крайтов с открытым исходным кодом, опубликованных на crates.io.
- Rust может генерировать пригодные для распространения двоичные файлы, включающие все зависимости, *за исключением* библиотеки C. В системах Linux для максимальной переносимости при распространении предварительно скомпилированных двоичных файлов следует вместо использования системной библиотеки C статически ссылаться на `musl`.
- Крайты можно организовывать в модули и рабочие пространства, что позволяет разделять код на части.

3

Инструменты Rust

В этой главе:

- введение в инструментарий языка Rust: rust-analyzer, rustfmt, Clippy и sccache;
- интеграция Rust-инструментария с Visual Studio Code;
- использование стабильных иочных наборов инструментов;
- изучение дополнительных инструментов, которые могут оказаться полезными.

Владение любым языком предполагает умение пользоваться его инструментарием, и здесь мы рассмотрим наиболее важные инструменты, способствующие эффективной работе с Rust.

В целях повышения производительности и сокращения объема трудоемкой работы, необходимой для создания высококачественных программных продуктов, Rust предлагает целый ряд разнообразных инструментов. Используемый в Rust компилятор `rustc` создан на основе LLVM, поэтому Rust наследует богатый арсенал инструментов этого проекта, и в частности LLVM-отладчик — LLDB. В дополнение к инструментам, пришедшем в Rust из других языков, Rust содержит также набор собственных инструментов, специфичных именно для него, к обсуждению которых мы сейчас и приступаем.

Главным образом в этой главе речь пойдет о таких инструментах, как `rust-analyzer`, `rustfmt`, `Clippy` и `sccache`, — с ними, вероятнее всего, вам придется встречаться при каждом сеансе работы с Rust. Кроме того, здесь приводится также ряд инструкций по работе с некоторыми другими, периодически используемыми инструментами: `cargo-update`, `cargo-expand`, `cargo-fuzz`, `cargo-watch` и `cargo-tree`.

3.1. Общий обзор инструментария Rust

В главе 2 основное внимание было уделено работе с Cargo — инструментом управления Rust-проектами. Кроме него существует еще целый ряд инструментов, которыми можно пользоваться при работе с Rust. В отличие от Cargo, эти инструменты считаются дополнительными и могут применяться по вашему собственному усмотрению. И всё же я считаю их весьма эффективными и пользуюсь ими практически в каждом из моих Rust-проектов. Некоторые из этих инструментов могут понадобиться и вам для разработки своих проектов, поэтому с ними полезно познакомиться.

Инструменты, рассматриваемые в этой главе, обычно задействуются с использованием текстового редактора или же запускаются из командной строки. Описание основных инструментов языка Rust приведено в табл. 3.1, а в табл. 3.2 содержится информация о ряде популярных редакторов и порядке их поддержки в Rust.

Таблица 3.1. Основной инструментарий языка Rust

Название	Описание
Cargo	Инструмент управления Rust-проектами, предназначенный для компиляции, тестирования и управления зависимостями (рассматривается в главе 2)
Rust-analyzer	Обеспечивает поддержку Rust для текстовых редакторов, реализующих протокол языкового сервера (Language Server Protocol, LSP)
Rustfmt	Инструмент категоричного задания стиля написания кода Rust, обеспечивающий автоматическое форматирование и проверку кода и способный интегрироваться в системы CI/CD
Clippy	Инструмент обеспечения качества кода Rust, выполняющий множество проверок (называемых линтами) и способный интегрироваться в системы CI/CD
Scccache	Универсальный инструмент кеширования компилятора для повышения скорости компиляции в крупных проектах

Таблица 3.2. Популярные редакторы для работы с Rust

Редактор	Расширение	Краткая справка	Ссылки
Emacs	Rust-analyzer	Поддержка Rust через LSP	http://mng.bz/Jd7V
Emacs	Rust-mode	Принадлежащие Emacs Rust-расширения	https://github.com/rust-lang/rust-mode
IntelliJ IDEA	IntelliJ Rust	Принадлежащая JetBrains интеграция для Rust	https://www.jetbrains.com/rust/
Sublime	Rust-analyzer	Поддержка Rust через LSP	https://github.com/sublimelsp/LSP-rust-analyzer
Sublime	Rust enhanced	Принадлежащий Sublime пакет для Rust	https://github.com/rust-lang/rust-enhanced
Vim	Rust-analyzer	Поддержка Rust через LSP	https://rust-analyzer.github.io/manual.html#vimneovim

Таблица 3.2 (окончание)

Редактор	Расширение	Краткая справка	Ссылки
Vim	Rust.vim	Принадлежащая Vim конфигурация для Rust	https://github.com/rust-lang/rust.vim
VS Code	Rust-analyzer	Поддержка Rust через LSP	https://rust-analyzer.github.io/manual.html#vs-code

3.2. IDE-интеграция Rust: инструмент rust-analyzer

Инструмент rust-analyzer — наиболее совершенный и полнофункциональный редактор для языка Rust, способный интегрироваться с любым редактором, реализующим LSP (Language Server Protocol, протокол языкового сервера)¹. В частности, им предоставляются следующие функции:

- автодополнение кода;
- вставка импорта;
- переход к определениям;
- переименование обозначений;
- генерация документации;
- рефакторинг (реструктуризация);
- магические завершения;
- вывод ошибок встроенного компилятора;
- встроенные подсказки по типам и параметрам;
- подсветка семантического синтаксиса;
- отображение встроенной справочной документации.

3.2.1. Установка rust-analyzer

При работе в среде VS Code (Visual Studio Code) rust-analyzer может быть установлен с помощью интерфейса командной строки (Command Line Interface, CLI):

```
$ code --install-extension rust-lang.rust-analyzer
```

После установки инструмента rust-analyzer среда VS Code при работе с кодом Rust будет выглядеть так, как показано на рис. 3.1. Обратите внимание на кнопки **Run** | **Debug** над строкой `fn main()` — они позволяют запускать код или вести его отладку по одному щелчку.

При использовании IntelliJ Rust устанавливать отдельное расширение для поддержки Rust не понадобится. Но следует всё же заметить, что в IntelliJ Rust часть кода

¹ См. <https://microsoft.github.io/language-server-protocol>.



Рис. 3.1. VS Code с запущенным rust-analyzer, показывающий аннотации предполагаемых типов

используется совместно с rust-analyzer — в частности, для поддержки этим плагином макросов².

3.2.2. Магические завершения

В rust-analyzer имеется функция постфиксного завершения текста, обеспечивающая быстрое выполнение самых распространенных задач, — например, таких, как отладочная печать или форматирование строк. Освоение *магического завершения* позволит сэкономить уйму времени, избавив вас от необходимости многократно набирать один и тот же текст. Кроме того, вместо заучивания синтаксиса вам понадобится всего лишь запомнить выражения завершения. Как только вы освоите синтаксис языка и научитесь применять приемы магического завершения, я рекомендую внедрить их в вашу постоянную практику.

Магические завершения похожи на *snippetы* (присущие VS Code и другим редакторам), но с рядом специфичных для Rust функций, превращающих их в некое по-

² См. <http://mng.bz/wjAP>.

добие «сниппетов ++». Магические завершения работают не только в VS Code, но и в любом редакторе, поддерживающем протокол языкового сервера.

Использовать магические завершения не сложнее ввода выражения и использования завершения из появившегося в редакторе выпадающего меню. Например, для создания в текущем исходном файле тестового модуля можно ввести `tmod` и выбрать самый первый предлагаемый результат завершения, в результате чего будет создан такой вот шаблон тестового модуля:

```
tmod ->
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_name() {
    }
}
```

Завершение `tmod` создает тестовый модуль с одной-единственной тестовой функцией, которая далее соответствующим образом может быть заполнена. Вдобавок к `tmod` в Rust имеется завершение `tfn`, создающее тестовую функцию.

Еще одно магическое завершение предназначено для вывода на экран строки. Строковая интерполяция в версиях, предшествующих Rust 1.58.0, не поддерживалась, поэтому для оказания разработчикам помощи в ситуации отсутствия интерполяции строк `rust-analyzer` предоставляет несколько завершений для вывода на экран, занесения в регистрационный журнал и форматирования строк.

ПРИМЕЧАНИЕ

Хотя в Rust 1.58.0 интерполяция строк уже была добавлена, этот раздел я оставил в книге, поскольку он позволяет продемонстрировать возможности `rust-analyzer`.

Наберите в своем редакторе следующий текст:

```
let bananas = 5.0;
let apes = 2.0;

"bananas={bananas} apes={apes} bananas_per_ape={bananas / apes}"
```

А теперь поместите курсор за кавычкой в конце строки и наберите команду `.print` — это превратит строку в вариант завершения `println` (рис. 3.2).

Если однократным нажатием клавиши `<Enter>` выбрать из появившегося выпадающего меню вариант `println`, `rust-analyzer` приведет ваш код к следующему виду:

```
let bananas = 5.0;
let apes = 2.0;

println!(
    "bananas={} apes={} bananas_per_ape={}",
```

```

bananas,
apes,
bananas / apes
)

```



Рис. 3.2. VS Code с rust-analyzer, демонстрирующий магическое завершение println

В табл. 3.3 показаны несколько важных и достойных внимания магических завершений. Список этот не является исчерпывающим, а полный список магических завершений и других функций rust-analyzer можно найти в руководстве по нему³.

Таблица 3.3. Памятка по магическим завершениям

Выражение	Результат	Описание
"str {arg}".format	format!("str {}", arg)	Форматирует строку с аргументами
"str {arg}".println	println!("str {}", arg)	Выводит строку с аргументами
".logL	log::level!("str {}", arg) где level — это что-либо из debug, trace, info, warn или error	Регистрирует строку с аргументами на указанном уровне

³ См. <https://rust-analyzer.github.io/manual.html>.

Таблица 3.3 (окончание)

Выражение	Результат	Описание
pr	eprintln!("arg = {:?}", arg)	Выполняет отладку снippetа print (вывод осуществляется в stderr)
ppd	eprintln!("arg = {:#?}", arg)	Выполняет отладку снippetа pretty-print (вывод осуществляется в stderr)
expr.ref	&expr	Задает выражение
expr.refm	&mut expr	Изменяет выражение
expr.if	if expr {}	Преобразует выражение в инструкцию if, что особенно важно с Option и Result.

3.3. Поддержка аккуратности кода: инструмент rustfmt

Результаты форматирования исходного кода могут вас расстроить, особенно если в разработке проекта участвует несколько специалистов. Для проектов с одним разработчиком эта проблема не актуальна, но вот если разработчиков несколько, могут возникнуть расхождения в стиле кодирования. Ответом Rust на проблемы стиля кодирования является инструмент Rustfmt, предоставляющий идиоматическое автоматизированное и категоричное инструментальное средство стилизации. Тем, кто знаком с работой с Goland, это средство покажется похожим на gofmt или на эквивалентные ему инструменты из других языков. Идея *категоричного форматирования* относительно нова и, по моему скромному мнению, является прекрасным дополнением к современным языкам программирования.

Пример результата, получаемого при запуске на выполнение команды:

```
cargo fmt -- --check -v
```

включающей режимы расширенного вывода и проверки, показан на рис. 3.3. Перехода ключа --check заставит команду вернуть ненулевое значение, если форматирование не будет соответствовать ожиданиям, что пригодится для проверки формата кода в системах непрерывной интеграции.

Невозможно подсчитать, сколько часов своей жизни я потерял, занимаясь форматированием кода. А ведь эту проблему можно решить мгновенно, воспользовавшись rustfmt и просто предписав системе придерживаться в каждом вносимом коде вполне определенного стиля. Вместо того чтобы публиковать и поддерживать длинные документы по стилю, вы можете использовать rustfmt и сэкономить всем кучу времени.

3.3.1. Установка rustfmt

Rustfmt устанавливается в виде компонента rustup:

```
$ rustup component add rustfmt
```

```
...
```

```

● ● ● ✎ 2 brenden@MacBook-Pro:~/dev/dryoc

→ dryoc git:(main) cargo fmt -- --check -v
Using rustfmt config file /Users/brenden/dev/dryoc/.rustfmt.toml for /Users/brenden/dev/dryoc/src/lib.rs
Formatting /Users/brenden/dev/dryoc/src/argon2.rs
Formatting /Users/brenden/dev/dryoc/src/auth.rs
Formatting /Users/brenden/dev/dryoc/src/blake2b/blake2b_simd.rs
Formatting /Users/brenden/dev/dryoc/src/blake2b/blake2b_soft.rs
Formatting /Users/brenden/dev/dryoc/src/blake2b/mod.rs
Formatting /Users/brenden/dev/dryoc/src/bytes_serde.rs
Formatting /Users/brenden/dev/dryoc/src/classic/crypto_auth.rs
Formatting /Users/brenden/dev/dryoc/src/classic/crypto_box.rs
Formatting /Users/brenden/dev/dryoc/src/classic/crypto_box_impl.rs
Formatting /Users/brenden/dev/dryoc/src/classic/crypto_core.rs
Formatting /Users/brenden/dev/dryoc/src/classic/crypto_generichash.rs
Formatting /Users/brenden/dev/dryoc/src/classic/crypto_hash.rs
Formatting /Users/brenden/dev/dryoc/src/classic/crypto_kdf.rs
Formatting /Users/brenden/dev/dryoc/src/classic/crypto_kx.rs
Formatting /Users/brenden/dev/dryoc/src/classic/crypto_onetimeauth.rs
Formatting /Users/brenden/dev/dryoc/src/classic/crypto_pwhash.rs
Formatting /Users/brenden/dev/dryoc/src/classic/crypto_secretbox.rs
Formatting /Users/brenden/dev/dryoc/src/classic/crypto_secretbox_impl.rs
Formatting /Users/brenden/dev/dryoc/src/classic/crypto_secretstream_xchacha20poly1305.rs
Formatting /Users/brenden/dev/dryoc/src/classic/crypto_shorthash.rs
Formatting /Users/brenden/dev/dryoc/src/classic/crypto_sign.rs
Formatting /Users/brenden/dev/dryoc/src/classic/crypto_sign_ed25519.rs
Formatting /Users/brenden/dev/dryoc/src/classic/generichash_blaKE2b.rs
Formatting /Users/brenden/dev/dryoc/src/constants.rs
Formatting /Users/brenden/dev/dryoc/src/dryocbox.rs
Formatting /Users/brenden/dev/dryoc/src/dryocsecretbox.rs
Formatting /Users/brenden/dev/dryoc/src/dryocstream.rs
Formatting /Users/brenden/dev/dryoc/src/error.rs
Formatting /Users/brenden/dev/dryoc/src/generichash.rs
Formatting /Users/brenden/dev/dryoc/src/kdf.rs
Formatting /Users/brenden/dev/dryoc/src/keypair.rs
Formatting /Users/brenden/dev/dryoc/src/kx.rs
Formatting /Users/brenden/dev/dryoc/src/lib.rs
Formatting /Users/brenden/dev/dryoc/src/onetimeauth.rs
Formatting /Users/brenden/dev/dryoc/src/poly1305/mod.rs
Formatting /Users/brenden/dev/dryoc/src/poly1305/poly1305_soft.rs
Formatting /Users/brenden/dev/dryoc/src/protected.rs
Formatting /Users/brenden/dev/dryoc/src/pwhash.rs
Formatting /Users/brenden/dev/dryoc/src/rng.rs
Formatting /Users/brenden/dev/dryoc/src/scalarmult_curve25519.rs
Formatting /Users/brenden/dev/dryoc/src/sha512.rs
Formatting /Users/brenden/dev/dryoc/src/sign.rs
Formatting /Users/brenden/dev/dryoc/src/siphash24.rs
Formatting /Users/brenden/dev/dryoc/src/types.rs
Formatting /Users/brenden/dev/dryoc/src/utils.rs
Spent 0.018 secs in the parsing phase, and 0.098 secs in the formatting phase
Using rustfmt config file /Users/brenden/dev/dryoc/.rustfmt.toml for /Users/brenden/dev/dryoc/tests/integration_tests.rs
Formatting /Users/brenden/dev/dryoc/tests/integration_tests.rs
Spent 0.001 secs in the parsing phase, and 0.008 secs in the formatting phase
→ dryoc git:(main) █

```

Рис. 3.3. Rustfmt в действии применительно к крейту dryoc

После установки этим средством можно воспользоваться, запустив Cargo:

```
$ cargo fmt
# Теперь задействованный вами код будет отформатирован с помощью rustfmt
```

3.3.2. Конфигурирование rustfmt

Хотя большинству разработчиков вполне подойдет и исходная конфигурация rustfmt, может всё же понадобиться немного ее перенастроить под личные предпочтения. Сделать это можно путем добавления в исходное дерево проекта файла конфигурации `.rustfmt.toml` (листинг 3.1).

Листинг 3.1. Пример конфигурации в файле `.rustfmt.toml`

```
format_code_in_doc_comments = true
group_imports = "StdExternalCrate"
```

```
imports_granularity = "Module"
unstable_features = true
version = "Two"
wrap_comments = true
```

В табл. 3.4 приведены некоторые варианты настройки rustfmt, иллюстрирующие ряд допустимых конфигураций.

Таблица 3.4. Некоторые варианты настройки rustfmt

Настройка	Исходное значение	Рекомендуемое значение	Описание
imports_granularity	Preserve	Module	Определяет степень детализации инструкций импорта
group_imports	Preserve	StdExternalGroup	Определяет порядок группировка импорта
unstable_features	false	true	Включает использование толькоочных функций (недоступных на стабильном канале)
wrap_comments	false	true	Автоматический пословный перенос комментариев в добавление к коду
format_code_in_doc_comments	false	true	Применяет rustfmt к примерам исходного кода в документации
version	One	Two	Выбирает используемую версию rustfmt. Часть функций rustfmt доступна только в версии 2

На момент подготовки книги часть значимых возможностей rustfmt относилась к ночных функциям. Свежий перечень доступных вариантов стилизации может быть найден на веб-сайте rustfmt⁴.

СОВЕТ

Прибывшим к нам из мира C или C++ и желающим применить здесь тот же шаблон форматирования, следует обязательно ознакомиться с инструментом clang-format, входящим в состав LLVM.

3.4. Повышение качества кода: инструмент Clippy

Clippy представляет собой инструментальное средство Rust, предназначенное для повышения качества кода и способное на момент подготовки книги реализовать свыше 450 соответствующих проверок. Если вам когда-либо докучал дотошный коллега, которому нравилось вмешиваться в ваш код со своим анализом и указы-

⁴ См. <https://rust-lang.github.io/rustfmt/>.

вать на несущественные улучшения синтаксиса, форматирования и на необходимость устранения каких-либо стилистических недостатков, то Clippy — как раз то, что вам нужно. Он способен проделать всё то же самое, что и ваш настойчивый коллега, но без малейшего сарказма и во многих случаях даже побуждая к внесению в код необходимых изменений.

Clippy зачастую способен найти в вашем коде реальные проблемные места. Но истинную пользу от Clippy принесет вам избавление от необходимости участвовать в разборках по поводу стилевых проблем кода, поскольку это средство обеспечивает соблюдение принятых в Rust идиоматического стиля и шаблонов. Clippy похож на рассмотренный в предыдущем разделе инструмент `rustfmt`, но в более совершенном исполнении.

3.4.1. Установка Clippy

Clippy распространяется как компонент `rustup`, и поэтому его установка осуществляется следующим образом:

```
$ rustup component add clippy  
...
```

После установки Clippy может быть запущен для любого Rust-проекта с помощью Cargo:

```
$ cargo clippy  
...
```

Запущенный Clippy выдаст на экран нечто похожее на вывод компилятора `rustc` (рис. 3.4).

3.4.2. Clippy-ленты

Учитывая общее число проверок качества кода (называемых линтами), превышающее цифру 450, Clippy вполне заслуживает написания отдельной книги. Линты классифицируются по уровню серьезности (разрешить, предупредить, запретить и признать устаревшим) и группируются по типу, который может быть одним из следующих: корректность, ограничение, стиль, устаревание, педантичность, сложность, производительность, что-то, связанное с Cargo, и первичная проверка. Примером может служить линт `blacklisted_name`, запрещающий использование таких имен переменных, как `foo`, `bar` или `quux`. Этот линт может быть настроен на включение пользовательского списка запрещенных имен переменных.

Вот еще один пример линта — `bool_comparison`, ведущего проверку на наличие ненужных сравнений между выражениями и булевыми значениями. Так, неприемлемым считается следующий код:

```
if function_returning_boolean() == true {}           ← это Clippy не понравится...
```

А вот следующий код будет вполне приемлемым:

```
if function_returning_boolean() {}                  ← B == true нет необходимости
```

```

● ● ● ℰ 2 brenden@MacBook-Pro:~/dev/dryoc
→ dryoc git:(main) ✘ cargo clippy
warning: unreachable statement
  → src/auth.rs:128:9
127 |     loop {}
128 |         ----- any code following this expression is unreachable
|         let mut output = Output::new_byte_array();
|         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ unreachable statement
|
| = note: `#[warn(unreachable_code)]` on by default

warning: unused variable: `key`
  → src/auth.rs:124:9
124 |     key: Key,
|     ^^^ help: if this is intentional, prefix it with an underscore: `_key`
|
| = note: `#[warn(unused_variables)]` on by default

warning: unused variable: `input`
  → src/auth.rs:125:9
125 |     input: &Input,
|     ^^^^^ help: if this is intentional, prefix it with an underscore: `_input`

warning: empty `loop {}` wastes CPU cycles
  → src/auth.rs:127:9
127 |     loop {}

|
| = help: you should either use `panic!()` or add `std::thread::sleep(..)` to the loop body
| = help: for further information visit https://rust-lang.github.io/rust-clippy/master/index.html#empty_loop
| = note: `#[warn(clippy::empty_loop)]` on by default

warning: `dryoc` (lib) generated 4 warnings
  Finished dev [unoptimized + debuginfo] target(s) in 0.02s
→ dryoc git:(main) ✘

```

Рис. 3.4. Clippy в действии применительно к коду крейта dryoc, куда преднамеренно добавлена ошибка

Большинство линтов имеют стилевую направленность, но они могут также помочь в обнаружении ошибок, влияющих на производительность. Например, линт redundant_clone способен обнаруживать места, где переменная клонируется безо всякой на то необходимости. Как правило, все это выглядит примерно так:

```
let my_string = String::new("my string");
println!("my_string='{}'", my_string.clone());
```

Здесь вызов clone() совершенно не нужен. И если запустить Clippy в отношении этого кода, будет получено следующее предупреждение:

```
$ cargo clippy
warning: redundant clone
--> src/main.rs:3:37
|
3 | println!("my_string='{}'", my_string.clone());
|          ^^^^^^^^^ help: remove this
|
| = note: `#[warn(clippy::redundant_clone)]` on by default
note: this value is dropped without further use
--> src/main.rs:3:28
|
```

```
3 | println!("my_string='{}'", my_string.clone());
|           =^^^^^^^
= help: for further information visit
https://rust-lang.github.io/rust-clippy/master/index.html#redundant_clone

warning: 1 warning emitted
```

Обновления Clippy происходят весьма часто, и актуальный перечень линтов для стабильной версии Rust можно найти в документации Clippy⁵.

3.4.3. Конфигурирование Clippy

Настроить конфигурацию Clippy можно либо путем добавления в исходное дерево проекта файла `.clippy.toml`, либо помещением атрибутов Clippy в ваши исходные файлы Rust. Чаще всего атрибуты потребуются для отключения линтов Clippy — случаев выдачи Clippy предупреждений будет довольно много, но код останется таким же, каким и был задуман.

В частности, некоторые предупреждения о сложности, выдаваемые Clippy, возможно, придется — в отсутствие лучшей альтернативы — перенастраивать или отключать. Например, предупреждение `too_many_arguments` будет выдаваться при наличии функции, число аргументов у которой окажется больше допустимых по умолчанию семи штук. Это число для той или иной функции можно увеличить или проверку на его превышение просто отключить:

```
#[allow(clippy::too_many_arguments)]
fn function(
    a: i32, b: i32, c: i32, d: i32, e: i32, f: i32, g: i32, h: i32
) {
    // ... ← здесь будет ваш код
}
```

Показанный в приведенном коде атрибут `allow()` относится только к Clippy и предписывает исключить применение линта `too_many_arguments` в отношении следующей строки кода.

Вместо этого можно было бы изменить пороговое значение для `too_many_arguments` применительно ко всему проекту, добавив в ваш файл `.clippy.toml` следующую строку:

```
too-many-arguments-threshold = 10 ← замена порога числа аргументов с 7 по умолчанию на 10
```

`.clippy.toml` является обычным TOML-файлом, в котором в соответствии с вашими предпочтениями должен содержаться список пар имя = значение. Каждый линт и соответствующие ему параметры конфигурации подробно описаны в документации Clippy⁶.

⁵ См. <http://mng.bz/qjAr>.

⁶ См. <https://rust-lang.github.io/rust-clippy/stable/index.html>.

3.4.4. Автоматическое применение предложений Clippy

В определенных случаях Clippy может автоматически исправить код. В частности, когда у Clippy возникает конкретное предложение по исправлению кода, он чаще всего способен также и автоматически внести соответствующие исправления. Для автоматического исправления кода Clippy нужно запустить с флагом `--fix`:

```
$ cargo clippy --fix -Z unstable-options  
...
```

Заметьте, что наряду с этим здесь указан параметр `-Z unstable-options`, поскольку на момент подготовки книги функциональность, вызываемая ключом `--fix`, была присуща только ночным выпускам.

3.4.5. Использование Clippy в CI/CD

Если у вас имеется какая-либо система CI/CD, то я рекомендую включить Clippy в качестве ее составной части. При этом Clippy будет запускаться вами уже после сборки, тестирования и форматирования. Кроме того, можно дать Clippy указание не выдавать предупреждения запускаться для всех функций, а также проверять тесты:

```
$ cargo clippy  
...  
$ cargo clippy -- -D warnings  
...  
$ cargo clippy --all-targets --all-features -- -D warnings  
...
```

Здесь:

- в поз. (1) — запуск Clippy с настройками по умолчанию;
- в поз. (2) — запуск Clippy с указанием не выдавать предупреждения (вместо их разрешения);
- в поз. (3) — запуск Clippy без выдачи предупреждений с включением всей функциональности крейтов, а также с проверкой тестов (по умолчанию Clippy игнорирует тесты).

В проектах с открытым исходным кодом включение Clippy в проверки CI/CD облегчит другим разработчикам внесение в ваш проект высококачественного кода, а также упростит доверенное сопровождение кода и принятие вносимых в него изменений (листинг 3.2).

Листинг 3.2. Небольшой пример использования Clippy с GitHub Actions

```
on: [push]
```

```
name: CI
```

```
jobs:  
  clippy:  
    name: Rust project  
    runs-on: ubuntu-latest  
    steps:  
      - uses: actions/checkout@v2  
      - name: Install Rust toolchain with Clippy  
        uses: actions-rs/toolchain@v1  
        with:  
          toolchain: stable  
          components: clippy  
      - name: Run Clippy  
        uses: actions-rs/cargo@v1  
        with:  
          command: clippy  
          args: --all-targets --all-features -- -D warnings
```

Полный пример использования Clippy и rustfmt вместе с системой CI/CD Actions от GitHub приведен в *главе 2* (см. листинг 2.5).

3.5. Сокращение времени компиляции: инструмент sccache

Инструментальное средство sccache представляет собой универсальный кеш-компилятор, которым можно воспользоваться при разработке Rust-проектов. Для крупных проектов время компиляции Rust-кода может существенно возрасти, а sccache позволяет сократить его за счет кеширования неизменившихся объектов, созданных компилятором. Проект sccache создан некоммерческой организацией Mozilla специально для содействия процессу компиляции кода Rust, но его универсальность позволяет пользоваться им с большинством компиляторов. На его разработку создателей sccache вдохновило наличие инструмента ccache, с которым, возможно, приходилось сталкиваться тем, кто знаком с программированием на C или C++.

Даже для небольшого проекта установка sccache и его локальное использование поможет сэкономить время на перекомпиляцию кода. К слову сказать, компиляция кода крейта dryoc из «чистого» проекта обычно занимает на моем компьютере 8,891 с. А на компиляцию в тех же условиях с применением sccache уходит 5,839 с. Получается, что на компиляцию кода сравнительно небольшого проекта без sccache тратится на 52% больше времени, чем с его применением! А для более крупных проектов время станет накапливаться и сможет вырасти в весьма существенную величину.

Следует заметить, что sccache помогает только в тех случаях, когда код уже был ранее скомпилирован. Компиляция свежих сборок ускоряться не будет.

3.5.1. Установка sccache

Программа sccache написана на Rust и может быть установлена с помощью Cargo:

```
$ cargo install sccache
```

После установки sccache включается для использования его в качестве оболочки для rustc с Cargo. В качестве переменной среды окружения Cargo принимает аргумент RUSTC_WRAPPER. Скомпилировать и собрать любой проект Rust с помощью sccache можно, экспортирував переменную среды оболочки следующим образом:

```
$ export RUSTC_WRAPPER=`which sccache`  
$ cargo build
```

Здесь команда `which sccache` возвращает путь к sccache, предполагая, что он доступен в \$PATH.

3.5.2. Конфигурирование sccache

Если ранее вам уже приходилось пользоваться ccache, то и с sccache разобраться будет нетрудно. В sccache есть несколько примечательных функций, отсутствующих в ccache, — и его можно напрямую использовать с рядом сетевых бэкендов хранения, поэтому оно идеально подходит для применения с системами CI/CD. Им поддерживается протокол S3, нейтральный к поставщику программных средств, несколько сервисов хранения от поставщиков, а также протоколы Redis и Memcached с открытым исходным кодом. По умолчанию sccache задействует до 10 Гбайт локального хранилища.

Настройку sccache можно выполнить с помощью платформенно-зависимых файлов конфигурации, но допускается также настроить его, просто указав переменные среды окружения. Так, чтобы настроить sccache на использование бэкенда Redis, для него можно задать адрес в виде переменной среды окружения:

```
$ export SCCACHE_REDIS=redis://10.10.10.10/sccache
```

Здесь предполагается, что экземпляр Redis запускается на используемом по умолчанию порту 10.10.10.10 с базой данных по имени sccache.

За более подробной информацией по конфигурированию и использованию sccache следует обратиться к его официальной документации⁷.

3.6. Интеграция Rust с IDE-средами, включая Visual Studio Code

Эта тема не будет здесь раскрываться подробно, но о некоторых возможностях работы с Rust в текстовых редакторах всё же стоит упомянуть. В настоящее время

⁷ См. <https://github.com/mozilla/sccache>.

мои предпочтения отданы VS Code, но такие инструменты Rust, как `rust-analyzer`, `Clippy`, `rustfmt` и другие, можно использовать с любым редактором.

В руководстве по `rust-analyzer` приведены инструкции по его установке для интеграции с редакторами Vim, Sublime, Eclipse, Emacs и др. `Rust-analyzer` должен работать с любыми редакторами, поддерживающими API Language Server, к числу которых, в дополнение к уже упомянутым здесь, можно отнести множество и других популярных редакторов.

Использование `rust-analyzer` применительно к VS Code не сложнее установки любого расширения:

1. Сначала в командной строке нужно убедиться в установке компонента `rust-src`, что можно сделать с помощью `rustup`:

```
$ rustup component add rust-src
```

2. Затем из командной строки следует установить расширение VS Code для `rust-analyzer`:

```
$ code --install-extension matklad.rust-analyzer
```

Использовать расширение в VS Code так же просто, как открыть в нем любой другой проект Rust. Среда автоматически распознает в каталоге вашего проекта файл `Cargo.toml` и загрузит проект.

СОВЕТ

Открыть VS Code с помощью командной строки можно непосредственно из любого каталога проекта, выполнив команду:

```
code .
```

где `.` — сокращение для текущего рабочего каталога.

3.7. Использование наборов инструментов: стабильные иочные версии

Начав использование Rust со стабильным набором инструментов, вы постепенно поймете, что у него есть такие функции, доступ к которым в стабильном канале просто невозможен, но что они вполне доступны в ночном канале. Эта проблемная особенность Rust часто вызывает недовольство разработчиков — поскольку целый ряд популярных крейтов имеет исключительноочные версии. То есть ими можно воспользоваться лишь при работе с ночным каналом.

По сути, мы имеем два варианта Rust: стабильный и очной. Все это может показаться ненужной путаницей или излишествами, но на практике оказывается не такой уж и плохой затеей. В большинстве случаев удобнее пользоваться стабильным Rust, но иногда возникает желание воспользоваться и очной его версией. Даже в опубликованных открытых крейтах нередко могут присутствовать те или иные версии под `nightly`-флагом.

И со временем возникает вопрос: почему бы для получения всех преимуществ Rust просто не пользоваться исключительно очными версиями? С практической точки

зрения это не такая уж плохая идея. Единственное предостережение касается тех случаев, когда возникает желание опубликовать крейты для применения другими потребителями, а ваши потенциальные клиенты при этом имеют право пользоваться только стабильной версией Rust. В этих ситуациях имеет смысл попробовать воспользоваться стабильной версией и поддерживать ее, помещая ночные функции под флагом особенности `nightly`.

3.7.1. Функции, доступные только в ночной версии

Если возникнет потребность в использовании исключительно ночной функции, надо указать `rustc`, какую именно функцию нужно использовать. Например, чтобы включить доступную на момент подготовки книги только в ночном Rust функцию `allocator_api`, следует явно объявить ее включение (листинг 3.3).

Листинг 3.3. Код для `lib.rs` из `dryoc`

```
#![cfg_attr(
    any(feature = "nightly", all(feature = "nightly", doc)),
    feature(allocator_api, doc_cfg)
)]
```

Здесь:

- в поз. (1) — `any()` возвращает `true` при вычислении в истину любого из предикатов, а `all()` возвращает `true`, если в истину вычисляются все предикаты. Атрибут `doc` устанавливается автоматически при каждом анализе кода с помощью `rustdoc`;
- в поз. (2) — если предикат вычисляется в истину, включаются функции `allocator_api` и `doc_cfg`.

В приведенный код включены две ночные функции: `allocator_api` и `doc_cfg`. Первая функция обеспечивает пользовательское выделение памяти в Rust, а вторая — включает атрибут компилятора `doc`, позволяющий настраивать `rustdoc` из кода.

СОВЕТ

Информацию по встроенным атрибутам Rust вы найдете в соответствующей документации⁸. Предикаты `any()` и `all()` характерны для `cfg` и `cfg_attr`, являющихся атрибутами условной компиляции⁹.

Обратите также внимание на то, что в коде листинга 3.3 используется флаг особенности `nightly`, означающий, что сборка крейта должна происходить с включенной `nightly`-функцией. Пока в Rust не существует способа определения, с каким именно каналом компилируется ваш код, указывать флаги особенностей надо обязательно.

⁸ См. <https://doc.rust-lang.org/reference/attributes.html>.

⁹ См. <http://mng.bz/7vRv>.

3.7.2. Использование nightly-функций в публикуемых крейтах

В крейте `dryoc` обращение к ночному каналу используется для предоставления функции защищенной памяти. Применительно к крейту `dryoc`, под *защищенной памятью* подразумевается функция, посредством которой в структурах данных, использующих пользовательский распределитель памяти (который на момент подготовки книги относился в Rust исключительно кочной версии API), реализуется блокировка и защита памяти. Селекция функций в крейте `dryoc` показана в листинге 3.4.

Листинг 3.4. Код из `src/lib.rs`

```
#![cfg(any(feature = "nightly", all(doc, not(doctest))))]
#![cfg_attr(all(feature = "nightly", doc), doc(cfg(feature = "nightly")))]
#[macro_use]
pub mod protected;
```

Поясню ряд моментов, реализуемых в этом коде. Во-первых, в нем можно заметить **ключевые слова** `doc` и `doctest`. Их наличие обусловлено желанием обеспечить включение защищенного модуля при сборке документации, если особенность `feature = "nightly"` в этом процессе не предусмотрена, но не при запуске `doctests` (т. е. при тестировании примеров кода в документации крейта). Первая строка означает следующее: включить следующий блок кода (который `pub mod protected`), только если включены особенность `feature = "nightly"` или атрибут `doc`, и не запускать `doctests`. Компоненты `doc` и `doctests` являются специальными атрибутами, которые включаются только при запуске `cargo doc` или `cargo test`.

Вторая строка включает присущий `rustdoc` атрибут, сообщающий `rustdoc` о необходимости пометить все содержимое модуля как `feature = "nightly"`. Иными словами, если заглянуть в документацию крейта `dryoc`¹⁰, можно увидеть примечание, в котором говорится следующее:

Доступно только в качестве особенности крейта в его ночной версии.

В GitHub можно найти более подробные сведения о функции в Rust API-распределителя¹¹ и о функции атрибута `doc`¹².

3.8. Дополнительные инструменты: `cargo-update`, `cargo-expand`, `cargo-fuzz`, `cargo-watch`, `cargo-tree`

Существует еще целый ряд инструментов Cargo, достойных упоминания, и краткие сведения о них будут представлены в следующих подразделах. Каждый из этих ин-

¹⁰ См. <http://mng.bz/mjAa>.

¹¹ См. <https://github.com/rust-lang/rust/issues/32838>.

¹² См. <https://github.com/rust-lang/rust/issues/43781>.

струментов дополняет уже рассмотренный инструментарий и может встречаться в этой книге и далее.

3.8.1. Поддержание пакетов в актуальном состоянии: инструмент cargo-update

Пакеты, установленные с Cargo, периодически могут нуждаться в обновлении, и способ их поддержки в актуальном состоянии предоставляется инструментом cargo-update. Его применение отличается от обновления зависимостей проекта, осуществляемого с помощью команды cargo update. Крейт cargo-update предназначен для управления собственными зависимостями Cargo в отрыве от проекта.

Для установки cargo-update выполните команду:

```
$ cargo install cargo-update
```

С помощью cargo-update в разрабатываемом проекте можно реализовать следующие действия:

```
$ cargo help install-update      ← вывод на экран страницы помощи
```

```
...
```

```
$ cargo install-update -a       ← обновление всех установленных пакетов
```

```
...
```

3.8.2. Отладка макросов: инструмент cargo-expand

Временами могут попадаться макросы, требующие отладки в других крейтах, или возникать потребность реализации собственного макроса. В rustc предусмотрен способ создания исходного кода, получающегося в результате применения макроса, а cargo-expand выступает в роли оболочки, в которую заключена эта функциональная возможность.

Для установки cargo-expand выполните команду:

```
$ cargo install cargo-expand
```

С помощью cargo-expand в разрабатываемом проекте можно реализовать следующие действия:

```
$ cargo help expand            ← вывод на экран страницы помощи
```

```
...
```

```
$ cargo expand outermod::innermod   ← демонстрация развернутой формы "outermod::innermod"
```

```
...
```

Для элементарного Rust-проекта "Hello, world!" вывод cargo-expand будет выглядеть следующим образом:

```
#[feature(prelude_import)]
#[prelude_import]
use std::prelude::rust_2018::*;
#[macro_use]
extern crate std;
```

```
fn main() {
    {
        ::std::io::_print(::core::fmt::Arguments::new_v1(
            &["Hello, world!\n"],
            &match () {
                () => [],
            },
        ));
    };
}
```

Инструмент cargo-expand можно запускать для всего проекта или же, как показано в приведенном примере, отфильтровать запуск этого средства по названию элемента. С cargo-expand стоит поэкспериментировать, чтобы увидеть, как выглядит другой код после раскрытия его макросов. Для любого не самого крупного проекта расширенный код может приобрести весьма солидный объем, поэтому я рекомендую сначала провести фильтрацию по конкретным функциям или модулям. Особую пользу от применения cargo-expand я заметил при использовании имеющих макросы библиотек, поскольку это средство помогало мне понять, что происходит в коде, созданном другими разработчиками.

3.8.3. Тестирование с применением libFuzzer

Нечеткое тестирование, или *Fuzz-тестирование*, является одной из стратегий поиска непредсказуемых ошибок, и поддержка нечеткости на основе инструмента libFuzzer LLVM¹³ обеспечивается средством cargo-fuzz.

Для установки cargo-fuzz выполните команду:

```
$ cargo install cargo-fuzz
```

С помощью cargo-fuzz в разрабатываемом проекте можно реализовать следующее действие:

```
$ cargo help fuzz           ← вывод на экран страницы помощи
```

```
...
```

Чтобы использовать cargo-fuzz, потребуется создание тестов, рассчитанных на применение API libFuzzer. Это можно сделать с помощью инструмента cargo-fuzz, запустив на выполнение команду cargo fuzz add, за которой следует указать название теста. Например, чтобы создать стандартный тест (с cargo-fuzz), выполните команды:

```
$ cargo fuzz new myfuzztest          (1)
```

```
$ cargo fuzz run myfuzztest         (2)
```

Здесь:

- в поз. (1) — создание нового fuzz-теста под названием myfuzztest;
- в поз. (2) — запуск только что созданного теста (на выполнение которого может уйти много времени).

¹³ См. <https://llvm.org/docs/LibFuzzer.html>.

Получившийся тест (созданный cargo-fuzz в файле fuzz/fuzz_targets/myfuzztest.rs) выглядит так:

```
#!/usr/bin/env rustc
use libfuzzer_sys::fuzz_target;

fuzz_target!(|data: &[u8]| {
    // ... ← сюда помещается код для нечеткого тестирования
});
```

Более подробно тестирование fuzz-методом рассматривается в главе 7. Если ваше знакомство с libFuzzer или в целом с тестированием fuzz-методом уже состоялось, то с самостоятельным освоением cargo-fuzz у вас не должно возникнуть никаких проблем.

3.8.4. Периодический запуск Cargo-команд: инструмент cargo-watch

Cargo-watch представляет собой инструмент, постоянно отслеживающий изменения в исходном дереве вашего проекта и выполняющий команду по факту их возникновения. Чаще всего cargo-watch используется для автоматического запуска тестов, для генерации документации с помощью rustdoc или просто для перекомпиляции проекта.

Для установки cargo-watch выполните команду:

```
$ cargo install cargo-watch
```

С помощью cargo-watch в разрабатываемом проекте можно реализовать следующие действия:

```
$ cargo help watch      ← вывод на экран страницы помощи
...
$ cargo watch            ← регулярный запуск cargo check
$ cargo watch -x doc     ← регулярная переработка документации при изменениях
```

3.8.5. Проверка зависимостей: инструмент cargo-tree

По мере усложнения проектов в их зависимостях можно будет просто запутаться, либо потому, что их слишком много, либо из-за возникающих конфликтов, либо из-за образования некой иной комбинации неблагоприятных факторов. Одним из инструментов, позволяющих разобраться в происхождении зависимостей, является cargo-tree.

Для установки cargo-tree выполните команду:

```
$ cargo install cargo-tree
```

С помощью cargo-tree в разрабатываемом проекте можно реализовать следующее действие:

```
$ cargo help tree           ← вывод на экран страницы помощи
```

...

Если, к примеру, запустить cargo-tree в отношении крейта dryoc, будет выведено дерево зависимостей, показанное в листинге 3.5.

Листинг 3.5. Фрагмент вывода cargo tree для крейта dryoc

```
$ cargo tree
dryoc v0.3.9 (/Users/brenden/dev/dryoc)
└── bitflags v1.2.1
└── chacha20 v0.6.0
    └── cipher v0.2.5
        ├── generic-array v0.14.4
        │   └── typenum v1.12.0
        └── [build-dependencies]
            └── version_check v0.9.2
└── rand_core v0.5.1
    └── getrandom v0.1.16
        ├── cfg-if v1.0.0
        └── libc v0.2.88
└── curve25519-dalek v3.0.2
    ├── byteorder v1.3.4
    ├── digest v0.9.0
    │   └── generic-array v0.14.4 (*)
    ├── rand_core v0.5.1 (*)
    ├── subtle v2.4.0
    └── zeroize v1.2.0
        └── zeroize_derive v1.0.1 (proc-macro)
            ├── proc-macro2 v1.0.26
            │   └── unicode-xid v0.2.1
            ├── quote v1.0.9
            │   └── proc-macro2 v1.0.26 (*)
            ├── syn v1.0.68
            │   ├── proc-macro2 v1.0.26 (*)
            │   ├── quote v1.0.9 (*)
            │   └── unicode-xid v0.2.1
            └── synstructure v0.12.4
                ├── proc-macro2 v1.0.26 (*)
                ├── quote v1.0.9 (*)
                ├── syn v1.0.68 (*)
                └── unicode-xid v0.2.1
... обрезка ...
```

Здесь можно увидеть иерархию обычных и предназначенных только для разработки зависимостей крейта. Пакеты, отмеченные (*), показаны с удаленными дубликатами.

Резюме

- ❑ Rust поддерживается многими популярными редакторами либо через протокол языкового сервера (LSP), либо через собственные расширения.
- ❑ Rust-analyzer — канонический IDE-инструмент для языка Rust, которым можно воспользоваться с любым редактором, поддерживающим LSP.
- ❑ Использование rustfmt и Clippy позволяет повысить производительность и поднять качество кода.
- ❑ Порой возникают насущные потребности в использовании в опубликованных крейтах только ночных функций. Тогда для поддержки пользователей стабильных версий эти функции следует помещать под флагом особенностей.
- ❑ cargo-update упрощает обновление пакетов Cargo.
- ❑ cargo-expand позволяет раскрывать макросы, чтобы увидеть код, получаемый в результате их применения.
- ❑ cargo-fuzz позволяет легко интегрироваться с libFuzzer для проведения нечеткого тестирования.
- ❑ cargo-watch автоматизирует периодический запуск Cargo-команд при изменении кода.
- ❑ cargo-tree позволяет визуализировать деревья зависимостей проекта.

Часть II

Основные данные

При создании программных средств много времени уходит на работу со структурами данных. Иногда приходится выстраивать собственные пользовательские структуры данных, но зачастую нами, как правило, используются встроенные структуры, предоставляемые каждым языком программирования. Rust содержит весьма богатый набор гибких структур данных, обеспечивающих неплохой баланс производительности, удобства, функциональности и возможности подстройки под конкретные нужды.

Прежде чем приступать к реализации собственных пользовательских структур данных, стоит потратить время на изучение всех тонкостей основных структур, включенных в стандартную библиотеку Rust. Это даст вам понимание того, что ими предоставляется вполне достаточный арсенал гибких возможностей, удовлетворяющих потребностям практически любого приложения. А когда встроенных структур Rust вам станет недостаточно, полученные базовые знания о существующих структурах данных вы сможете с немалой пользой применить при разработке своих собственных структур.

В *части II* подробно рассматриваются основные структуры данных Rust и приемы управления памятью, позволяющие разобраться с главными условиями, обеспечивающими создание высококачественного кода на языке Rust. Как только вы освоите эффективные приемы использования имеющихся в Rust структур данных и функций управления памятью, работа с Rust для вас существенно упростится.

Структуры данных

В этой главе:

- использование основных структур данных Rust: строк, векторов и отображений;
- описание типов, применяемых в Rust: примитивных типов, структур, перечислений и псевдонимов;
- эффективные приемы применения основных типов языка Rust;
- конвертация типов данных;
- демонстрация порядка отображения примитивных типов на внешние библиотеки.

Пока изучению собственно языка Rust в этой книге внимания практически не уделялось — предыдущие две главы были посвящены его инструментальным средствам. Разобравшись с ними, можно начать погружение в язык Rust и его возможности, которым будет уделено основное внимание в оставшейся части книги. И в этой главе мы рассмотрим самую важную после его базового синтаксиса область языка Rust — структуры данных.

Как и в работе с любым другим языком, при программировании на Rust вам много времени придется уделять его структурам данных. В Rust, как и в любом современном языке программирования, предлагается базовый набор функциональных возможностей, ожидаемых при работе со структурами данных, но при этом еще и обеспечивается исключительная безопасность и высокий уровень производительности. Сумев разобраться с основными типами данных, имеющихся в Rust, вы увидите, что вся остальная часть языка стала для вас предельно ясной, поскольку его паттерны часто повторяются.

В этой главе обсуждаются отличия подхода к данным, выбранного в Rust, от подходов к ним в других языках, предоставлены сведения по основным типам данных и структурам и приведены приемы их эффективного использования. Мы также определимся здесь с порядком сопоставления примитивных типов Rust с типами

языка C, что позволит вам обеспечить интеграцию своих разработок с программными средствами, созданными не на Rust.

При работе с Rust большая часть времени, вероятнее всего, будет тратиться вами на три основные структуры данных: строки, векторы и отображения. Их реализации, включенные в стандартную библиотеку Rust, отличаются быстродействием и полноценными функциональными возможностями и способны покрыть потребности большинства типичных сценариев программирования.

И начнем мы с рассмотрения строк, которые обычно используются для представления множества источников и приемников данных.

4.1. Строковые типы *String*, *str*, *&str* и *&'static str*

Поначалу от количества и особенностей строковых типов, имеющихся в Rust, у меня кружилась голова. Вам же из-за всего этого можно не беспокоиться, поскольку у меня есть для вас хорошие новости: пусть концепции заимствований, времени жизни и управления памятью, имеющиеся в Rust, и навевают на строковые типы некую таинственность, смею вас заверить, что стоит только разобраться с базовой структурой памяти, и вы поймете, что в них нет ничего сложного.

Иногда вам может понадобиться тип *String*, а у вас будет *str*, или имеется *String*, но функции нужен *&str*. Перейти от одного к другому несложно, но как это сделать, поначалу может быть непонятно. С этим, как и со многим другим, в этом разделе мы и разберемся.

Важно отличать основные данные (непрерывную последовательность символов) от интерфейса, используемого для работы с ними. В Rust имеется только один вид строки, но есть несколько способов обработки выделения строки и ссылок на эту строку.

4.1.1. Сравнение *String* и *str*

Для начала проясним некоторые моменты — в Rust действительно есть два отдельных основных типа строк: *String* и *str*. И хотя технически это разные типы, по большому счету они представляют собой одно и то же — оба они являются последовательностями символов формата UTF-8 произвольной длины, хранящимися в непрерывной области памяти. Единственное практическое различие между *String* и *str* заключается в способе управления памятью. Поэтому, разбираясь во *всех* основных типах данных Rust, полезно осмысливать их с точки зрения управления памятью. Итак, имеющиеся в Rust два типа строк можно кратко охарактеризовать следующим образом:

- *str* — строка в формате UTF-8, размещенная в стеке, подлежащая заимствованию, но не перемещению или изменению (следует заметить, что *&str* может указывать на данные, размещенные в куче, — более подробный разговор об этом состоится далее);
- *String* — строка в формате UTF-8, размещенная в куче, подлежащая заимствованию и изменению.

Разница между данными, размещенными в куче и в стеке, в таких языках, как C и C++, может быть размытой, поскольку указатели C не сообщают о том, как именно была выделена память. В лучшем случае известно лишь то, что есть подходящая область памяти определенного типа, которая может иметь длину от 0 до N элементов. В Rust выделение памяти осуществляется явным образом — т. е. ваши типы данных в дополнение к количеству элементов обычно сами определяют, как именно выделяется для них память.

В C вы можете выделять строки в стеке и изменять их, но в Rust без использования ключевого слова `unsafe` это не разрешено, — сразу становится понятно, что в этом и кроется основной источник ошибок программирования на языке C.

Проиллюстрируем это парой строк на C:

```
char *stack_string = "stack-allocated string";
char *heap_string = strdup("heap-allocated string");
```

Здесь показаны два одинаковых типа указателей на разные виды памяти. Первый (`stack_string`) — это указатель на память, выделенную в стеке. Обычно выделение такой памяти обрабатывается компилятором и происходит по сути мгновенно. А `heap_string` представляет собой указатель того же типа на строку, выделенную в куче, — `strdup()` является стандартной функцией библиотеки C, выделяющей область памяти в куче с помощью функции `malloc()`, копирующей ввод в эту область и возвращающей адрес новой выделенной области.

ПРИМЕЧАНИЕ

Проявив особую педантичность, можно заметить, что для строки из приведенного примера с выделенной памятью в куче изначально была выделена память в стеке, но затем, после вызова `strdup()`, она была преобразована в строку с выделением ей памяти в куче. Доказать это можно, исследовав двоичный файл, сгенерированный компилятором, который будет содержать в двоичном коде строку, для которой была выделена память в куче.

В языке C все строки одинаковы — это просто непрерывные области памяти произвольной длины, завершающиеся нулевым символом (шестнадцатеричным байтовым значением `0x00`). А если вернуться к Rust, можно составить представление о `str` как об эквиваленте первой строки — `stack_string`. А `String` тогда будет эквивалентом второй строки — `heap_string`. Хотя это и несколько упрощенное пояснение, но всё же неплохой пример, помогающий понять суть строк в Rust.

4.1.2. Эффективное применение строк

При программировании на языке Rust работать в основном придется либо со `String`, либо со `&str`, но никак не со `str`. Неизменяемая строковая функциональность стандартной библиотеки Rust реализована для типа `&str`, а изменяемая — только для типа `String`.

Создать напрямую тип `str` невозможно — можно только заимствовать ссылку на него. А тип `&str` служит удобным минимально необходимым элементом — напри-

мер, при использовании в качестве аргумента функции, поскольку строку всегда можно заимствовать в виде `&str`.

Давайте кратко рассмотрим статические времена жизни: в Rust `'static` является специальным спецификатором времени жизни, определяющим ссылку (или заимствованную переменную), действительную в течение всего срока жизни процесса. Имеется ряд особых случаев, когда вам может понадобиться явное указание `&'static str`, но на практике это встречается довольно редко.

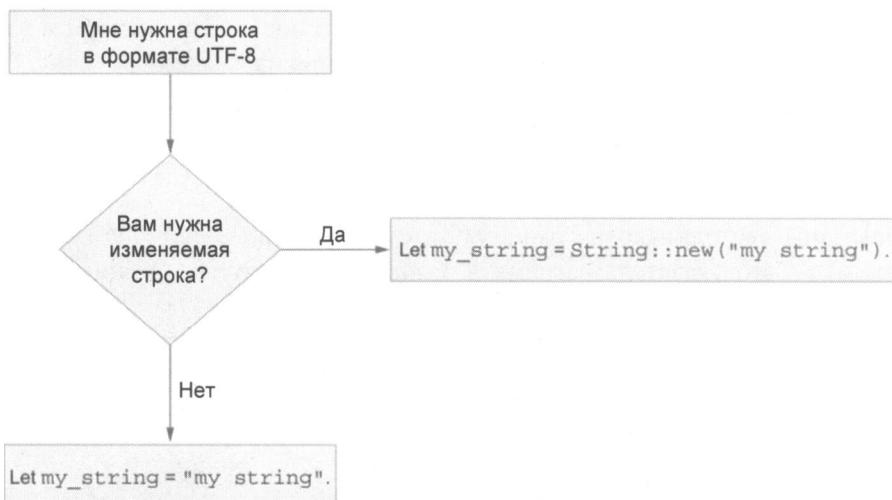


Рис. 4.1. Весьма простая блок-схема, позволяющая решить, что именно использовать: `str` или `String`

Решение о том, что использовать: `String` или статическую строку — сводится, как показано на рис. 4.1, к изменчивости. Если изменчивость не требуется, лучшим выбором практически всегда является статическая строка.

Единственное реальное различие между `&'static str` и `&str` заключается в том, что строка `String` может быть заимствована как `&str`, но не может заимствоваться как `&'static str`, поскольку жизнь `String` никогда не бывает такой же долгой, как и жизнь процесса. Когда `String` выходит из области видимости, то благодаря наличию типа-жа `Drop` (более подробно *типажи* рассматриваются в главе 8) происходит высвобождение выделенной под нее памяти.

«Под капотом» `String`, по сути, просто является типом `Vec`, составленным из символов в формате UTF-8 (более подробно тип `Vec` будет рассмотрен чуть позже). Кроме того, `str` — просто слайс из символов в формате UTF-8 (речь о *слайсах* пойдет в следующем разделе). Краткие сведения об основных типах строк, с которыми вам придется сталкиваться, и их отличиях друг от друга, представлены в табл. 4.1.

Еще одно различие между `str` и `String` заключается в том, что `String` можно перемещать, а `str` — нет. Фактически владеть переменной типа `str` невозможно, можно только удерживать ссылку на `str`. Рассмотрим в качестве иллюстрации код листинга 4.1.

Таблица 4.1. Строковые типы

Тип	Разновидность	Компоненты	Использование
Str	Размещаемый в стеке строковый слайс в формате UTF-8	Указатель на массив символов, дополненный его длиной	Неизменяемая строка, применяемая, например, в качестве инструкции регистрации или отладки или же в качестве чего-либо еще, где может быть неизменяемая строка, размещаемая в стеке
String	Размещаемая в куче строка в формате UTF-8	Вектор символов	Изменяемая строка с возможностью изменения размера и выделения и высвобождения памяти по мере необходимости
&str	Ссылка на неизменяемую строку	Указатель либо на заимствованное значение str, либо на String, дополненный длиной его значения	Может использоваться везде, где требуется неизменяемое заимствование либо str, либо String
&'static str	Ссылка на неизменяемую статическую строку	Указатель на объект str, дополненный его длиной	Ссылка на str с явно указанным временем жизни static

Листинг 4.1. Перемещаемые и неперемещаемые строки

```

fn print_String(s: String) {
    println!("print_String: {}", s);
}

fn print_str(s: &str) {
    println!("print_str: {}", s);
}

fn main() {
    // let s: str = "impossible str"; (1)
    print_String(String::from("String")); (2)
    print_str(&String::from("String")); (3)
    print_str("str"); (4)
    // print_String("str"); (5)
}

```

Здесь:

- в поз. (1) — не компилируется: rustc выдаст сообщение об ошибке «error[E0277]: the size for values of type str cannot be known at compilation time.» («ошибка[E0277]: размер значений типа str не может быть известен в ходе компиляции»);
- в поз. (2) — компилируется: перемещение строки из main в print_String;

- в поз. (3) — компилируется: возвращает `&str` из строки в `main`;
- в поз. (4) — компилируется: создает `str` в стеке в рамках функции `main` и передает ссылку на эту `str` как `&str` в `print_str`;
- в поз. (5) — не компилируется: `rustc` выдаст сообщение об ошибке «`error[E0308]: mismatched types, expected struct String, found &str.`» («ошибка[E0308]: неподходящие типы, ожидалась структура `String`, а обнаружена `&str`»).

При запуске приведенный здесь код выводит на экран следующие данные:

```
print_String: String
print_str: String
print_str: str
```

4.2. Что такое слайсы и массивы?

Слайсы и массивы относятся в Rust к специальным типам. Они представляют собой последовательности произвольных значений одного и того же типа. Допускается также наличие многомерных слайсов или массивов (т. е. слайсов слайсов, массивов массивов, массивов слайсов или слайсов массивов).

Слайсы — сравнительно новая концепция программирования, поскольку термин «слайс» при рассмотрении последовательностей в синтаксисе языков Java, C, C++, Python или Ruby обычно не встречается. Чаще всего последовательности в них называются *массивами* (как в Java, C, C++ и Ruby), *списками* (как в Python) или просто *последовательностями* (как в Scala). Эквивалентное поведение может быть представлено и в других языках программирования, но слайсы в них не обязаны входить в полноправные концепции языка или в типы, как в Rust или Go (хотя абстракция слайса все же завоевывает популярность в других языках). В C++ есть `std::span` и `std::string_view`, обеспечивающие эквивалентное поведение, но термин «слайс» при их описании в C++ не используется.

ПРИМЕЧАНИЕ

Понятие слайсов, исходя из публикации в блоге Роба Пайка (Rob Pike) от 2013 года¹, возникло, по-видимому, в языке Go.

Применительно к Rust слайсы и массивы немного отличаются друг от друга. Массив является последовательностью значений фиксированной длины, а слайс — последовательностью значений произвольной длины. То есть слайс может иметь переменную длину, определяемую во время выполнения, а массив имеет фиксированную длину, известную во время компиляции. Слайсы обладают еще одним интересным свойством, заключающимся в том, что их можно деструктурировать на сколько угодно неперекрывающихся подслайсов, и это может пригодиться при реализации алгоритмов, использующих рекурсивные стратегии или стратегии «разделяй и властвуй».

¹ См. <https://go.dev/blog/slices>.

Работа с массивами в Rust может порой складываться непросто, поскольку знание длины последовательности в процессе компиляции требует передачи информации компилятору в ходе ее проведения и присутствия этих данных в сигнатуре типа. Начиная с Rust версии 1.51, для определения массивов обобщенного типа произвольной длины можно, но только во время компиляции, воспользоваться функцией `const generics` (подробнее рассматриваемой в главе 10).

Давайте для понимания разницы между слайсами и массивами рассмотрим код листинга 4.2.

Листинг 4.2. Создание массива и слайса

```
let array = [0u8; 64];                                (1)  
let slice: &[u8] = &array;                            (2)
```

Здесь:

- в поз. (1) — сигнатура типа здесь: `[u8; 64]`, и это массив, инициализированный нулями;
- в поз. (2) — заимствование слайса, состоящего из массива.

В этом коде сначала (поз. (1)) происходит инициализация массива байтов, содержащего 64 элемента из одних нулей (`0u8` является сокращением для беззнакового целочисленного типа длиной 8 битов со значением 0). При этом 0 — значение, а `u8` — тип. Во второй строке (поз. (2)) массив заимствуется как слайс.

Пока здесь нет ничего интересного. Но со слайсами можно сделать что-нибудь более удивительное — например, провести заимствование дважды:

```
let (first_half, second_half) = slice.split_at(32);      (1)  
println!()  
    "first_half.len()={} second_half.len()={}",  
    first_half.len(),  
    second_half.len()  
);
```

- Здесь в поз. (1) производится разбиение и заимствование слайса дважды — с деструктурированием его на два отдельных, неперекрывающихся подслайса.

Приведенный в этом примере код вызывает функцию `split_at()`, являющуюся частью базовой библиотеки Rust и реализованную для всех слайсов, массивов и векторов, — она *деструктурирует* слайс (который уже заимствован из `array`) и дает нам два неперекрывающихся слайса, соответствующих первой и второй половине исходного массива.

Показанная здесь концепция деструктуризации играет в Rust немаловажную роль, поскольку вы можете столкнуться с ситуациями, при которых потребуется заимствовать часть массива или слайса. Фактически, используя этот шаблон, можно заимствовать один и тот же слайс или массив несколько раз, поскольку слайсы не перекрываются. Одним из распространенных вариантов использования этого приема является *парсинг*, или декодирование текста или двоичных данных. Например:

```
let wordlist = "one,two,three,four";
for word in wordlist.split(',') {
    println!("word={}", word);
}
```

Взглянув на этот код, можно сразу заметить, что имеющаяся в нем строка разбивается в тех местах, где есть символ запятой, а затем выводится каждое слово, на которые эта строка разбита. Вывод на экран при запуске указанного кода выглядит так:

```
word=one
word=two
word=three
word=four
```

В коде этого примера следует особо отметить отсутствие выделения памяти в куче. Вся память выделяется в стеке и имеет фиксированную длину, известную во время компиляции, без скрытых вызовов `malloc()`. Это аналогично работе с сырьими указателями в C, но здесь нет подсчета ссылок или сборки мусора, следовательно, нет никаких накладных расходов. И, в отличие от указателей C, код лаконичен, безопасен и не слишком многословен.

У слайсов, кроме того, предусмотрен целый ряд оптимизаций для работы с непрерывными областями памяти. Одна из таких оптимизаций представляет собой применение метода `copy_from_slice()`, который работает со слайсами. При вызове этого метода из стандартной библиотеки используется функция `memcopy()`, предназначенная, как показано в листинге 4.3, для копирования памяти.

Листинг 4.3. Фрагмент slice/mod.rs из <http://mng.bz/5oRO>

```
pub fn copy_from_slice(&mut self, src: &[T])
where
    T: Copy,
{
    ← код здесь опущен намеренно

    // БЕЗОПАСНОСТЬ: `self` допустимо для элементов `self.len()`
    // по определению, а `src` был проверен на ту же длину. Слайсы не могут
    // перекрываться, поскольку изменяемые ссылки являются исключительными.
    unsafe {
        ptr::copy_nonoverlapping(
            src.as_ptr(),
            self.as_mut_ptr(),
            self.len()
        );
    }
}
```

В приведенном коде, который взят из основной библиотеки Rust, `ptr::copy_nonoverlapping()` — просто обертка `memcpuy()` из библиотеки C. На некоторых платформах у `memcpuy()` имеются дополнительные оптимизации, выходящие за рамки того, что можно было бы сделать с помощью обычного кода. Другие оптимизированные функции — `fill()` и `fill_with()`, и в обоих для заполнения памяти используется функция `memset()`.

Итак, основные признаки массивов и слайсов:

- массив является последовательностью значений фиксированной длины, известной во время компиляции;
- слайсы являются указателями на смежные области памяти, включая длину, и представляют собой последовательности значений произвольной длины;
- и слайсы, и массивы можно рекурсивно деструктурировать в неперекрывающиеся подслайсы.

4.3. Векторы

Векторы, пожалуй, являются в Rust самым важным типом данных (следующим по важности является `String`, основанный на типе `Vec`). Работа с данными в Rust, когда нужна последовательность значений с изменяемым размером, зачастую связана с созданием векторов. Знатоки C++, вероятно, уже слышали термин «векторы», и векторный тип Rust во многих отношениях очень похож на тот, что имеется в C++. Векторы служат универсальным контейнером практически для любого вида последовательности.

Создание векторов является в Rust одним из способов выделения памяти в куче (другим способом является создание умных указателей, таких как `Box`, — эти указатели будут подробно рассмотрены в главе 5). У векторов имеется несколько внутренних оптимизаций, предназначенных для ограничения излишнего выделения памяти, в числе которых, например, поблочное выделение памяти. Кроме того, в ночном канале Rust для реализации задуманного разработчиком поведения при выделении памяти предусмотрена возможность предоставления пользовательского выделителя (подробнее рассматриваемого в главе 5).

4.3.1. Более глубокое погружение в `Vec`

Поскольку ссылку на слайс можно получить из вектора, `Vec` наследует методы слайсов. Исходя из объектно-ориентированного понимания, в Rust нет наследования, но `Vec` — особый тип, одновременно являющийся и `Vec`, и слайсом. Давайте в качестве примера рассмотрим реализацию `as_slice()`, имеющуюся в стандартной библиотеке (листинг 4.4).

Листинг 4.4. Фрагмент `vec/mod.rs` из <http://mng.bz/6nRe>

```
pub fn as_slice(&self) -> &[T] {  
    self  
}
```

Здесь выполняется специальное преобразование, которое (при обычных обстоятельствах) не сработало бы. В этом фрагменте берется тип `self`, бывший в предыдущем коде `Vec<T>`, и просто возвращается как `&[T]`. Попытка компиляции этого кода самого по себе обернется неудачей.

Так как же это работает? В Rust предоставляется *типаж* под названием `Deref` (и его изменяемый напарник `DerefMut`), который может использоваться компилятором для неявного приведения одного типа к другому. Будучи реализованным для того или иного типа, он автоматически придает этому типу реализацию всех методов разыменованного типа. В листинге 4.5 показано, что `Deref` и `DerefMut` применительно к `Vec` реализованы в стандартной библиотеке Rust.

Листинг 4.5. Фрагмент реализации `Deref` для `Vec` из <http://mng.bz/6nRe>

```
impl<T, A: Allocator> ops::Deref for Vec<T, A> {
    type Target = [T];

    fn deref(&self) -> &[T] {
        unsafe { slice::from_raw_parts(self.as_ptr(), self.len) }
    }
}

impl<T, A: Allocator> ops::DerefMut for Vec<T, A> {
    fn deref_mut(&mut self) -> &mut [T] {
        unsafe { slice::from_raw_parts_mut(self.as_mut_ptr(), self.len) }
    }
}
```

Здесь разыменование вектора приведет, исходя из его обычного указателя и длины, к его принудительному преобразованию в слайс. Следует заметить, что такая операция является промежуточной — т. е. слайс не может быть изменен, а длина предоставляемся слайсу во время разыменования.

Если по какой-либо причине взять слайс вектора и изменить размер этого вектора, то размер слайса не изменится. Однако такой прием возможен лишь в небезопасном коде, поскольку проверка заимствования не позволит заимствовать слайс из вектора и одновременно вносить в этот вектор изменения. Проиллюстрируем сказанное следующим кодом:

```
let mut vec = vec![1, 2, 3];
let slice = vec.as_slice();                                (1)
vec.resize(10, 0);                                         (2)
println!("{}", slice[0]);                                  (3)
```

Здесь:

- в поз. (1) — возвращение `&[i32]`, поскольку `vec` здесь заимствован;
- в поз. (2) — операция изменения;
- в поз. (3) — эта строка не пройдет компиляцию.

Этот код не скомпилируется, поскольку средство проверки заимствований вернет следующую ошибку:

```
error[E0502]: cannot borrow `vec` as mutable because it is also borrowed as immutable
--> src/main.rs:4:5
|
3 |     let slice = vec.as_slice();
|           --- immutable borrow occurs here
4 |     vec.resize(10, 0);
|           ^^^^^^^^^^^^^^^^^^ mutable borrow occurs here
5 |     println!("{}", slice[0]);
|           ----- immutable borrow later used here
```

4.3.2. Обертывание векторов

Некоторые типы в Rust, например `String`, просто обертывают `Vec`. Тип `String` — это `Vec<u8>`, который разыменовывается (используя ранее упомянутый типаж `Deref`) в `str` (листинг 4.6).

Листинг 4.6. Фрагмент `string.rs` из <http://mng.bz/orAZ>

```
pub struct String {
    vec: Vec<u8>,
}
```

Обертывание векторов является весьма распространенным паттерном, поскольку `Vec` — наиболее предпочтительный способ реализации изменяемой последовательности любого типа.

4.3.3. Типы, связанные с векторами

Применять `Vec` потребуется в 90% случаев. В остальных 10% случаев вам, скорее всего, понадобится `HashMap` (рассматриваемый в следующем разделе). Применение контейнерных типов, отличных от `Vec` или `HashMap`, может иметь смысл в определенных ситуациях или случаях, когда нужна особая оптимизация, но, как правило, вполне достаточно будет воспользоваться типом `Vec`, а применение другого типа не даст заметного улучшения производительности. Вспоминается следующая цитата:

Программисты тратят огромное количество времени на размышления или беспокойство по поводу скорости некритических частей своих программ, и эти попытки повышения эффективности на самом деле оказывают сильное негативное влияние при рассмотрении вопросов отладки и обслуживания. Мы должны забыть о малой эффективности, скажем, в 97% случаев: преждевременная оптимизация — корень всего зла. Тем не менее мы не должны упускать наши возможности в этих критических 3%.

Дональд Кнут (Donald Knuth)

В случаях обеспокоенности выделением слишком больших областей непрерывной памяти или тем, где именно расположена эта память, от этого беспокойства можно

легко избавиться, просто поместив Box в Vec (т. е. воспользовавшись `Vec<Box<T>>`). Надо также отметить, что в стандартной библиотеке Rust есть несколько других, по-рой востребованных типов коллекций, и некоторые из них обертывают `Vec` внутри себя:

- `VecDeque` — двусторонняя очередь, размер которой можно изменять на основе `Vec`;
- `LinkedList` — двусвязный список;
- `HashMap` — хеш-отображение, более подробно рассматриваемое в следующем разделе;
- `BTreeMap` — отображение, основанное на В-дереве;
- `HashSet` — хеш-набор, основанный на `HashMap`;
- `BTreeSet` — набор В-деревьев, основанный на `BtreeMap`;
- `BinaryHeap` — приоритизированная очередь, реализованная с помощью двоичной кучи и внутренне использующая `Vec`.

Дополнительные рекомендации, включая актуальные сведения о производительности основных структур данных, используемых в Rust, можно найти в справочнике по коллекциям стандартной библиотеки Rust².

Совет

В случае необходимости также имеет смысл выстраивать поверх `Vec` и собственные структуры данных. Чтобы понять, как это делается, можно обратиться к типу `BinaryHeap` из стандартной библиотеки Rust, полный пример создания которого задокументирован по адресу <http://mng.bz/n1A5>.

4.4. Отображения

`HashMap` — еще один контейнерный тип языка Rust, которым вы будете часто пользоваться. Если `Vec` считается в Rust наиболее предпочтительным типом с изменяемым размером, то `HashMap` — наиболее предпочтителен для тех случаев, когда нужна коллекция элементов, извлекаемых за постоянное время с помощью ключа. `HashMap` в Rust не сильно отличается от хеш-отображений, с которыми вам уже приходилось сталкиваться в других языках программирования, но реализация Rust благодаря ряду оптимизаций, предоставляемых этим языком, наверное, все же быстрее и безопаснее, чем то, что может быть найдено в других библиотеках.

Для хеширования в `HashMap` служит функция `Siphash-1-3`, которая также применяется в Python (начиная с версии 3.4), Ruby, Swift и Haskell. Эта функция обеспечивает для общих случаев вполне разумные компромиссы, но она может не подойти для использования очень маленьких или очень больших ключей — таких как целочисленные типы или очень большие строки.

Для использования с `HashMap` может быть предоставлена и собственная хеш-функция. Она может понадобиться при желании воспользоваться очень маленьки-

² См. <https://doc.rust-lang.org/std/collections/index.html>.

ми или очень большими ключами, но для большинства случаев вполне сгодится и исходная реализация этой функции.

4.4.1. Пользовательские функции хеширования

Чтобы воспользоваться `HashMap` с пользовательской функцией хеширования, сначала нужно найти уже существующую реализацию или написать хеш-функцию, реализующую необходимые типажи. `HashMap` требует, чтобы для желаемой хеш-функции были реализованы типажи `std::hash::BuildHasher`, `std::hash::Hasher` и `std::default::Default` (типажи более подробно рассматриваются в главе 8).

Давайте рассмотрим показанную в листинге 4.7 реализацию `HashMap` из стандартной библиотеки.

Листинг 4.7. Фрагмент `HashMap` из <http://mng.bz/vPAp>

```
impl<K, V, S> HashMap<K, V, S>
where
    K: Eq + Hash,
    S: BuildHasher,
{
    ← код опущен намеренно
}
```

Здесь вы можете увидеть `BuildHasher`, указанный как требуемый типаж для параметра типа `S`. Если копнуть чуть глубже, то из кода листинга 4.8 можно понять, что `BuildHasher` — это просто обертка для типажа `Hasher`.

Листинг 4.8. Фрагмент `BuildHasher` из <http://mng.bz/46RR>

```
pub trait BuildHasher {
    /// Type of the hasher that will be created.
    type Hasher: Hasher;   <- Здесь требуется типаж Hasher
    ← код опущен намеренно
}
```

API-интерфейсы `BuildHasher` и `Hasher` оставляют основную часть подробностей реализации на усмотрение автора хеш-функции. Для `BuildHasher` нужен только метод `build_hasher()`, возвращающий новый экземпляр `Hasher`. Для типажа `Hasher` требуются два метода: `write()` и `finish()`. Метод `write()` принимает байтовый слайс (`&[u8]`), а метод `finish()` возвращает беззнаковое 64-разрядное целое число, представляющее собой вычисленный хеш. Типаж `Hasher` также предоставляет ряд общих реализаций, которые при его использовании наследуются абсолютно бесплатно. Для получения более четкого представления о порядке их работы стоит изучить документацию по самим этим типажам³.

³ См. <http://mng.bz/QR76> и <http://mng.bz/Xqo9>.

СОВЕТ

Большое число доступных крейтов, уже реализующих широкий спектр хеш-функций, можно найти по адресу <https://crates.io>.

Давайте в качестве примера создадим, как показано в листинге 4.9, `HashMap` с крейтом `MetroHash`, разработанным Дж. Эндрю Роджерсом (J. Andrew Rogers), описание которого можно найти на его сайте⁴ и который в рассматриваемом случае послужит альтернативой для `SipHash`. Все существенно упрощается тем обстоятельством, что крейт `MetroHash` уже включает в себя необходимую реализацию типажей `std::hash::BuildHasher` и `std::hash::Hasher`.

Листинг 4.9. Листинг кода для использования `HashMap` с `MetroHash`

```
use metrohash::MetroBuildHasher;
use std::collections::HashMap;

let mut map = HashMap::<String, String, MetroBuildHasher>::default();      (1)
map.insert("hello?".into(), "Hello!".into());                                (2)

println!("{:?}", map.get("hello?"));                                         (3)
```

Здесь:

- в поз. (1) — создание нового экземпляра `HashMap` с использованием `MetroHash`;
- в поз. (2) — вставка в отображение пары ключ-значение с использованием типажа `Into` для преобразования `&str` в `String`;
- в поз. (3) — извлечение из отображения возвращаемого как `Option` значения аргумента `{:?}` макроса `println!` и форматирование этого значения с помощью типажа `fmt::Debug`.

4.4.2. Создание хешируемых типов

`HashMap` можно использовать с произвольными ключами и значениями, но в ключах должны реализовываться типажи `std::cmp::Eq` и `std::hash::Hash`. Многие типажи, в том числе и такие, как `Eq` и `Hash`, могут быть автоматически получены с помощью атрибута `#[derive]`. Рассмотрим следующий пример (листинг 4.10).

Листинг 4.10. Код для составного типа ключа

```
#[derive(Hash, Eq, PartialEq, Debug)]
struct CompoundKey {
    name: String,
    value: i32,
}
```

⁴ См. <https://www.jandrewrogers.com/2015/05/27/metrohash/>.

Приведенный здесь код представляет собой составной ключ, состоящий из имени и значения. Для получения четырех типажей: `Hash`, `Eq`, `PartialEq` и `Debug` — используется атрибут `#[derive]`. Хотя для `HashMap` требуются только `Hash` и `Eq`, нам также нужно получить `PartialEq`, поскольку `Eq` зависит от `PartialEq`. Мной был добавлен еще и `Debug`, предоставляющий автоматические методы отладки вывода данных, — это очень удобно для отладки и тестирования кода.

Пока до рассмотрения атрибута `#[derive]` дело в книге еще не дошло, но он будет использоваться в Rust довольно часто. Более подробно типажи и атрибут `#[derive]` мы рассмотрим в главах 8 и 9. Сейчас же этот атрибут следует считать автоматическим способом генерации реализаций типажей. Такие реализации типажей имеют дополнительное преимущество в том, что они пригодны для компоновки, — пока они существуют для любого подмножества типов, они также могут быть выведены и для надмножества типов.

4.5. Типы Rust: примитивы, структуры, перечисления и псевдонимы

Будучи строго типизированным языком, Rust предоставляет несколько способов моделирования данных. В нижней части иерархии собраны примитивные типы, обращающиеся с самыми основными единицами данных: числовыми значениями, байтами и символами. Выше их расположены структуры и перечисления, используемые для инкапсуляции других типов. На самом верху находятся псевдонимы, позволяющие переименовывать и объединять другие типы в новые типы.

Подытоживая сказанное, отметим, что в Rust можно выделить четыре категории типов:

- **примитивы**, куда входят строки, массивы, кортежи и целочисленные типы;
- **структуры**, относящиеся к составным типам, состоящим из любой произвольной комбинации других типов, — подобные, например, структурам языка C;
- **перечисления**, относящиеся в Rust к специальному типу, несколько похожему на перечисления в C, C++, Java и других языках;
- **псевдонимы** — синтаксическое средство для создания новых определений типов на основе уже существующих.

4.5.1. Применение примитивных типов

Примитивные типы предоставляются языком Rust и базовой библиотекой. Они эквивалентны примитивам, которые можно найти в любом другом строго типизированном языке, за некоторыми исключениями, которые мы и рассмотрим в этом разделе. Описание основных примитивных типов, включающее целые числа, числа с плавающей точкой, кортежи и массивы, приведено в табл. 4.2.

Таблица 4.2. Примитивные типы языка Rust

Класс	Вид	Описание
Скалярный	Целые числа	Может быть целым числом как со знаком, так и без знака, длиной от 8 до 128 битов (привязано к байту, т. е. 8 битов)
Скалярный	Размеры	Размерный тип, зависящий от архитектуры, который может быть знаковым или беззнаковым
Скалярный	С плавающей точкой	32- или 64-разрядные числа с плавающей точкой
Составной	Кортежи	Коллекция типов или значений фиксированной длины, которая может быть деструктурирована
Последовательность	Массивы	Последовательность значений типа фиксированной длины, который может разбиваться на слайсы

Целочисленные типы

Целочисленные типы можно распознать по их обозначению (либо `i`, либо `u` — для знаковых и беззнаковых величин соответственно), за которым следует количество битов. Размеры начинаются с обозначения `i` или `u`, за которым следует слово `size`. Типы с плавающей точкой начинаются с обозначения `f`, за которым следует количество битов. Описание примитивных целочисленных типов приведено в табл. 4.3.

Таблица 4.3. Идентификаторы целочисленных типов

Длина	Знаковый идентификатор	Беззнаковый идентификатор	C-эквивалент
8 битов	<code>i8</code>	<code>u8</code>	<code>char</code> и <code>uchar</code>
16 битов	<code>i16</code>	<code>u16</code>	<code>short</code> и <code>unsigned short</code>
32 бита	<code>i32</code>	<code>u32</code>	<code>int</code> и <code>unsigned int</code>
64 бита	<code>i64</code>	<code>u64</code>	<code>long</code> , <code>long long</code> , <code>unsigned long</code> и <code>unsigned long long</code> в зависимости от платформы
128 битов	<code>i128</code>	<code>u128</code>	Расширенные целые числа не являются стандартными для C, но предоставляются как <code>_int128</code> или <code>_uint128</code> с GCC и Clang

Тип для целочисленного литерала можно указать, добавив идентификатор типа. Например, `0u8` обозначает беззнаковое 8-битное целое число со значением 0. Целочисленные значения могут иметь префикс `0b`, `0o`, `0x` или `b` для двоичных, восьмеричных, шестнадцатеричных и байтовых литералов соответственно. Рассмотрим код листинга 4.11, выводящий на экран каждое значение как десятичное (по основанию 10) целое число.

Листинг 4.11. Листинг кода с целочисленными литералами

```
let value = 0u8;
println!("value={}, length={}", value, std::mem::size_of_val(&value));
```

```
let value = 0b1u16;
println!("value={}, length={}", value, std::mem::size_of_val(&value));
let value = 0o2u32;
println!("value={}, length={}", value, std::mem::size_of_val(&value));
let value = 0x3u64;
println!("value={}, length={}", value, std::mem::size_of_val(&value));
let value = 4u128;
println!("value={}, length={}", value, std::mem::size_of_val(&value));

println!("Binary (base 2)      0b1111_1111={}", 0b1111_1111);
println!("Octal (base 8)       0o1111_1111={}", 0o1111_1111);
println!("Decimal (base 10)    1111_1111={}", 1111_1111);
println!("Hexadecimal (base 16) 0x1111_1111={}", 0x1111_1111);
println!("Byte literal          b'A'={}", b'A');
```

При запуске этого кода мы получим следующий результат (листинг 4.12).

Листинг 4.12. Вывод на экран при запуске кода из листинга 4.11

```
value=0, length=1
value=1, length=2
value=2, length=4
value=3, length=8
value=4, length=16
Binary (base 2)      0b1111_1111=255
Octal (base 8)       0o1111_1111=2396745
Decimal (base 10)    1111_1111=11111111
Hexadecimal (base 16) 0x1111_1111=286331153
Byte literal          b'A'=65
```

Размерные типы

Для размерных типов идентификаторами являются `usize` и `isize`. Сами размеры зависят от применяемой платформы и обычно имеют длину 32 или 64 бита для 32- и 64-битных систем соответственно. Идентификатор `usize` эквивалентен `size_t` в C, а `isize` предоставляется, чтобы позволить воспользоваться знаковой арифметикой с размерами. В стандартной библиотеке Rust функции, возвращающие или ожидающие параметр длины, ожидают идентификатор `usize`.

Арифметика на примитивных типах

Многие языки допускают непроверяемую арифметику на примитивных типах. В C и C++, в частности, ряд арифметических операций имеют неопределенные результаты и не приводят к ошибкам. Одним из примеров этого является деление на ноль. Рассмотрим следующую программу на C (листинг 4.13).

Листинг 4.13. Код divide_by_zero.c

```
#include <stdio.h>

int main() {
    printf("%d\n", 1 / 0);
}
```

Если этот код скомпилировать и запустить на выполнение:

```
clang divide_by_zero.c && ./a.out
```

он выведет значение, которое окажется случайным. И Clang, и GCC успешно компилируют этот код, и оба выведут предупреждение, но в них нет проверки времени выполнения на неопределенную операцию.

В Rust вся арифметика проверяется по умолчанию. Рассмотрим следующую программу на Rust:

```
// println!("{}", 1 / 0);           ← не проходит компиляцию

let one = 1;
let zero = 0;
// println!("{}", one / zero);     ← не проходит компиляцию

let one = 1;
let zero = one - 1;
// println!("{}", one / zero);     ← также не проходит компиляцию

let one = { || 1 }();
let zero = { || 0 }();
println!("{}", one / zero);       ← а здесь код паникует!
```

В приведенном коде компилятор Rust весьма успешно справляется с отловом ошибок в ходе компиляции. Но нам нужно обмануть компилятор, чтобы код мог скомпилироваться и запуститься. Здесь это делается путем инициализации переменной из значения, возвращаемого замыканием. Еще одним способом добиться такого же результата является создание обычной функции, возвращающей желаемое значение. В любом случае при запуске проблемного кода получается следующий вывод на экран:

```
Running `target/debug/unchecked-arithmetic`
thread 'main' panicked at 'attempt to divide by zero', src/main.rs:14:20
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
```

Если в Rust нужен более жесткий контроль над арифметикой, то для того, чтобы справиться с такими операциями, примитивными типами предоставляются несколько методов. Например, для безопасной обработки деления на ноль можно воспользоваться методом `tested_div()`, возвращающим `Option`:

```
assert_eq!((100i32).checked_div(1i32), Some(100i32));    ← 100 / 1 = 1
assert_eq!((100i32).checked_div(0i32), None);           ← 100 / 0 — неопределенный результат
```

Для скалярных типов (целочисленных, размерных и чисел с плавающей точкой) Rust предоставляет набор методов, выполняющих основные арифметические операции (например, деление, умножение, сложение и вычитание) в проверенных, непроверенных, переполненных и оберточных формах.

Если нужно добиться совместимости с поведением в таких языках, как C, C++, Java, C# и других, то метод, которым, вероятно, захочется воспользоваться, будет реализован в *оберточной форме*, выполняющей модульную арифметику и совместимой с эквивалентными операциями языка C. Не забывайте, что переполнение для целых чисел со знаком в C не определено. Пример модульной арифметики в Rust выглядит так:

```
assert_eq!(0xffff8.wrapping_add(1), 0);
assert_eq!(0xfffffffffu32.wrapping_add(1), 0);
assert_eq!(0u32.wrapping_sub(1), 0xffffffff);
assert_eq!(0x80000000u32.wrapping_mul(2), 0);
```

Полный список арифметических функций для каждого примитива доступен в документации по языку Rust⁵.

4.5.2. Использование кортежей

Кортежи Rust похожи на своих собратьев в других языках. Кортеж является последовательностью значений фиксированной длины, и у каждого значения может быть разный тип. Кортежи в Rust не являются рефлексивными, поэтому, в отличие от массивов, они не подлежат итерации. Невозможно также во время выполнения кода взять слайс кортежа или определить типы его компонентов. Кортежи по сути являются в Rust формой синтаксического средства, и при всей своей полезности они весьма ограничены в возможностях.

Рассмотрим следующий пример кортежа:

```
let tuple = (1, 2, 3);
```

Код на вид примерно такой же, как и тот, который можно было бы ожидать от массива, за исключением упомянутых ранее ограничений: кортежи нельзя разбивать на слайсы, проводить над ними операцию итерации или отражать. Чтобы получить доступ к отдельным элементам кортежа, на них можно ссылаться по их позиции, начиная с 0:

```
println!("tuple = ({}, {}, {})", tuple.0, tuple.1, tuple.2);    ← ВЫВОДИТ "tuple = (1, 2, 3)"
```

В качестве альтернативы можно воспользоваться управляющей конструкцией `match`, обеспечивающей временную деструктуризацию при условии соответствия образцу (сопоставление образцу более подробно рассматривается в главе 8):

⁵ Для i32 см. <https://doc.rust-lang.org/std/primitive.i32.html>.

```
match tuple {
    (one, two, three) => println!("{} {}, {}", one, two, three),      ← ВЫВОДИТ "1, 2, 3"
}
```

Мы также можем разбить кортеж на части с помощью следующего синтаксиса, перемещающего значения из кортежа:

```
let (one, two, three) = tuple;
println!("{} {}, {}", one, two, three);                                ← ВЫВОДИТ "1, 2, 3"
```

По моему опыту, наиболее распространенным вариантом использования кортежей является возврат нескольких значений из функции. Рассмотрим, к примеру, следующую лаконичную функцию `swap()`:

```
fn swap<A, B>(a: A, b: B) -> (B, A) {
    (b, a)
}

fn main() {
    let a = 1;
    let b = 2;

    println!("{:?}", swap(a, b));                                         ← ВЫВОДИТ "(2, 1)"
}
```

СОВЕТ

Создавать кортежи с более чем 12 аргументами не рекомендуется, хотя строго установленного верхнего предела длины кортежа нет. Однако в стандартной библиотеке предоставляются реализации типажей для кортежей с числом элементов, не превышающим 12.

4.5.3. Применение структур

Структуры в Rust являются основными строительными блоками. Они относятся к составным типам данных, которые могут содержать любые заданные типы и значения. По своей природе они похожи на структуры в объектно-ориентированных языках или классы языка С. Они также могут быть составлены в общем виде — по аналогии с шаблонами в C++ или с обобщениями в Java, C# или TypeScript (обобщения более подробно рассматриваются в главе 8).

Структуры применяются, когда нужно:

- предоставлять функции с отслеживанием состояния (т. е. функции или методы, работающие только с внутренним состоянием);
- управлять доступом к внутреннему состоянию (т. е. к закрытым переменным) или инкапсулировать состояние за API-интерфейсом.

Структуры применять не обязательно. API-интерфейсы по желанию можно создавать только с функциями, как это делается с API на языке С. Кроме того, структуры нужны лишь для определения реализаций, а не для указания интерфейсов. В этом и состоит отличие Rust от таких объектно-ориентированных языков, как C++, Java и C#.

Самой простой структурной формой является пустая структура:

```
struct EmptyStruct {}
```

```
struct AnotherEmptyStruct;           ← структура блока, заканчивающаяся точкой с запятой без фигурных скобок
```

Иногда вам могут попадаться пустые структуры (еще их называют *юнит-структурами*). Еще одной формой является структура кортежа, имеющая следующий вид:

```
struct TupleStruct(String);
```

```
let tuple_struct = TupleStruct("string value".into());          (1)
```

```
println!("{}", tuple_struct.0);                                (2)
```

Здесь:

- в поз. (1) — инициализация структуры, подобной кортежу;
- в поз. (2) — доступ к первому элементу кортежа можно получить с указанием .0, ко второму — .1, к третьему — .2 и т. д.

Структура кортежа является особой формой структуры с поведением, похожим на кортеж. Главное отличие структуры кортежа от обычной структуры заключается в том, что в структуре кортежа значения не имеют имен, а имеют только типы. Следует заметить, что структура кортежа несет в конце объявления точку с запятой (;), которая не требуется для обычных структур (за исключением пустого объявления). В конкретных случаях структуры кортежа могут быть удобны тем, что они позволяют опускать имена полей (тем самым экономится несколько символов в исходном коде), но при этом они также создают неоднозначность.

Обычная структура содержит список элементов с именами и типами, например:

```
struct TypicalStruct {
    name: String,
    value: String,
    number: i32,
}
```

Каждый элемент в структуре имеет по умолчанию *модульную видимость* — т. е. значения в структуре доступны в любом месте в пределах текущего модуля. Видимость может быть установлена для каждого элемента:

```
pub struct MixedVisibilityStruct {                               (1)
    pub name: String,                                         (2)
    pub(crate) value: String,                                 (3)
    pub(super) number: i32,                                  (4)
}
```

Здесь:

- в поз. (1) — открытая структура, видимая за пределами крейта;
- в поз. (2) — этот элемент является открытим и доступен за пределами крейта;
- в поз. (3) — этот элемент открыт в пределах крейта;

- в поз. (4) — этот элемент находится в пределах родительской области видимости.

В большинстве случаев делать элементы структуры открытыми вам не понадобится. Элемент внутри структуры может быть доступен и изменен любым кодом в пределах области видимости, открытой для этого элемента структуры. Видимость по умолчанию (эквивалентная получаемой по объявлению `pub(self)`) позволяет любому коду внутри этого же модуля получать доступ к элементам внутри структуры и к их изменению.

Семантика видимости применяется не только к составляющим структуры элементам, но и к самим структурам. Чтобы обеспечить видимость структуры за пределами крейта (т. е. чтобы она могла использоваться из библиотеки), ее нужно объявить с указанием `pub struct MyStruct { ... }`. Структура, не объявленная открытой явным образом, не будет доступна вне крейта (это положение также применимо к функциям, типажам и к любым другим объявлениям).

При объявлении структуры вам, вероятнее всего, понадобится получить какие-либо стандартные реализации типажей:

```
#[derive(Debug, Clone, Default)]
struct DebuggableStruct {
    string: String,
    number: i32,
}
```

В приведенном коде объявлено получение типажей `Debug`, `Clone` и `Default`. Эти типажи можно кратко охарактеризовать следующим образом:

- `Debug` — предоставляет метод `fmt()`, выполняющий форматирование (для вывода на экран) содержимого типа;
- `Clone` — предоставляет метод `clone()`, создающий копию (или клон) типа;
- `Default` — предоставляет реализацию метода `default()`, возвращающего (обычно пустой) исходный экземпляр типа.

При желании эти типажи можно получить и самостоятельно (например, если захотется настроить их поведение), но пока каждый типаж реализуется во всех элементах структуры, их можно получить автоматически, сэкономив при этом на вводе текста уйму времени.

Теперь с полученными в предыдущем примере тремя типажами можно сделать следующее:

```
let debuggable_struct = DebuggableStruct::default();
println!("{:?}", debuggable_struct);                                (1)
println!("{:?}", debuggable_struct.clone());                         (2)
```

Здесь:

- в поз. (1) — вывод на экран `DebuggableStruct{ string: "", number: 0 }`;
- в поз. (2) — также вывод на экран `DebuggableStruct{ string: "", number: 0 }`

Чтобы определить для структуры те или иные методы, их *реализацию* нужно предварить указанием ключевого слова `impl`:

```
impl DebuggableStruct {
    fn increment_number(&mut self) {           ← функция, принимающая изменяемую ссылку на self
        self.number += 1;
    }
}
```

В этом коде берется изменяемая ссылка на нашу структуру и увеличивает указываемое с ее помощью значение на 1. Это можно сделать и по-другому — воспользоваться структурой и вернуть ее из функции:

```
impl DebuggableStruct {
    fn incremented_number(mut self) -> Self {           ← функция, принимающая принадлежащий ей
        self.number += 1;                                изменяемый экземпляр self
        self
    }
}
```

При небольшом имеющемся различии эти две реализации функционально эквивалентны друг другу. Порой для поглощения методом могут понадобиться входные данные, но в большинстве случаев предпочтительнее будет воспользоваться первой версией (с использованием `&mut self`).

4.5.4. Применение перечислений

Перечисления можно считать специализированным типом структуры, содержащим перечисляемые взаимоисключающие *варианты*. Перечисление в конкретный момент времени может быть представлено *одним* из своих вариантов — т. е. в структуре присутствуют *все* элементы структуры-перечисления, а в перечислении присутствует лишь *один* из его вариантов. Содержимое перечисления не ограничивается только целочисленными типами и может быть любого типа. Типы могут быть именованными или безымянными.

Этим перечисления Rust сильно отличаются от перечислений в таких языках, как C, C++, Java или C#. В них перечисления фактически используются в качестве способа определения константных значений. Перечисления Rust могут эмулировать перечисления, встречающиеся в других языках, но концептуально они совершенно другие. Хотя в C++ имеются перечисления, Rust-перечисления больше похожи на C++-перечисления, а на `std::variant`.

Рассмотрим следующее перечисление:

```
#[derive(Debug)]
enum JapaneseDogBreeds {
    AkitaKen,
    HokkaidoInu,
    KaiKen,
    KishuInu,
```

```

ShibaInu,
ShikokuKen,
}

```

Для этого перечисления `JapaneseDogBreeds` является названием типа перечисления, и каждый из элементов в перечислении — это `unit`-подобный тип. Поскольку типы в перечислении вне его самого не существуют, они создаются *внутри* перечисления. Теперь можно запустить следующий код:

```

println!("({:?}", JapaneseDogBreeds::ShibaInu);           (1)
println!("({:?}", JapaneseDogBreeds::ShibaInu as u32);   (2)

```

Здесь:

- в поз. (1) — вывод на экран "ShibaInu";
- в поз. (2) — вывод на экран "4" — 32-разрядного беззнакового целочисленного представления значения перечисления.

Приведение типа `enum` к `u32` успешно работает, поскольку `enum`-типы перечисляются. А что если потребуется перейти от числа 4 к `enum`-значению? Автоматического преобразования для этого не существует, но его можно реализовать самостоятельно, воспользовавшись типажом `From`:

```

impl From<u32> for JapaneseDogBreeds {
    fn from(other: u32) -> Self {
        match other {
            other if JapaneseDogBreeds::AkitaKen as u32 == other => {
                JapaneseDogBreeds::AkitaKen
            }
            other if JapaneseDogBreeds::HokkaidoInu as u32 == other => {
                JapaneseDogBreeds::HokkaidoInu
            }
            other if JapaneseDogBreeds::KaiKen as u32 == other => {
                JapaneseDogBreeds::KaiKen
            }
            other if JapaneseDogBreeds::KishuInu as u32 == other => {
                JapaneseDogBreeds::KishuInu
            }
            other if JapaneseDogBreeds::ShibaInu as u32 == other => {
                JapaneseDogBreeds::ShibaInu
            }
            other if JapaneseDogBreeds::ShikokuKen as u32 == other => {
                JapaneseDogBreeds::ShikokuKen
            }
            _ => panic!("Unknown breed!"),
        }
    }
}

```

В этом коде для выполнения сравнения нужно `enum`-тип привести к `u32`, а затем при наличии совпадения вернуть `enum`-тип. В случае, если ни одно значение не совпа-

дает, вызывается `panic!()`, приводящий к сбою программы. В синтаксисе приведенного кода используется условная функция `match`, позволяющая выполнять сопоставления с помощью инструкции `if`.

В перечислении также можно указать типы вариантов перечисления. Этим можно воспользоваться, чтобы добиться поведения, похожего на то, что имеется у C-перечислений:

```
enum Numbers {
    One = 1,
    Two = 2,
    Three = 3,
}

fn main() {
    println!("one={}", Numbers::One as u32); (1)
}
```

- Здесь в поз. (1) показан вывод на экран "one=1". Стоит заметить, что без приведения `as` этот код не компилируется, поскольку в `One` не реализуется `std::fmt`.

Перечисления в качестве вариантов могут содержать кортежи, структуры и анонимные (т. е. безымянные) типы:

```
enum EnumTypes {
    NamedType, (1)
    String, (2)
    NamedString(String), (3)
    StructLike { name: String }, (4)
    TupleLike(String, i32), (5)
}
```

Здесь:

- в поз. (1) — именованный тип;
- в поз. (2) — безымянный тип `String`;
- в поз. (3) — именованный тип `String`, записанный как кортеж с одним элементом;
- в поз. (4) — тип, подобный структуре, с одним элементом, называемым `name`;
- в поз. (5) — тип, подобный кортежу, с двумя элементами.

Чтобы стало понятнее, *безымянным* (неименованным) вариантом перечисления является вариант, указанный как тип, а не с использованием имени. Именованный вариант перечисления эквивалентен созданию внутри перечисления нового типа, также соответствующего перечислимому целочисленному значению. Иными словами, если нужно эмулировать поведение перечислений из таких языков, как C, C++ или Java, будут использоваться именованные варианты, удобно эмулирующие поведение перечисления, приводя значение к целочисленному типу, хотя варианты перечисления также являются типами (т. е. не просто значениями).

Как правило, внутри перечисления принято не смешивать именованные и безымянные варианты, поскольку это может внести неразбериху.

4.5.5. Применение псевдонимов

Псевдонимы — специальный тип в Rust, позволяющий предоставить альтернативное и эквивалентное имя для любого другого типа. Они эквивалентны `typedef` в C и C++ или ключевому слову `using` в C++. Определение псевдонима не создает новый тип.

Псевдонимы имеют два наиболее распространенных применения:

- предоставление определений типов с назначенным псевдонимом для открытых типов — в целях повышения эргономики и удобства для пользователя библиотеки;
- предоставление иных, более кратких обозначений типов, соответствующих более сложным композициям типов.

К примеру, может понадобиться создать псевдоним типа для хеш-отображения, часто используемого в собственном крейте:

```
pub(crate) type MyMap = std::collections::HashMap<String, MyStruct>;
```

Теперь вместо ввода полного названия `std::collections::HashMap<String, MyStruct>` можно использовать `MyMap`.

Для библиотек обычным приемом является экспорт открытых псевдонимов типов с рациональными значениями по умолчанию для построения типов при использовании обобщений. Порой бывает сложно определить, какие типы требуются для того или иного интерфейса, и псевдонимы предоставляют авторам библиотек один из способов сообщить эту информацию.

В контейнере `dryoc` для удобства мною предоставляются несколько псевдонимов типов, а в API-интерфейсе активно используются обобщения. Один из подходящих примеров показан в листинге 4.14.

Листинг 4.14. Фрагмент для `kdf.rs` из <http://mng.bz/yZAp>

```
/// Stack-allocated key type alias for key derivation with [`Kdf`].  
pub type Key = StackByteArray<CRYPTO_KDF_KEYBYTES>;  
/// Stack-allocated context type alias for key derivation with [`Kdf`].  
pub type Context = StackByteArray<CRYPTO_KDF_CONTEXTBYTES>;
```

Здесь внутри модуля предоставляются псевдонимы типов `Key` и `Context`, поэтому пользователю этой библиотеки не нужно беспокоиться о деталях реализации.

4.6. Обработка ошибок с помощью *Result*

Rust предоставляет несколько функций, упрощающих обработку ошибок. Все они основаны на использовании перечисления *Result*, определение которого приводится в листинге 4.15.

Листинг 4.15. Фрагмент std::result::Result с <http://mng.bz/M97Q>

```
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Result представляет собой успешную (возвращающую результат) или же провальную (возвращающую ошибку) операцию, и в качестве типа возврата для многих функций Rust очень скоро станет вполне привычным явлением.

В собственном крейте вам, вероятнее всего, захочется создать собственный тип ошибки. Этот тип может быть либо перечислением, содержащим всевозможные виды ожидаемых ошибок, либо просто структурой с чем-либо полезным, — например, с сообщением об ошибке. Не имея привычки что-либо усложнять, лично я предпочитаю просто предоставлять полезное сообщение и двигаться дальше. Очень простая структура ошибки может иметь следующий вид:

```
#[derive(Debug)]
struct Error {
    message: String,
}
```

Создавая крейт, нужно решить, какие типы ошибок будут возвращаться вашими функциями. Мое предложение таково: ваш крейт должен возвращать собственный тип ошибки. Это будет удобно для всех, кто его использует, потому что им будет ясно, где именно возникла эта ошибка.

Чтобы такой подход работал, нужна реализация типажа *From*, позволяющая выполнять преобразование вашего типа ошибки в целевой тип, возвращаемый функцией, в которой оператор `?` используется в тех случаях, когда типы различаются. Сделать это сравнительно несложно, поскольку компилятор, если нужно, выведет соответствующее сообщение.

Теперь предположим, что в вашем крейте имеется функция, считывающая содержимое файла, — например:

```
fn read_file(name: &str) -> Result<String, Error> {
    use std::fs::File;
    use std::io::prelude::*;

    let mut file = File::open(name)?;
    let mut contents = String::new();
```

(1)

```

file.read_to_string(&mut contents)?;           (2)
Ok(contents)
}

```

Здесь:

- в поз. (1) — оператор ? служит в этой строке для подразумеваемой обработки ошибок;
- в поз. (2) — в этой строке также присутствует оператор ?.

В приведенном коде имеется функция, открывающая файл `name`,читывающая его содержимое в строку и возвращающая это содержимое в качестве результата. Оператор ? применяется дважды: либо в случае успеха возвращается результат выполнения функции, либо в случае провала тут же возвращается ошибка. Как в `File::open`, так и в `read_to_string()` используется тип `std::io::Error`, поэтому нами предоставляется следующая реализация `From`, автоматически разрешающая нужное преобразование:

```

impl From<std::io::Error> for Error {
    fn from(other: std::io::Error) -> Self {
        Self {
            message: other.to_string(),
        }
    }
}

```

4.7. Преобразование типов с помощью `From/Into`

В базовой библиотеке Rust предоставлены два очень популярных типажа: `From` и `Into`. Просматривая стандартную библиотеку Rust, можно заметить, что из-за той пользы, которую приносит их применение, они реализованы для огромного количества различных типов. Так что эти типажи будут вам попадаться при работе с Rust достаточно часто.

Типажи `From` и `Into` предоставляют стандартный способ преобразования между типами. Иногда они также используются компилятором для автоматического преобразования типов от вашего имени.

Как правило, нужна реализация только типажа `From`, а необходимость в реализации типажа `Into` почти никогда не возникает. Типаж `Into` является функциональной противоположностью типажа `From` и будет автоматически выводиться компилятором. Но из этого правила есть одно исключение — версии Rust до 1.41 придерживались немного более строгих правил, не позволяющих реализовывать `From`, когда назначением преобразования был внешний тип.

Использование `From` предпочтительнее, поскольку он не требует указывать тип назначения, что немножко упрощает синтаксис. Сигнатура для типажа `From` (из стандартной библиотеки) выглядит следующим образом:

```
pub trait From<T>: Sized {  
    /// Performs the conversion.  
    fn from(_: T) -> Self;  
}
```

Давайте создадим очень простую обертку для `String` и реализуем этот типаж для нашего типа:

```
struct StringWrapper(String);  
  
impl From<&str> for StringWrapper {  
    fn from(other: &str) -> Self {  
        Self(other.into())      ← возвращение копии строки, заключенной в новую обертку StringWrapper  
    }  
}  
  
fn main() {  
    println!("{}", StringWrapper::from("Hello, world!").0);  
}
```

Здесь разрешается преобразование заимствованной строки из `&str` в другую строку. Чтобы в нашу строку преобразовать другую строку, нами просто вызывается функция `into()`, которая берется из типажа `Into`, реализованного для `String`. В этом примере используется как `From`, так и `Into`.

В реальной работе преобразование типов может понадобиться по разным причинам. Одной из таких причин является потребность в обработке ошибок при использовании `Result`. Когда вызывается функция, возвращающая результат, и внутри этой функции применяется оператор `?`, то в случае, если тип ошибки, возвращаемый внутренней функцией, отличается от типа ошибки, используемого `Result`, вам потребуется предоставить реализацию `From`.

Рассмотрим следующий код:

```
use std::fs::File, io::Read;  
  
struct Error(String);  
  
fn read_file(name: &str) -> Result<String, Error> {  
    let mut f = File::open(name)?;  
    let mut output = String::new();  
  
    f.read_to_string(&mut output)?;  
  
    Ok(output)  
}
```

Здесь предпринимается попытка чтения файла в строку и возвращения результата, и есть пользовательский тип ошибки, просто содержащий строку. Этот код не проходит компиляцию:

```

error[E0277]: `?` couldn't convert the error to `Error`
--> src/main.rs:6:33
|
5 | fn read_file(name: &str) -> Result<String, Error> {
|                               ----- expected `Error`
because of this
6 | let mut f = File::open(name)?;
|                               ^ the trait `From<std::io::Error>` is
not implemented for `Error`
|
= note: the question mark operation (`?`) implicitly performs a conversion
on the error value using the `From` trait
= note: required by `from`


error[E0277]: `?` couldn't convert the error to `Error`
--> src/main.rs:9:34
|
5 | fn read_file(name: &str) -> Result<String, Error> {
|                               ----- expected `Error`
because of this
...
9 |     f.read_to_string(&mut output)?;
|                               ^ the trait `From<std::io::Error>` is
not implemented for `Error`
|
= note: the question mark operation (`?`) implicitly performs a conversion
on the error value using the `From` trait
= note: required by `from`
```

Чтобы заставить его компилироваться, для `Error` нужно реализовать типаж `From` так, чтобы компилятор знал, как `std::io::Error` преобразовать в нашу собственную ошибку. Эта реализация выглядит следующим образом:

```

impl From<std::io::Error> for Error {
    fn from(other: std::io::Error) -> Self {
        Self(other.to_string())
    }
}
```

Теперь, если выполнить компиляцию и запустить код, он будет работать ожидаемым образом.

4.7.1. Типажи `TryFrom` и `TryInto`

В дополнение к типажам `From` и `Into` применяются еще и типажи `TryFrom` и `TryInto`. Они почти идентичны своим собратьям — за исключением тех случаев, когда преобразование типа может не сработать. Методы преобразования в этих трейтах воз-

вращают `Result`, а при использовании `From` и `Into` способа вернуть ошибку, кроме поднятия паники, приводящей к сбою всей программы, не имеется.

4.7.2. Наиболее рациональные приемы преобразования типов с использованием `From` и `Into`

В кратком изложении самые рациональные приемы преобразования типов с типажами `From` и `Into` можно описать следующим образом:

- типаж `From` должен быть реализован для тех типов, которые требуют преобразования в другие типы и из других типов;
- не стоит создавать пользовательские процедуры преобразования — вместо этого следует везде, где только возможно, полагаться на уже известные типажи.

4.8. Обеспечение совместимости интерфейса внешних функций с типами Rust

Иногда приходится вызывать функции из библиотек, не имеющих отношения к языку Rust (или же наоборот), и во многих случаях тогда требуется создавать в Rust модели структур языка C. Для этого необходимо воспользоваться возможностями имеющегося в Rust *интерфейса внешних функций* (Foreign Function Interface, FFI). Rust-структуры несовместимы со структурами языка C, поэтому, чтобы добиться их совместимости, следует:

- объявлять структуры с атрибутом `#[repr(C)]`, сообщающим компилятору о необходимости упаковать структуру в представление, совместимое с C;
- использовать типы C из крейта `libc`, обеспечивающего отображение типов Rust на типы C и наоборот. Типы Rust — это *не* типы C, и предполагать их повсеместную совместимость не представляется возможным, даже если полагать, что они эквивалентны.

Чтобы существенно упростить весь этот процесс, команда Rust предоставляет инструмент под названием `rust-bindgen`. С его помощью можно автоматически генерировать из C-заголовков привязки к библиотекам C. Для создания привязок в большинстве случаев нужно воспользоваться именно `rust-bindgen`, следуя при этом соответствующим инструкциям⁶.

В ряде случаев в целях тестирования или по какой-либо другой причине у меня возникала потребность вызова функций языка C, и для работы с `rust-bindgen` в простых случаях не приходилось затрачивать никаких особых усилий.

При этом процесс отображения структур языка C в Rust организуется в следующем порядке:

⁶ См. <http://mng.bz/amgj>.

1. Копирование определения структуры языка C.
2. Преобразование типов C в типы Rust.
3. Реализация интерфейсов функций.

Давайте, продолжая пример zlib из разд. 2.6, быстро создадим структуру файла zlib, имеющую на языке C следующий вид:

```
struct gzFile_s {
    unsigned have;
    unsigned char *next;
    z_off64_t pos;
};
```

После преобразования соответствующая структура на языке Rust будет выглядеть следующим образом:

```
#[repr(C)]                                     (1)
struct GzFileState {                           (2)
    have: c_uint,
    next: *mut c_uchar,
    pos: i64,
}
```

Здесь:

- в поз. (1) — предписание rustc с целью совместимости с C выровнять память в этой структуре так, как это делает компилятор C;
- в поз. (2) — структура C, представляющая состояние файла zlib, в соответствии с определением в zlib.h.

Собрав все воедино, можно вызывать функции C из zlib со структурой, предполагаемой zlib:

```
type GzFile = *mut GzFileState;

#[link(name = "z")]                                     (1)
extern "C" {
    fn gzopen(path: *const c_char, mode: *const c_char) -> GzFile;      (2)
    fn gzread(file: GzFile, buf: *mut c_uchar, len: c_uint) -> c_int; (2)
    fn gzclose(file: GzFile) -> c_int;          (2)
    fn gzeof(file: GzFile) -> c_int;          (2)
}

fn read_gz_file(name: &str) -> String {
    let mut buffer = [0u8; 0x1000];
    let mut contents = String::new();
    unsafe {
        let c_name = CString::new(name).expect("CString failed");       (3)
        let c_mode = CString::new("r").expect("CString failed");
        let file = gzopen(c_name.as_ptr(), c_mode.as_ptr());
        let len = gzread(file, buffer.as_mut_ptr(), buffer.len() as c_int);
        if len > 0 {
            contents.push_str(unsafe { std::str::from_utf8_unchecked(&buffer[0..len]) });
        }
    }
}
```

```
if file.is_null() {
    panic!(
        "Couldn't read file: {}",
        std::io::Error::last_os_error()
    );
}
while gzeof(file) == 0 {
    let bytes_read = gzread(
        file,
        buffer.as_mut_ptr(),
        (buffer.len() - 1) as c_uint,
    );
    let s = std::str::from_utf8(&buffer[..(bytes_read as usize)])
        .unwrap();
    contents.push_str(s);
}
gzclose(file);
}

contents
}
```

Здесь:

- в поз. (1) — сообщение `rustc` о принадлежности функций внешней библиотеке `z`;
- в поз. (2) — четыре внешние функции `zlib`, определенные в файле `zlib.h`;
- в поз. (3) — преобразование строки Rust в формате UTF-8 в строку C в формате ASCII с выдачей ошибки в случае сбоя;

Функция `read_gz_file()` откроет сжатый файл, считает его содержимое и вернет его в виде строки.

Резюме

- `str` — тип строки в формате UTF-8, размещенный в стеке Rust. `String` — основанная на `Vec` строка в формате UTF-8, размещенная в куче.
- `&str` — слайс строки, который может быть заимствован как у `String`, так и у `&'static str`.
- `Vec` — размещенная в куче изменяемая последовательность значений, находящаяся в непрерывной области памяти. При моделировании последовательности значений в большинстве случаев следует воспользоваться именно `Vec`.
- `HashMap` — стандартный тип контейнера хеш-отображения Rust, подходящий для большинства случаев, требующих поиска по ключу за постоянное время.
- В библиотеке коллекций Rust имеются также типы `VecDeque`, `LinkedList`, `BTreeMap`, `HashSet`, `BTreeSet` и `BinaryHeap`.

- Структуры представляют собой компонуемые контейнеры и являются основными строительными блоками Rust. Они служат для хранения состояния и реализации методов, работающих с этим состоянием.
- Перечисления являются в Rust особым вариантым типом, способным эмулировать поведение перечислений `enum` из таких языков, как C, C++, C# и Java.
- Реализации многих стандартных типажей могут быть получены с применением атрибута `#[derive]`. В случае необходимости эти типажи можно реализовать самостоятельно, но в большинстве случаев вполне достаточно автоматически полученных реализаций.

5

Работа с памятью

В этой главе:

- изучение тонкостей управления памятью в Rust на основе кучи и стека;
- описание применяемой в Rust семантики владения;
- использование указателей с подсчетом ссылок;
- приемы эффективного использования умных указателей;
- реализация пользовательских распределителей для конкретных вариантов применения.

В главе 4 мы обсудили структуры данных Rust, но чтобы в них всецело разобраться, нужно также рассмотреть вопросы управления памятью и то, как это управление работает с имеющимися в Rust структурами данных. В основных структурах данных для управления выделением и высвобождением памяти предоставлены неплохие абстракции, но некоторым приложениям могут потребоваться более совершенные функции, использующие пользовательские распределители, подсчеты ссылок, умные указатели или функции системного уровня, не относящиеся к языку Rust.

Можно, конечно, вполне эффективно пользоваться языком Rust, и не вникая в подробности управления памятью, но порой складываются такие ситуации, при которых весьма полезно знать, что происходит, так сказать, «за кулисами». И эта глава посвящена подробному рассмотрению порядка управления памятью при работе с языком Rust.

5.1. Управление памятью: куча и стек

В Rust имеется весьма мощная и детальная семантика управления памятью. Поначалу новичкам может показаться, что в ней будет трудно разобраться. Например, при использовании строки или вектора вы вряд ли задумываетесь о том, как именно выделяется память. В каком-то смысле это похоже на ситуацию при работе с таки-

ми языками сценариев, как Python или Ruby, где управление памятью в значительной степени абстрагировано и редко является чем-то, о чём стоит задумываться.

Закулисно управление памятью в Rust не слишком отличается от такового в языках С или C++. Но в Rust язык без особой надобности старается не вмешивать нас в управление памятью. А когда такая потребность всё же возникает, язык предоставляет все инструменты, необходимые для того, чтобы в той или иной степени это управление усложнить или упростить, — в зависимости от того, чего вы пытаетесь добиться. Давайте вкратце рассмотрим различия между кучей и стеком (рис. 5.1).



Рис. 5.1. Пример раскладки стека и кучи

Куча представляет собой раздел памяти для ее динамического выделения. Обычно это место в памяти, зарезервированное для структур данных с изменяемым размером или чего-либо, чей размер известен только во время выполнения программы. Это не означает, что в куче не могут храниться статические данные, но для них лучше всё же воспользоваться *стеком* (компилятор, как правило, в качестве оптимизации помещает статические данные в сегмент статической памяти, поэтому *на самом деле* в стек они не помещаются). Куча в большинстве случаев управляет自己 используемой операционной системой или библиотеками основного языка, но при желании программисты могут реализовать свою собственную кучу. Для систем с ограниченной памятью — например, для встроенных систем, код, как правило, пишется без кучи.

Куча, как уже отмечалось, управляет自己 операционной системой, средой выполнения языка или библиотекой языка С с помощью предоставляемого ими распределителя (например, `malloc()`). Данные в куче можно рассматривать в качестве распреде-

ленных случайным образом по всей куче, и в течение жизненного цикла процесса их объем может увеличиваться и уменьшаться.

В листинге 5.1 показано, что в Rust распределение в куче выполняется с использованием любой структуры данных, для которой предусмотрено такое распределение, — в качестве примера можно привести `Vec` или `Box` (более подробно структура `Box` будет рассмотрена чуть позже).

Листинг 5.1. Код, показывающий значения, для которых память выделяется в куче

```
let heap_integer = Box::new(1);
let heap_integer_vec = vec![0; 100];
let heap_string = String::from("heap string");    ← как отмечалось в главе 4, тип String основан на Vec,
                                                    что превращает его в строку, размещенную в куче
```

Стек представляет собой локальное пространство памяти потока, привязанное к области видимости функции. Память для стека выделяется с использованием порядка «последним вошел, первым вышел» (Last In, First Out, LIFO): при входе в функцию происходит выделение памяти и ее проталкивание в стек, а при выходе из функции память высвобождается и выталкивается из стека. Размер тех данных, для которых память выделяется в стеке, должен быть известен во время компиляции. Выделение памяти в стеке происходит обычно намного быстрее, чем при использовании кучи. В операционных системах, поддерживающих стековую память, на каждый поток выполнения приходится один стек.

Стек управляет самой программой на основе кода, сгенерированного компилятором. При входе в функцию в стек помещается новый фрейм (добавляется в конец стека), а при выходе из функции фрейм из стека удаляется. Программистам об управлении стеком не стоит озабочиваться, поскольку вся работа со стеком выполняется системой за них. У стека имеется целый ряд полезных свойств (отметим, в частности, скорость его работы), а стек вызовов функций, как показано в листинге 5.2, может использоваться в качестве структуры данных при выполнении рекурсивных вызовов, что исключает необходимость беспокоиться по поводу управления памятью.

Листинг 5.2. Код, показывающий значения, размещенные в стеке

```
let stack_integer = 69420;
let stack_allocated_string = "stack string";
```

Концепции стека и кучи во многих языках запутаны или абстрагированы, поэтому разбираться с этим нам сейчас не имеет никакого смысла. Отметим только, что в C и C++ память в куче обычно выделяется с помощью `malloc()` или ключевого слова `new`, а чтобы протолкнуть значение переменной в стек, достаточно просто объявить ее в функции. В Java для выделения памяти в куче также имеется ключевое слово `new`, но в Java память обрабатывается сборщиком мусора, и самостоятельно управлять очисткой кучи не приходится.

В Rust стек управляет компилятором и конкретными средствами реализации платформы. Впрочем, распределение памяти для данных в куче можно настроить в соответствии с вашими потребностями (пользовательские распределители будут рассмотрены чуть позже), и это похоже на приемы, используемые для этого в C или C++.

В стеке могут размещаться только примитивные и составные типы (например, кортежи и структуры), str и сами контейнерные типы (но не обязательно их содержимое).

5.2. Представление о владении: копирование, заимствование, ссылки и перемещения

В Rust введена новая концепция программирования, называемая *владением* и представляющая собой часть особенностей, отличающих его от других языков. Владение в Rust является основой его гарантий безопасности и предоставляет компилятору возможность узнавать, когда память находится в области видимости, является совместно используемой, вышла из области видимости или используется неправильно. Имеющаяся в компиляторе проверка заимствования отвечает за обеспечение соблюдения небольшого набора правил владения: у каждого значения есть владелец, в каждый конкретный момент времени у значения может быть только один владелец, и когда владелец выходит из области видимости, значение удаляется.

Если с владением в Rust вы уже ранее освоились, материал этого раздела даст вам возможность повторить пройденное, хотя при желании его можно и пропустить. Но если вы всё же собираетесь разобраться с владением в Rust, этот материал поможет вам прояснить новую для вас концепцию на основе уже знакомых понятий.

Семантика владения Rust в некотором роде похожа на C, C++ и Java, за исключением того, что в Rust отсутствуют концепции *конструкторов копирования* (создающих копию объекта при присваивании), причем взаимодействие с обычными указателями в Rust организуется довольно редко. Когда значение одной переменной присваивается другой переменной (например, let a = b;), это называется *перемещением*, представляющим собой передачу права собственности (а мы помним, что у значения может быть только один владелец). При перемещении значения не создается его копия, если только не присваивается базовый тип (т. е. присваивание целого числа другому значению создает копию).

Вместо использования указателей данные в Rust зачастую передаются с помощью ссылок. Ссылка в Rust создается путем *заимствования*. Данные могут передаваться в функции по значению (что является перемещением) или по ссылке. Хотя в Rust имеются указатели, подобные тем, что используются в C, в Rust они, как уже отмечалось, встречаются достаточно редко, за исключением, возможно, случаев взаимодействия с кодом на языке C.

Заимствованные данные (по ссылке) могут быть как неизменяемыми, так и изменяемыми. По умолчанию при заимствовании данных срабатывает неизменяемый вариант (т. е. данные, на которые указывает ссылка, изменять нельзя). Но если при

заимствований указывается ключевое слово `mut`, может получиться изменяемая ссылка, позволяющая изменять данные. Неизменяемые данные могут заимствоваться многократно и одновременно (т. е. на них может быть сразу несколько ссылок), но многократное одновременное изменяемое заимствование одних и тех же данных не допускается.

Заимствование обычно выполняется с помощью оператора `&` (или `&mut` — для изменяемого заимствования); но порой можно заметить, что вместо этого используются методы `as_ref()` или `as_mut()`, которые берутся из типажей `AsRef` и `AsMut` соответственно. Методы эти часто задействуются контейнерными типами не для получения ссылки на сам контейнер, а для предоставления доступа к его внутренним данным (что будет рассмотрено чуть позже). Чтобы прояснить эти концепции, рассмотрим код листинга 5.3.

Листинг 5.3. Пример кода для демонстрации владения

```
fn main() {
    let mut top_grossing_films =
        vec!["Avatar", "Avengers: Endgame", "Titanic"]; (1)
    let top_grossing_films_mutable_reference =
        &mut top_grossing_films; (2)
    top_grossing_films_mutable_reference
        .push("Star Wars: The Force Awakens");
    let top_grossing_films_reference = &top_grossing_films; (3)
    println!(
        "Printed using immutable reference: {:?}", (4)
        top_grossing_films_reference
    );
    let top_grossing_films_moved = top_grossing_films; (5)
    println!("Printed after moving: {:?}", top_grossing_films_moved); (6)
    // println!("Print using original value: {:?}", top_grossing_films); (7)
    // println!()
    //     "Print using mutable reference: {:?}", (8)
    //     top_grossing_films_mutable_reference
    // ); (9)
}
```

Здесь:

- в поз. (1) — создание изменяемого `Vec` и заполнение его рядом значений;
- в поз. (2) — заимствование изменяемой ссылки на предыдущий `Vec`;
- в поз. (3) — эта изменяемая ссылка может использоваться для изменения данных, заимствованных в предыдущей строке;
- в поз. (4) — теперь берется неизменяемая ссылка на те же данные, после чего прежняя изменяемая ссылка становится недействительной;
- в поз. (5) — вывод на экран содержимого `Vec`;

- в поз. (6) — это присваивание является перемещением, передающим право собственности на Vec;
- в поз. (7) — вывод на экран содержимого Vec после его перемещения;
- в поз. (8) — исходная переменная из-за перемещения больше недействительна, поэтому код строки не пройдет компиляцию;
- в поз. (9) — этот код также не скомпилируется, поскольку данная ссылка после создания неизменяемой ссылки стала недействительной.

Выполнение приведенного кода даст следующий результат:

```
Printed using immutable reference: [
    "Avatar",
    "Avengers: Endgame",
    "Titanic",
    "Star Wars: The Force Awakens",
]
Printed after moving: [
    "Avatar",
    "Avengers: Endgame",
    "Titanic",
    "Star Wars: The Force Awakens",
]
```

5.3. Глубокое копирование

Возможно, концепция *глубокого копирования* вам уже попадалась в других языках — например, в Python или в Ruby. Глубокое копирование требуется, когда для предотвращения копирования данных в языке или в структурах данных реализуется оптимизация, проводимая чаще всего с помощью указателей, ссылок и семантики копирования при записи.

Копии структур данных могут быть либо *поверхностными* (копирование указателя или создание ссылки), либо *глубокими* (рекурсивное копирование или клонирование всех значений внутри структуры). В некоторых языках при присваивании ($a = b$) или при вызове функции по умолчанию выполняется поверхностное копирование. А если ранее вы работали с такими языками, как Python, Ruby или JavaScript, вам, возможно, уже приходилось выполнять явно заданное глубокое копирование. В Rust никаких предположений о ваших намерениях не выстраивается, поэтому указания о предстоящих действиях даются компилятору явным образом. Иными словами, концепции поверхностного копирования в Rust не существует, но вместо этого имеются заимствования и ссылки.

В языках, использующих неявные ссылки на данные, могут создаваться нежелательные побочные эффекты, застающие порой врасплох или выдающие ошибки, трудно поддающиеся обнаружению. Проблема обычно возникает, когда предполагается создание копии, а язык вместо этого предоставляет ссылку. И хорошо, что

в Rust до тех пор, пока вы придерживаетесь основных структур данных, не делается никаких неявных ссылок на данные.

Для описания процесса создания новой структуры данных и копирования (или, правильнее сказать, клонирования) всех данных из старой структуры в новую в Rust используется понятие **клонирования** (а не копирования). Операция обычно выполняется с помощью метода `clone()`, который берется из типажа `Clone` и может быть автоматически получен с помощью атрибута `#[derive(Clone)]` (типажи и получаемые типажи более подробно рассматриваются в главах 8 и 9). С реализованным для вас типажом `Clone` в Rust поставляются многие структуры данных, поэтому обычно на доступность `clone()` вполне можно рассчитывать.

Рассмотрим код листинга 5.4.

Листинг 5.4. Код для демонстрации `clone()`

```
fn main() {  
    let mut most_populous_us_cities =  
        vec!["New York City", "Los Angeles", "Chicago", "Houston"];  
    let most_populous_us_cities_cloned = most_populous_us_cities.clone();      (1)  
    most_populous_us_cities.push("Phoenix");                                     (2)  
    println!("most_populous_us_cities = {:?}", most_populous_us_cities);  
    println!()  
        "most_populous_us_cities_cloned = {:?}",  
        most_populous_us_cities_cloned  
    );  
}
```

Здесь:

- в поз. (1) — клонирование исходного `Vec`;
- в поз. (2) — добавление нового города в список в исходном `Vec`;
- в поз. (3) — при выводе на экран клонированного `Vec` слово "Phoenix" будет отсутствовать, поскольку это уже совершенно другая структура.

При выполнении этого кода будет выведено:

```
most_populous_us_cities = [  
    "New York City",  
    "Los Angeles",  
    "Chicago",  
    "Houston",  
    "Phoenix",  
]  
most_populous_us_cities_cloned = [  
    "New York City",  
    "Los Angeles",  
    "Chicago",  
    "Houston",  
]
```

Полученный типаж `Clone` работает рекурсивно. Получается, что вызов `clone()` для любой структуры данных верхнего уровня, такой как `Vec`, вполне достаточен для создания глубокой копии содержимого `Vec` при условии, что во всех таких структурах реализован `Clone`. Глубоко вложенные структуры можно легко клонировать, не делая ничего, кроме проверки наличия в них реализации типажа `Clone`.

5.4. Предотвращение копирования

Случается так, что структуры данных, возможно, непреднамеренно, могут клонироваться или копироваться чаще, чем надо. Это может происходить при обработке строк — например, когда множество копий строки создается повторно в алгоритмах, выполняющих сканирование, вносящих изменения или же обрабатывающих некий произвольный набор данных каким-либо другим образом.

Одним из недостатков `Clone` является, наверное, слишком легкая возможность копирования структур данных, позволяющая при неумеренном применении этого типа-жа получать множество копий одних и тех же данных. Для большинства замыслов и намеченных целей это не проблема, и она вряд ли возникнет, пока не начнется работа с очень большими наборами данных.

Многие основные библиотечные функции Rust не изменяют объекты в месте их расположения, а возвращают копии этих объектов. В большинстве случаев такое поведение считается более предпочтительным, поскольку за счет дублирования, возможно, временного, участков памяти помогает поддерживать неизменность данных. Давайте в качестве иллюстрации рассмотрим несколько строковых операций из основной библиотеки Rust, представленных в табл. 5.1.

В операциях, представленных в табл. 5.1, можно заметить закономерность, заключающуюся в том, что зачастую будет нетрудно определить, создает алгоритм копию или нет, исходя из того, изменяет ли функция исходные данные на месте или же возвращает новую копию. В качестве еще одной иллюстрации можно привести то, что я называю *проходом*. Рассмотрим следующий код:

```
fn lowercased(s: String) -> String {
    s.to_lowercase()                                (1)
}
```

```
fn lowercased_ascii(mut s: String) -> String {
    ss.make_ascii_lowercase();                      (2)
}
```

Здесь:

- в поз. (1) — создание копии внутри `to_lowercase()` и возвращение новой строки;
- в поз. (2) — непосредственный проход строки с изменяемой на прежнем месте памятью. Право собственности возвращается вызывающей стороне путем возврата ей того же находящегося в собственности объекта, поскольку вся работа функции `make_ascii_lowercase()` осуществляется на месте.

Таблица 5.1. Изучение основных строковых функций Rust с позиции копирования

Функция	Описание	Копии?	Алгоритм	Чем идентифицируется?
<code>pub fn replace<'a, P>(&'a self, from: P, to: &str) -> String where P: Pattern<'a>,</code>	Заменяет все совпадения паттерна другой строкой	Да	Создает новую строку, помещает обновленное содержимое в новую строку, возвращает новую строку и оставляет исходную строку нетронутой	Параметр <code>self</code> является неизменяемой ссылкой. Функция возвращает принадлежащую ей строку <code>String</code>
<code>pub fn to_lowercase(&self) -> String</code>	Возвращает эквивалент фрагмента строки с символами в нижнем регистре в виде новой строки	Да	Создает новую строку, копирует каждый символ в новую строку, преобразуя заглавные символы в строчные	Параметр <code>self</code> является неизменяемой ссылкой. Функция возвращает принадлежащую ей строку <code>String</code>
<code>pub fn make_ascii_lowercase(&mut self)</code>	Преобразует строку в ее эквивалент с символами в нижнем регистре в формате ASCII на месте расположения этой строки	Нет	Выполняет итерацию по каждому символу, применяя преобразование в нижний регистр к заглавным символам ASCII	Функция принимает изменяемую ссылку <code>self</code> , внося изменения в память на месте
<code>pub fn trim(&self) -> &str</code>	Возвращает фрагмент строки с удаленными начальными и конечными пробелами	Нет	Применяет поисковую функцию для поиска начала и конца подстроки без пробелов и возвращает фрагмент строки с удаленными начальными и конечными пробелами	Функция возвращает ссылку, а не принадлежащую ей строку

В приведенном коде первая функция, `lowercased()`, принимает находящуюся в собственности строку, а возвращает новую копию этой строки, вызывая `to_lowercase()`. Вторая функция (работающая только со строками в формате ASCII) берет изменяемую, находящуюся в собственности строку и возвращает в *то же место* ее версию в нижнем регистре.

Вот краткое изложение того, что мы узнали о функциях по части копирования:

- функции, принимающие неизменяемые ссылки и возвращающие ссылку или слайс, вряд ли будут создавать копии (например: `fn func(&self) -> &str`);
- функции, принимающие ссылку (т. е. `&`) и возвращающие объект, находящийся в собственности, могут создавать копии (например: `fn func(&self) -> String`);
- функции, принимающие изменяемую ссылку (т. е. `&mut`), могут изменять данные на месте (например: `fn func(&mut self)`);
- функции, принимающие объект, находящийся в собственности, и возвращающие объект, находящийся в собственности того же типа, скорее всего, создают копию (например: `fn func(String) -> String`);

- функции, принимающие изменяемый, находящийся в собственности объект и возвращающие находящийся в собственности объект того же типа, скорее всего, не создают копию (например: `fn func(mut String) -> String`).

Если нет полной уверенности в том, как поступает функция: делает ли копию, работает ли на месте или же просто передает владение, следует всё же обратиться к документации и исходному коду. Семантика, используемая в Rust при работе с памятью, позволяет относительно легко понять, как алгоритмы работают с данными, просто рассматривая то, что у них на входе и на выходе, но это срабатывает только при условии, что вызываемые функции следуют соответствующим паттернам. В случаях возникновения серьезных проблем с производительностью следует внимательно изучить базовые алгоритмы.

5.5. Умные указатели: тип `Box`

Тип `Box` в Rust представляет собой вариант умного указателя, краткие сведения о котором уже приводились в главе 4. `Box` несколько отличается от умных указателей в таких языках, как C++, поскольку его основная цель — предоставить способ выделения данных в куче. В Rust есть два основных способа выделения данных в куче: использование `Vec` или применение `Box`. По возможностям `Box` не представляет собой ничего особо выдающегося — им обрабатывается только выделение и высвобождение памяти для содержащегося в нем объекта и не предоставляется никакой другой функциональности, но, собственно, так и было задумано. Польза от применения `Box` не вызывает сомнений, и этот тип должен быть первым, к которому нужно обращаться в тех случаях, когда надо сохранить данные в куче (кроме, конечно, применения `Vec`).

Совет

Поскольку `Box` не может быть пустым (за исключением ряда ситуаций, на которых лучше не останавливаться), он зачастую содержится внутри `Option` на случай, когда упакованные данные могут отсутствовать.

Если данные или объект присутствуют опционально, `Box` следует размещать в `Option`. *Опциональные* (или *вероятные*) типы не являются уникальными только для Rust — с ними навскидку можно повстречаться в таких языках, как Ada, Haskell, Scala или Swift. Опционалы представляют собой разновидность *монад* — паттернов функционального проектирования, позволяющих обертывать значения, не предназначенные для неограниченного доступа, в функцию. Чтобы работа с опционалами стала приятнее, в Rust предоставляется соответствующее синтаксическое сокращение.

Тип `Option` будет встречаться в Rust довольно часто, и если ранее с опционалами вам еще не приходилось сталкиваться, вы можете воспринимать их в качестве способа безопасной работы с нулевыми значениями (например, с указателями). В Rust (исключая небезопасный код) не бывает нулевых указателей, но есть `None`, являющийся функциональным эквивалентом нулевого указателя и не создающий никаких проблем с безопасностью.

Особенно впечатляет эффективность совместного применения `Box` и `Option`, практически не допускающая возможности получения ошибки времени выполнения (например, исключающая нулевой указатель) из-за недействительной, неинициализированной или дважды высвобожденной памяти. И всё же есть одно предостережение: выделение памяти в куче может завершиться сбоем. Обработка такой ситуации реализуется непросто, она далека от тем, освещаемых в этой книге, и в определенной степени зависит от операционной системы и ее настроек.

Распространенной причиной сбоев выделения памяти является нехватка свободной памяти в системе, и обработка такого сбоя (если все же решиться на это) во многом зависит от приложения. В большинстве случаев ожидаемым результатом сбоя при выделении памяти является крах программы и выход из нее с ошибкой *нехватки памяти* (*Out Of Memory*, OOM), что практически всегда является поведением по умолчанию (т. е. именно так и произойдет, если разработчик не станет обрабатывать ошибки OOM). Вероятно, вам уже приходилось сталкиваться с подобными ситуациями. К известным приложениям, предоставляющим собственные функции управления памятью, относятся веб-браузеры, у которых, как и у операционных систем, зачастую имеются свои собственные встроенные диспетчеры задач и управления памятью. Корректная обработка сбоев выделения памяти может потребоваться при создании критически важного программного средства — например, базы данных или системы онлайн-обработки транзакций.

На случай возникновения подозрений на возможные сбои при выделении памяти в `Box` предоставляется метод `try_new()`, возвращающий `Result`. Этот тип, как и `Option`, может находиться как в состоянии успеха, так и в состоянии сбоя. Имеющийся в `Box` исходный метод `new()` в случае сбоя выделения памяти вызовет панику, приводящую к сбою программы. В большинстве случаев такой сбой и является лучшим способом обработки сбоев выделения памяти. В качестве альтернативы можно перехватывать сбои выделения памяти в пользовательском распределителе (который будет рассматриваться чуть позже).

СОВЕТ

Чтобы более детально разобраться в типах `Option` и `Result`, попробуйте реализовать их самостоятельно с помощью перечисления. В Rust при создании и использовании собственных опционалов обычно применяются перечисления и сопоставления с образцом.

Чтобы проиллюстрировать применение `Box`, рассмотрим созданный на языке Rust элементарный односвязный список (листинг 5.5).

Листинг 5.5. Код для простого односвязного списка в Rust

```
struct ListItem<T> {
    data: Box<T>,                                     (1)
    next: Option<Box<ListItem<T>>>,                (2)
}

struct SinglyLinkedList<T> {
    head: ListItem<T>,                                (3)
}
```

```

impl<T> ListItem<T> {
    fn new(data: T) -> Self {
        ListItem {
            data: Box::new(data),                                (4)
            next: None,                                         (5)
        }
    }
    fn next(&self) -> Option<&Self> {                   (6)
        if let Some(next) = &self.next {                     (7)
            Some(next.as_ref())                            (8)
        } else {
            None
        }
    }
    fn mut_tail(&mut self) -> &mut Self {
        if self.next.is_some() {                           (9)
            self.next.as_mut().unwrap().mut_tail()          (10)
        } else {
            Self                                         (11)
        }
    }
    fn data(&self) -> &T {
        self.data.as_ref()                                (12)
    }
}

impl<T> SinglyLinkedList<T> {
    fn new(data: T) -> Self {
        SinglyLinkedList {
            head: ListItem::new(data),                      (13)
        }
    }
    fn append(&mut self, data: T) {
        let mut tail = self.head.mut_tail();                (14)
        tail.next = Some(Box::new(ListItem::new(data)));     (15)
    }
    fn head(&self) -> &ListItem<T> {
        &self.head                                         (16)
    }
}

```

Здесь:

- в поз. (1) — в каждом элементе списка имеются упакованные в нем данные. Поле данных не может быть пустым или нулевым;
- в поз. (2) — next является опционалом, значение None обозначает конец списка;
- в поз. (3) — структура самого списка содержит только заголовок, упаковка которого нас не волнует, поскольку его присутствие обязательно;

- в поз. (4) — новые данные перемещаются в список внутри `Box`, при этом память выделяется в куче;
- в поз. (5) — указатель `next` инициализируется значением `None`, поскольку новые элементы пока не знают, где их место в списке. Кроме того, в этой реализации отсутствуют операции вставки и имеется только операция добавления;
- в поз. (6) — метод `next()` возвращает для каждого элемента optionalную ссылку на следующий элемент, если таковой существует. Эта функция нужна, чтобы помочь распаковать вложенные ссылки с целью упрощения кода;
- в поз. (7) — такая конструкция кода используется для того, чтобы перед попыткой разыменования следующего указателя проверить, указывает ли он на что-либо;
- в поз. (8) — возвращение внутренней ссылки на следующий элемент, что эквивалентно коду `Some(&*next)`;
- в поз. (9) — использование `if let ...` здесь не сработает, поскольку мы не можем одновременно заимствовать `self.next` и возвращать изменяемую ссылку на внутренний указатель;
- в поз. (10) — наличие `Box` внутри `Option` требует распаковать `Option` из изменяемой ссылки и вернуть изменяемую ссылку изнутри;
- в поз. (11) — если следующего элемента нет, этот элемент является хвостом, и возвращается просто `self`;
- в поз. (12) — этот метод обеспечивает удобную ссылку на `T`;
- в поз. (13) — для нового списка нужен первый элемент. Чтобы допустить наличие пустого списка, элемент `head` должен быть optionalным;
- в поз. (14) — при добавлении нового элемента можно предположить, что следующим элементом хвоста будет `None`;
- в поз. (15) — новый элемент добавляется к следующему указателю элемента хвоста и становится новым хвостом;
- в поз. (16) — для удобства прямой доступ к элементу `head` предоставляется через метод.

В приведенном примере связанного списка есть многое, в чем следует разобраться. Для Rust-новичка реализация связанного списка является одним из лучших способов узнать об уникальных возможностях языка Rust. В коде из листинга 5.5 предоставлен целый ряд великолепных и абсолютно безопасных функциональных возможностей. Список никогда не будет пустым, недействительным или содержащим нулевые указатели. И это на самом деле впечатляющая особенность Rust, обусловленная исключительно наличием правил этого языка о владении объектами.

Только что созданный связанный список может быть протестирован с помощью приведенного далее кода:

```

fn main() {
    let mut list = SinglyLinkedList::new("head");           (1)
    list.append("middle");                                (1)
    list.append("tail");                                 (1)
    let mut item = list.head();                          (2)
    loop {                                              (3)
        println!("item: {}", item.data());                (4)
        if let Some(next_item) = item.next() {            (5)
            item = next_item;
        } else {
            break;                                     (6)
        }
    }
}

```

Здесь:

- в поз. (1) — создание нового связанного списка строк с элементом-заголовком и последующим добавлением среднего (`middle`) и хвостового (`tail`) элементов;
- в поз. (2) — получение ссылки на заголовок списка;
- в поз. (3) — выполнение до тех пор, пока не завершится перебор всех элементов списка;
- в поз. (4) — вывод на экран значений каждого элемента;
- в поз. (5) — извлечение следующего элемента в списке с помощью инструкции `if let`, в которой происходит распаковка `Option`;
- в поз. (6) — прекращение цикла с помощью `break` при достижении конца списка, узнать о котором позволяет значение `None`, оказавшееся в следующем элементе.

Выполнение этого кода даст такой результат:

```

item: head
item: middle
item: tail

```

СОВЕТ

Прежде чем продолжать чтение, вам было бы неплохо уделить немного времени изучению односвязного списка в Rust, поскольку в следующем разделе будет создаваться более совершенная версия связанного списка. Попробуйте реализовать односвязный список самостоятельно с нуля, обращаясь при необходимости к приведенному здесь примеру. Если же вам тем не менее захочется продолжить чтение, то при желании как следует разобраться с управлением памятью в Rust вам после получения достаточного представления о языке в целом всё же целесообразнее будет вернуться к выполнению предлагаемого упражнения.

А впрочем, на практике вам вряд ли когда-либо понадобится в Rust собственная реализация связанного списка. На случай возникновения надобности в связанным списке в основной библиотеке Rust предоставляется вызов `std::collections::LinkedList`. Но в большинстве случаев вполне достаточно будет просто воспользово-

ваться типом `Vec`. Кроме того, следует учесть, что приведенный здесь пример не оптимизирован.

5.6. Подсчет ссылок

В предыдущем разделе говорилось, что `Box` является полезным, но весьма ограниченным умным указателем, который будет вам периодически встречаться. В частности, `Box` ограничен в том, что он не допускает совместного использования. То есть в Rust-программе не может быть двух отдельных `Box`'ов, указывающих на одни и те же данные, — `Box` является владельцем своих данных и не допускает за раз более одного заимствования. Это по большей части полезная (или же бесполезная) особенность, несомненно, заслуживающая внимания. Но порой возникает насущная потребность совместного использования данных — возможно, несколькими потоками выполнения, или в силу того, что одни и те же данные для разной адресации хранятся сразу в нескольких структурах (например, в `Vec` и в `HashMap`).

В тех случаях, когда `Box` не подходит, вам, наверное, понадобятся умные указатели с подсчетом ссылок. Подсчет ссылок является весьма распространенным в управлении памятью приемом, позволяющим избежать отслеживания количества существующих копий указателя и высвобождать память, когда копий больше не останется. Реализация подсчета ссылок обычно основана на поддержании статического счетчика количества копий того или иного указателя, увеличении счетчика при каждом создании новой копии и уменьшении его, когда копия уничтожается. Если счетчик когда-либо достигнет нуля, память можно будет высвободить, поскольку это будет означать, что копий указателя больше не существует и, следовательно, память больше не используется или недоступна.

СОВЕТ

Реализация умного указателя с подсчетом ссылок — увлекательное упражнение для самостоятельного выполнения, но в Rust это сделать непросто, и для него потребуется использование простых (т. е. небезопасных) указателей. Если упражнение со связанным списком окажется для вас слишком простым, попробуйте создать собственный умный указатель с подсчетом ссылок.

В Rust предоставляются два разных указателя с подсчетом ссылок:

- `Rc` — однопоточный умный указатель с подсчетом ссылок, обеспечивающий совместное владение объектом;
- `Arc` — многопоточный умный указатель с подсчетом ссылок, обеспечивающий совместное владение объектами между потоками.

Однопоточные и многопоточные объекты в Rust

Во многих языках программирования функции или объекты, которые могут применяться в разных потоках, делятся по отношению к потокам на безопасные и небезопасные. В Rust это различие напрямую не проявляется, т. к. в нем всё изначально безопасно. Но вместо этого некоторые объекты могут перемещаться или синхронизироваться между потоками, а некоторые нет. Такое поведение зависит от того, реализуются ли в объекте типажи `Send` и `Sync`, более подробно рассматриваемые в главе 6.

Применительно к `Rc` и `Arc` надо отметить, что в `Rc` типажи `Send` ИЛИ `Sync` не предоставляются (фактически в `Rc` эти типажи явно помечены как нереализованные), поэтому `Rc` можно использовать только в одном потоке. А вот в `Arc` реализован как `Send`, так и `Sync`, что позволяет задействовать его в многопоточном коде.

В частности, в `Arc` применяются атомарные счетчики, зависящие от платформы и обычно реализуемые на уровне операционной системы или центрального процессора. Атомарные операции обходятся дороже обычной арифметики, поэтому `Arc` следует использовать, когда действительно нужна атомарность.

Важно отметить, что пока для обхода правил языка не используется ключевое слово `unsafe`, код `Rust` всегда безопасен. С другой стороны, если как следует не разобраться в уникальных паттернах и терминологии этого языка, заставить скомпилироваться созданный на нем код может быть непросто.

Эффективное применение указателей с подсчетом ссылок не обойдется без ввода в обиход еще одной `Rust`-концепции, которая называется *внутренней изменчивостью*. Это именно то, что может понадобиться, когда проверка заимствований `Rust` не обеспечивает достаточной степени гибкости при работе с изменяемыми ссылками. Если у вас это вызвало ассоциацию с аварийным выходом, то похлопайте сами себя по плечу за догадливость, — это именно он. Но не стоит беспокоиться — здесь не нарушаются никакие меры безопасности `Rust`, и он по-прежнему позволяет создавать безопасный код.

Чтобы включить внутреннюю изменчивость, в `Rust` нужно ввести два специальных типа: `Cell` и `RefCell`. `Rust`-новички с этими типами еще вряд ли сталкивались, и мало-вероятно, что с ними им придется встретиться при обычных обстоятельствах. В большинстве случаев возникнет потребность в использовании `RefCell`, а не `Cell`, поскольку `RefCell` позволяет заимствовать ссылки, а `Cell` перемещает значения в себя и из себя (что, вероятно, не является слишком востребованным поведением).

Польза от применения `RefCell` и `Cell` заключается еще и в том, что они позволяют предоставить `Rust`-компилятору больше информации о способе заимствования данных. При всех своих достоинствах компилятор всё же ограничен в гибкости — бывает, что абсолютно безопасный код не будет компилироваться, поскольку компилятор не поймет, что именно вы пытаетесь сделать (независимо от того, насколько это может быть правильно).

Потребностей в частом применении `RefCell` или `Cell` у вас не возникнет, а если же обнаружится, что попытка их применения связана с обходом проверки заимствования, то вам, возможно, придется переосмыслить всё то, что вы делаете. Эти типы в основном нужны для особых случаев — например, при использовании контейнеров и структур данных, содержащих данные, к которым нужно получить доступ с возможностью их изменения.

Одно из ограничений применения `Cell` и `RefCell` связано с тем, что они предназначены только для однопоточных приложений. В случае, когда требуется соблюдение безопасности между потоками выполнения, можно воспользоваться типами `Mutex` или `RwLock`, которые предоставляют ту же функциональную возможность для включения внутренней изменчивости, но могут использоваться между потоками. Обыч-

но они будут работать в паре с Arc, а не с Rc (более подробно конкурентность будет рассматриваться в главе 10).

Давайте обновим пример связанного списка из предыдущего раздела (см. листинг 5.5), чтобы вместо Box воспользоваться Rc и RefCell, что придаст коду больше гибкости. В частности, теперь, как показано в листинге 5.6, можно будет сделать наш односвязный список двусвязным. Реализовать это при использовании Box не представляется возможным, поскольку этот тип не допускает совместного владения.

Листинг 5.6. Код двусвязного списка с использованием Rc, RefCell и Box

```
use std::cell::RefCell;
use std::rc::Rc;

struct ListItem<T> {
    prev: Option<ItemRef<T>>,                                     (1)
    data: Box<T>,                                                 (2)
    next: Option<ItemRef<T>>,
}

type ItemRef<T> = Rc<RefCell<ListItem<T>>;                      (3)

struct DoublyLinkedList<T> {
    head: ItemRef<T>,
}

impl<T> ListItem<T> {
    fn new(data: T) -> Self {
        ListItem {
            prev: None,
            data: Box::new(data),                                         (4)
            next: None,
        }
    }
    fn data(&self) -> &T {
        self.data.as_ref()
    }
}

impl<T> DoublyLinkedList<T> {
    fn new(data: T) -> Self {
        DoublyLinkedList {
            head: Rc::new(RefCell::new(ListItem::new(data))),
        }
    }
}
```

```

fn append(&mut self, data: T) {
    let tail = Self::find_tail(self.head.clone()); (5)
    let new_item = Rc::new(RefCell::new(ListItem::new(data))); (6)
    new_item.borrow_mut().prev = Some(tail.clone()); (7)
    tail.borrow_mut().next = Some(new_item); (8)
}
fn head(&self) -> ItemRef<T> {
    self.head.clone()
}
fn tail(&self) -> ItemRef<T> {
    Self::find_tail(self.head())
}
fn find_tail(item: ItemRef<T>) -> ItemRef<T> {
    if let Some(next) = &item.borrow().next { (9)
        Self::find_tail(next.clone()) (10)
    } else {
        item.clone() (11)
    }
}
}

```

Здесь:

- в поз. (1) — добавление указателя на предыдущий элемент списка;
- в поз. (2) — данные по-прежнему хранятся в Box. Здесь не нужно использовать Rc, поскольку мы делимся не правом собственности на данные, а всего лишь указателями на узлы в списке;
- в поз. (3) — этот псевдоним типа помогает поддерживать чистоту кода;
- в поз. (4) — здесь данные перемещаются в Box;
- в поз. (5) — сначала нужно найти указатель на последний элемент списка;
- в поз. (6) — создание указателя на намеченный к добавлению новый элемент;
- в поз. (7) — обновление указателя prev в новом элементе, чтобы он указывал на предыдущий хвост;
- в поз. (8) — обновление следующего указателя предыдущего хвоста, чтобы он указывал на новый хвост, являющийся только что вставленным элементом;
- в поз. (9) — проверка, является ли следующий указатель пустым, и продолжение рекурсивного поиска, если не является;
- в поз. (10) — клонирование следующего указателя и его возвращение с продолжением поиска;
- в поз. (11) — если следующий указатель пуст, значит, мы в конце (или в хвосте) списка. Возвращение текущего указателя элемента после его клонирования.

Эта версия связанного списка выглядит совершенно не так, как предыдущая. Введение Rc и RefCell несколько усложняет код, но дает гораздо больше гибкости. Далее

мы еще вернемся к этому примеру — когда дело дойдет до изучения дополнительных возможностей языка. Подводя итог, можно сказать, что `Rc` и `Arc` предоставляют указатели с подсчетом ссылок, но для доступа к внутренним данным с возможностью изменения надо будет воспользоваться объектом типа `RefCell` или `Cell` (а для многопоточных приложений — объектом `Mutex` или `RwLock`).

5.7. Клонирование при записи

Ранее в этой главе уже рассматривался вопрос избавления от копий. Но порой предпочтение всё же отдается созданию копий данных, а не изменению этих данных на месте. У такого подхода есть целый ряд полезных функциональных возможностей, особенно если предпочтение отдается паттернам функционального программирования. Возможно, вы еще ничего не слышали о *клонировании при записи*, но с *копированием при записи* вы, вероятно, уже знакомы.

Копирование при записи — это паттерн проектирования, в котором данные никогда не изменяются на месте. Вместо этого при каждой потребности внесения изменений они копируются в новое место и там подвергаются изменению, после чего возвращается ссылка на новую копию данных. В некоторых языках программирования применение этого паттерна — дело принципа, и в качестве примера можно привести язык `Scala`, где структуры данных классифицируются как изменяемые или неизменяемые, и во всех неизменяемых структурах реализуется копирование при записи. На этом паттерне целиком основана такая весьма популярная библиотека `JavaScript`, как `Immutable.js`, причем все изменения структур данных приводят к созданию новой копии данных. Построение структур данных на основе этого паттерна существенно упрощает рассуждения о том, как данные обрабатываются в программах.

Например, применительно к списку или массиву с копированием при записи операция добавления вернет новый список со всеми старыми элементами, а также добавленным новым элементом, оставляя при этом исходный список элементов нетронутым. Программист при этом предполагает, что компилятор способен справиться с оптимизацией и очисткой старых данных.

В `Rust` этот подход называется *клонированием при записи*, т. к. его применение зависит от использования типажа `Clone`. У `Clone` имеется двоюродный брат — типаж `Copy`, и они отличаются друг от друга тем, что применение `Copy` означает *побитовое копирование* (т. е. буквальное копирование байтов объекта в новое место памяти), а применение `Clone` — явное копирование. В случае применения `Clone` для клонирования объекта вызывается метод `clone()`, а копирование в случае `Copy` происходит через присваивание (т. е. `let x = y;`). Типаж `Clone` обычно реализуется автоматически с помощью атрибута `#[derive(Clone)]`, но в особых случаях его можно реализовать самостоятельно.

С целью оказания разработчикам помощи в реализации клонирования при записи в `Rust` предоставляются три умных указателя:

- `Cow` — умный указатель на основе перечисления, предоставляющий удобную семантику;

- Rc и Arc — эти умные указатели с подсчетом ссылок обеспечивают семантику клонирования при записи с помощью метода make_mut(), причем Rc — однопоточная версия, а Arc — многопоточная.

Давайте сейчас рассмотрим сигнатуру типа Cow (листинг 5.7).

Листинг 5.7. Фрагмент определения Cow из стандартной библиотеки Rust

```
pub enum Cow<'a, B> where
    B: 'a + ToOwned + ?Sized, {
    Borrowed(&'a B),
    Owned(<B as ToOwned>::Owned),
}
```

Cow является перечислением, способным содержать либо заимствованный, либо собственный вариант. Для собственного варианта его поведение во многом напоминает Box, за исключением того, что в Cow данные не обязательно размещаются в куче. Если при использовании Cow нужны данные, размещенные в куче, надо в Cow воспользоваться типом Box или же вместо этого задействовать Rc или Arc. Функция клонирования при записи в Rust также не относится к функции на уровне языка и требует явного использования типажа Cow.

Обновим для демонстрации использования Cow пример односвязного списка так, чтобы структура данных стала неизменяемой. Сначала рассмотрим код листинга 5.8, который, за исключением добавления #[derive(Clone)], не слишком отличается от предыдущей версии.

Листинг 5.8. Код ListItem для односвязного списка с использованием Cow

```
# [derive(Clone)]           ← получение типажа Clone для обеих структур, поскольку Cow зависит
struct ListItem<T>        от поведения, предоставляемого типажом Clone
where
    T: Clone,
{
    data: Box<T>,          ←
    next: Option<Box<ListItem<T>>>, ←
}
impl<T> ListItem<T>
where
    T: Clone,
{
    fn new(data: T) -> Self {
        ListItem {
            data: Box::new(data),
            next: None,
        }
    }
}
```

```
fn next(&self) -> Option<&Self> {
    if let Some(next) = &self.next {
        Some(&*next)
    } else {
        None
    }
}
fn mut_tail(&mut self) -> &mut Self {
    if self.next.is_some() {
        self.next.as_mut().unwrap().mut_tail()
    } else {
        self
    }
}
fn data(&self) -> &T {
    self.data.as_ref()
}
}
```

Теперь обратимся к коду листинга 5.9, в котором показывается использование Cow в нашем списке.

Листинг 5.9. Листинг кода SinglyLinkedList для односвязного списка с использованием Cow

```
#[derive(Clone)]
struct SinglyLinkedList<'a, T>
where
    T: Clone,
{
    head: Cow<'a, ListItem<T>>, (1)
}

impl<T> ListItem<T>
where
    T: Clone,
{
    fn new(data: T) -> Self {
        ListItem {
            data: Box::new(data),
            next: None,
        }
    }
    fn next(&self) -> Option<&Self> {
        if let Some(next) = &self.next {
            Some(&*next)
        } else {
            None
        }
    }
}
```

```

        } else {
            None
        }
    }
fn mut_tail(&mut self) -> &mut Self {
    if self.next.is_some() {
        self.next.as_mut().unwrap().mut_tail()
    } else {
        self
    }
}
fn data(&self) -> &T {
    self.data.as_ref()
}
}

impl<'a, T> SinglyLinkedList<'a, T>
where
    T: Clone,
{
    fn new(data: T) -> Self {
        SinglyLinkedList {
            head: Cow::Owned(ListItem::new(data)),           (2)
        }
    }
    fn append(&self, data: T) -> Self {                  (3)
        let mut new_list = self.clone();
        let mut tail = new_list.head.to_mut().mut_tail();   (4)
        tail.next = Some(Box::new(ListItem::new(data)));
        new_list
    }
    fn head(&self) -> &ListItem<T> {
        &self.head
    }
}

```

Здесь:

- в поз. (1) — указатель `head` хранится в `Cow`. Для структуры нужно включить спецификатор времени жизни — чтобы компилятор знал, что время жизни структуры и параметра `head` ОДНО И ТО ЖЕ;
- в поз. (2) — инициализация списка с помощью указателя на `head`;
- в поз. (3) — сигнатура добавления изменилась: теперь уже в ней не требуется изменяемый `self`, а вместо него возвращается совершенно новый связанный список;
- в поз. (4) — вызов `to_mut()` запускает клонирование при записи, что происходит рекурсивно, путем получения изменяемой ссылки на заголовок `head`.

5.8. Пользовательские распределители

Иногда при выделении памяти возникают потребности в настройке поведения. К типичным примерам того, где они могут возникать, можно отнести:

- встроенные системы с сильно ограниченными ресурсами памяти или не имеющие операционной системы;
- приложения с критически важной высокой производительностью, которым требуется оптимизированное выделение памяти, включая пользовательские диспетчеры кучи — например, `jemalloc`¹ или `TCMalloc`²;
- приложения со строгими требованиями к безопасности, где могут, к примеру, возникнуть потребности в защите страниц памяти с помощью системных вызовов `mprotect()` и `mlock()`;
- некоторые интерфейсы библиотек или плагинов, которым во избежание утечек памяти при передаче данных могут потребоваться особые распределители памяти, — такое поведение довольно часто бывает при работе с преодолением языковых границ (т. е. при интеграции Rust с каким-либо языком, имеющим сборщик мусора);
- реализацию пользовательского диспетчера кучи, позволяющую отслеживать использование памяти из вашего приложения.

По умолчанию для выделения памяти в Rust используется стандартная системная реализация, которая в большинстве систем представлена функциями `malloc()` и `free()`, предоставляемыми системной библиотекой языка С. Именно такое поведение реализуется имеющимся в Rust *глобальным распределителем*, который может быть переопределен для всей программы Rust с помощью API `GlobalAlloc`, а для отдельных структур данных — с помощью пользовательских распределителей с API `Allocator`.

ПРИМЕЧАНИЕ

На момент подготовки книги API `Allocator`³ в Rust фигурировал только в ночных версиях. А в стабильной версии Rust можно по-прежнему применять API `GlobalAlloc`.

Даже если вам никогда не понадобится создавать собственный распределитель (большинству разработчиков пользовательский распределитель вряд ли пригодится), интерфейс распределителя все же стоит прочувствовать, чтобы лучше понять, как в Rust осуществляется управление памятью. На практике, за исключением особых упомянутых ранее случаев, беспокоиться о распределителях вам вряд ли когда-либо придется.

¹ См. <http://jemalloc.net/>.

² См. <https://github.com/google/tcmalloc>.

³ См. <https://github.com/rust-lang/rust/issues/32838>.

5.8.1. Создание пользовательского распределителя

Давайте посмотрим, как создается пользовательский распределитель, используемый с `Vec`. Наш распределитель будет просто вызывать функции `malloc()` и `free()`. Для начала рассмотрим типаж `Allocator` (листинг 5.10), определенный в стандартной библиотеке Rust⁴.

Листинг 5.10. Кода для типажа Allocator из стандартной библиотеки Rust

```
pub unsafe trait Allocator {
    fn allocate(&self, layout: Layout)
        -> Result<NonNull<[u8]>, AllocError>; (1)
    unsafe fn deallocate(&self, ptr: NonNull<u8>, layout: Layout); (1)
    fn allocate_zeroed(
        &self,
        layout: Layout
    ) -> Result<NonNull<[u8]>, AllocError> { ... } (2)
    unsafe fn grow(
        &self,
        ptr: NonNull<u8>,
        old_layout: Layout,
        new_layout: Layout
    ) -> Result<NonNull<[u8]>, AllocError> { ... } (2)
    unsafe fn grow_zeroed(
        &self,
        ptr: NonNull<u8>,
        old_layout: Layout,
        new_layout: Layout
    ) -> Result<NonNull<[u8]>, AllocError> { ... } (2)
    unsafe fn shrink(
        &self,
        ptr: NonNull<u8>,
        old_layout: Layout,
        new_layout: Layout
    ) -> Result<NonNull<[u8]>, AllocError> { ... } (2)
    fn by_ref(&self) -> &Self { ... } (2)
}
```

Здесь:

- в поз. (1) — необходимые методы;
- в поз. (2) — optionalные методы с предоставленными исходными реализациями.

Для реализации распределителя требуется предоставление только двух методов: `allocate()` и `deallocate()`. Они аналогичны методам `malloc()` и `free()`. Другие методы

⁴ См. <http://mng.bz/g7ze>.

предназначены для случаев, когда возникает желание дальнейшей оптимизации распределения. Эквивалентным для `allocate_zeroed()` в языке С является вызов `calloc()`, а для функций `grow` и `shrink` в нем бы использовался вызов `realloc()`.

ПРИМЕЧАНИЕ

В некоторых методах типажа `Allocator` можно заметить применение ключевого слова `unsafe`. Выделение и высвобождение памяти почти всегда подразумеваются в Rust выполнение небезопасных операций, поэтому эти методы помечены как небезопасные.

Для optionalных методов типажа `Allocator` в Rust предоставляются исходные реализации. В случаях разрастания (`growing`) или сокращения (`shrinking`) исходная реализация выделяет новую память, копируя все данные, после чего высвобождает старую память. Для выделения памяти под обнуленные (`zeroed`) данные исходная реализация вызывает функцию `allocate()` и записывает во все ячейки памяти нули. Начнем с создания распределителя, проходящего через глобальный распределитель (листинг 5.11).

Листинг 5.11. Код для проходящего распределителя

```
#![feature(allocator_api)]  
  
use std::alloc::{AllocError, Allocator, Global, Layout};  
use std::ptr::NonNull;  
  
pub struct PassThruAllocator;  
  
unsafe impl Allocator for PassThruAllocator {  
    fn allocate(&self, layout: Layout) -> Result<NonNull<[u8]>, AllocError> {  
        Global.allocate(layout)  
    }  
    unsafe fn deallocate(&self, ptr: NonNull<u8>, layout: Layout) {  
        Global.deallocate(ptr, layout)  
    }  
}
```

ПРИМЕЧАНИЕ

Примеры кода для API распределителя работают только вочных версиях, и для их компиляции или запуска необходимо либо воспользоваться командой:

`cargo +nightly ...`

либо переопределить цепочку инструментов в каталоге проекта с помощью команды:

`rustup override set nightly`

Приведенный в листинге 5.11 код создает сквозной распределитель, который просто вызывает базовую реализацию глобального распределителя с минимально необходимым кодом. Чтобы протестировать наш распределитель, запустите следующий код:

```
fn main() {
    let mut custom_alloc_vec: Vec<i32, _> =
        Vec::with_capacity_in(10, BasicAllocator); (1)
    for i in 0..10 {
        custom_alloc_vec.push(i as i32 + 1);
    }
    println!("custom_alloc_vec={:?}", custom_alloc_vec);
}
```

- Здесь в поз. (1) наш пользовательский распределитель создает вектор `Vec`, инициализируя его емкостью 10 элементов.

Выполнение этого кода ожидаемо приведет к следующему результату:

```
custom_alloc_vec=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

А теперь давайте изменим распределитель так, чтобы взамен того, что в нем используется, вызвать функции `malloc()` и `free()` неосредственно из библиотеки С. Для определения объема выделяемой памяти воспользуемся сведениями, получаемыми из метода `size()`, предоставляемого структурой `Layout` (листинг 5.12).

Листинг 5.12. Код для простого пользовательского распределителя с использованием `malloc()` и `free()`

```
#![feature(allocator_api)]

use std::alloc::{AllocError, Allocator, Layout};
use std::ptr::NonNull;

use libc::{free, malloc};

pub struct BasicAllocator;

unsafe impl Allocator for BasicAllocator { (1)
    fn allocate(
        &self,
        layout: Layout,
    ) -> Result<NonNull<[u8]>, AllocError> {
        unsafe {
            let ptr = malloc(layout.size() as libc::size_t); (2)
            let slice = std::slice::from_raw_parts_mut( (3)
                ptr as *mut u8,
                layout.size(),
            );
            Ok(NonNull::new_unchecked(slice)) (4)
        }
    }
    unsafe fn deallocate(&self, ptr: NonNull<u8>, _layout: Layout) {
        free(ptr.as_ptr() as *mut libc::c_void); (5)
    }
}
```

Здесь:

- в поз. (1) — в определении метода `allocate()` в типаже `Allocator` отсутствует ключевое слово `unsafe`, но нам всё равно нужно делать небезопасные вызовы. Поэтому указанный блок кода обернут в блок `unsafe {}`;
- в поз. (2) — вызов функции `malloc()` библиотеки С в предположении, что используется обычное стандартное выравнивание из структуры `Layout`;
- в поз. (3) — блок памяти возвращается как слайс, поэтому сначала выполняется преобразование простого указателя С в слайс Rust;
- в поз. (4) — создание и возвращение финального указателя на байтовый слайс;
- в поз. (5) — `deallocate()` по сути является обратным по функциональности методу `allocate()`, но этот метод для нас уже помечен как небезопасный. Указатель должен быть преобразован из его простого Rust-представления в С-указатель.

ПРИМЕЧАНИЕ

В структуре `Layout` содержатся свойства `size` и `align`, и оба они для переносимости должны пройти обработку. Свойство `size` указывает минимальное количество байтов для выделения, а свойство `align` — минимальное выравнивание байтов для блока в степенях числа два. Подробные сведения об этом можно найти в документации Rust по `Layout`⁵.

Обратите внимание на использование в коде листинга 5.12 ключевого слова `unsafe`: в методе `deallocate()` слово `unsafe` включено в качестве части сигнатуры самой функции, а `allocate()` требует использования `unsafe` внутри метода. В обоих случаях `unsafe` применяется в обязательном порядке, и обойтись без него нельзя, поскольку работа ведется с простыми указателями и памятью. Метод `deallocate()` помечен как небезопасный, поскольку, если он будет вызван с недопустимыми данными (например, с неверным указателем или неправильной компоновкой), его поведение будет неопределенным, а следовательно, и не безопасным. Если потребуется создать собственный распределитель, то отправная точка для этого может быть предоставлена приведенным здесь кодом, независимо от сути ваших потребностей в распределении.

5.8.2. Создание пользовательского распределителя для защищенной памяти

Давайте хотя бы в общих чертах рассмотрим более совершенный пример пользовательского распределителя памяти, проливающий свет на сценарий, в котором у вас появится желание воспользоваться API распределителя, имеющегося в Rust. В этом примере распределитель может применяться по частям к отдельным структурам данных, а не к программе в целом, что позволяет выполнять его тонкую настройку с целью повышения производительности. В крейте `dryoc`, уже не раз служившем в этой книге в качестве примера, для реализации функции защищенной памяти `dryoc`

⁵ См. <http://mng.bz/eEY9>.

используется типаж Allocator. Современными операционными системами разработчикам, создающим системы с критически важными аспектами безопасности, предоставляется несколько функций защиты памяти, и для реализации этих функций в Rust-программе приходится писать собственный код распределения памяти. В частности, в крейте `dryoc` в UNIX-подобных системах используются системные вызовы `mprotect()` и `mlock()`, а в Windows — системные вызовы `VirtualProtect()` и `VirtualLock()`. Эти системные вызовы предоставляют возможность блокировки и контроля доступа к определенным областям памяти внутри процесса как для внутреннего, так и для наружного его кода, что весьма важно для кода, управляющего конфиденциальными данными — например, секретными ключами.

В рамках реализации функций блокировки и защиты памяти она должна выделяться специальными платформенно-зависимыми функциями памяти (`posix_memalign()` в UNIX и `VirtualAlloc()` в Windows) таким образом, чтобы быть выровненной со страницами памяти, специфичными для платформы. Кроме того, в далее идущем коде выделяются два дополнительных блока памяти — до и после целевой ее области — подвергаемые блокировке в целях обеспечения дополнительной защиты от определенных типов атак на память. Эти области можно рассматривать в качестве своего рода бамперов — наподобие тех, что используются в автомобилях.

На рис. 5.2 показано, что в работе нашего пользовательского распределителя память будет выделена в куче. Активная область является подмножеством общей выделенной памяти, а подмножество, исключающее первую и последнюю страницы, возвращается распределителем в виде слайса.

Давайте рассмотрим фрагмент из примера кода этого распределителя (полный вариант примера содержится в исходном коде этой книги).

ПРИМЕЧАНИЕ

Напомню, что код примеров, включенных в книгу, доступен для загрузки с веб-сайта издательства Manning⁶, а также с GitHub⁷. Копия этого кода может быть также скачана вами на свой компьютер с помощью следующей Git-команды:

```
$ git clone https://github.com/brndnmthws/code-like-a-pro-in-rust-book
```

Сначала обратим внимание на код, приведенный в листинге 5.13.

Листинг 5.13. Фрагмент кода для `allocate()` из распределителя с выравниванием по страницам

```
fn allocate(&self, layout: Layout,
) -> Result<ptr::NonNull<[u8>, AllocError> {
    let pagesize = *PAGESIZE;
    let size = _page_round(layout.size(), pagesize) + 2 * pagesize;           (1)
    #[cfg(unix)]
    let out = {
        let mut out = ptr::null_mut();
        out
```

⁶ См. <https://www.manning.com/books/code-like-a-pro-in-rust>.

⁷ См. <https://github.com/brndnmthws/code-like-a-pro-in-rust-book>.

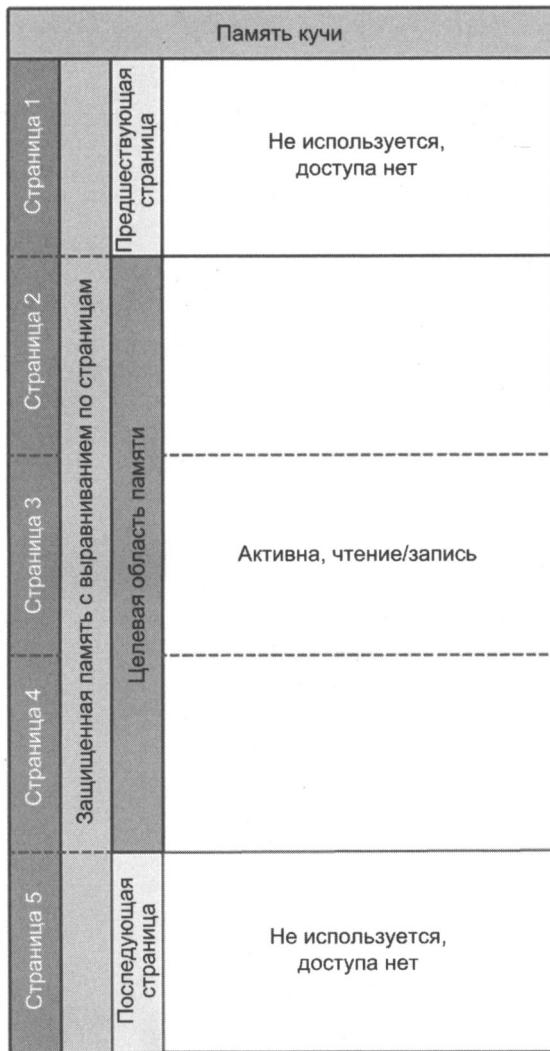


Рис. 5.2. Схема, показывающая защищенную структуру памяти с предшествующей и последующей страницами

```

let ret = unsafe {
    libc::posix_memalign(&mut out, pagesize as usize, size)      (2)
};

if ret != 0 {
    return Err(AllocError);
}

out
};

#[cfg(windows)]
let out = {
    use winapi::um::winnt::{MEM_COMMIT, MEM_RESERVE, PAGE_READWRITE};
}

```

```

unsafe {
    winapi::um::memoryapi::VirtualAlloc(
        ptr::null_mut(), size, MEM_COMMIT | MEM_RESERVE,
        PAGE_READWRITE,
    )
}
};

let fore_protected_region = unsafe {
    std::slice::from_raw_parts_mut(out as *mut u8, pagesize)
};

mprotect_noaccess(fore_protected_region) (4)
.map_err(|err| {
    eprintln!("mprotect error = {:?}", err)
})
.ok();

let aft_protected_region_offset = pagesize + _page_round(layout.size(), pagesize);
let aft_protected_region = unsafe {
    std::slice::from_raw_parts_mut(
        out.add(aft_protected_region_offset) as *mut u8,
        pagesize,
    )
};
mprotect_noaccess(aft_protected_region) (5)
.map_err(|err| {
    eprintln!("mprotect error = {:?}", err)
})
.ok();

let slice = unsafe {
    std::slice::from_raw_parts_mut(
        out.add(pagesize) as *mut u8,
        layout.size(),
    )
};
mprotect_readwrite(slice) (6)
.map_err(|err| {
    eprintln!("mprotect error = {:?}", err)
})
.ok();

unsafe { Ok(ptr::NonNull::new_unchecked(slice)) } (7)
}

```

Здесь:

- в поз. (1) — округление размера области памяти до ближайшей длины страницы с добавлением двух дополнительных страниц до и после целевой области памяти;
- в поз. (2) — выделение выровненной по страницам памяти в системах на базе POSIX;

- в поз. (3) — выделение выровненной по страницам памяти в системах на базе Windows;
- в поз. (4) — пометка страницы памяти перед новой областью, сделанной недоступной с целью предотвращения сканирования;
- в поз. (5) — пометка страницы памяти после новой области, сделанной недоступной с целью предотвращения сканирования;
- в поз. (6) — пометка новой области памяти, доступной для чтения/записи;
- в поз. (7) — возвращение нового указателя в виде слайса, состоящего из расположения в памяти и размера.

Теперь взглянем на реализацию `deallocate()`, показанную в листинге 5.14.

Листинг 5.14. Фрагмент кода функции `deallocate()` из распределителя с выравниванием по страницам

```
unsafe fn deallocate(&self, ptr: ptr::NonNull<u8>, layout: Layout) {  
    let pagesize = *PAGESIZE;  
    let ptr = ptr.as_ptr().offset(-(pagesize as isize));  
    // unlock the fore protected region  
    let fore_protected_region =  
        std::slice::from_raw_parts_mut(ptr as *mut u8, pagesize);  
    mprotect_readwrite(fore_protected_region) (1)  
        .map_err(|err| eprintln!("mprotect error = {:?}", err))  
        .ok();  
    // unlock the aft protected region  
    let aft_protected_region_offset =  
        pagesize + _page_round(layout.size(), pagesize);  
    let aft_protected_region = std::slice::from_raw_parts_mut(  
        ptr.add(aft_protected_region_offset) as *mut u8,  
        pagesize,  
    );  
    mprotect_readwrite(aft_protected_region) (2)  
        .map_err(|err| eprintln!("mprotect error = {:?}", err))  
        .ok();  
    #[cfg(unix)]  
    {  
        libc::free(ptr as *mut libc::c_void); (3)  
    }  
    #[cfg(windows)]  
    {  
        use winapi::shared::minwindef::LPVOID;  
        use winapi::um::memoryapi::VirtualFree;  
        use winapi::um::winnt::MEM_RELEASE;  
        VirtualFree(ptr as LPVOID, 0, MEM_RELEASE); (4)  
    }  
}
```

Здесь:

- в поз. (1) — возвращение предшествующей странице памяти исходной доступности по чтению/записи;
- в поз. (2) — возвращение последующей странице памяти исходной доступности по чтению/записи;
- в поз. (3) — высвобождение выровненной по странице памяти в системах на базе POSIX;
- в поз. (4) — высвобождение выровненной по странице памяти в системах на базе Windows.

Как уже отмечалось ранее, код листингов 5.13 и 5.14 основан на коде из крейта `dryoc`. Полный вариант этого кода, который может включать будущие усовершенствования, можно найти на GitHub⁸.

Использование атрибутов `cfg` и `cfg_attr`, а также макроса `cfg` для условной компиляции

Речь об атрибутах постоянно идет почти на всех страницах этой книги, но сейчас, судя по содержимому приведенного здесь примера пользовательского распределителя, стало время уделить атрибуту `cfg` более пристальное внимание. Тем, кому приходилось работать с такими языками, как C или C++, вероятно, знакомо использование макросов для включения или отключения кода во время компиляции (например: `#ifdef FLAG { ... } #endif`). Включение и отключение функций в ходе компиляции является весьма распространенным приемом, особенно для компилируемых языков, которым требуется доступ к функциям, специфичным для операционной системы (как в нашем примере с пользовательским распределителем). Эквивалентные функции Rust выглядят похоже, но ведут себя иначе, чем те, что могли вам встречаться в C и C++.

Rust для обработки условной компиляции кода предоставляет три встроенных инструмента:

- атрибут `cfg`, позволяющий условно включать прикрепленный код (т. е. элемент в следующей строке кода, будь то блок или инструкция);
- атрибут `cfg_attr`, который ведет себя как `cfg`, за исключением того, что он позволяет устанавливать новые атрибуты компилятора на основе уже существующих;
- макрос `cfg`, возвращающий в ходе компиляции значения `true` или `false`.

Чтобы проиллюстрировать их использование, рассмотрим следующий пример:

```
#[cfg(target_family = "unix")]
fn get_platform() -> String {
    "UNIX".into()
}

#[cfg(target_family = "windows")]
fn get_platform() -> String {
    "Windows".into()
}
```

⁸ См. <http://mng.bz/p1R5>.

```

fn main() {
    println!("Этот код работает на ОС семейства {}", get_platform());
    if cfg!(target_feature = "avx2") {
        println!("avx2 включена");
    } else {
        println!("avx2 не включена");
    }
    if cfg!(not(any(target_arch = "x86", target_arch = "x86_64"))) {
        println!("Этот код работает на процессоре не Intel");
    }
}

```

Здесь атрибут `cfg` применяется ко всему функциональному блоку для `get_platform()`, поэтому он фигурирует дважды. Макрос `cfg` служит для проверки факта включения функции цели `avx2` и использования архитектуры, отличной от `Intel`.

В примере пользовательского распределителя показано, что такие сокращенные предикаты конфигурации, как `unix` и `windows`, определяются самим компилятором. Иными словами, вместо того, чтобы написать: `#[cfg(target_family = "unix")]`, можно ограничиться записью: `#[cfg(unix)]`. Полный список значений конфигурации для вашего целевого центрального процессора можно получить, запустив на выполнение команду:

```
rustc --print=cfg -C target-cpu=native.
```

Предикаты также можно комбинировать с помощью `all()`, `any()` и `not()`. Функции `all()` и `any()` принимают список предикатов, а функция `not()` принимает только один предикат. К примеру, можно воспользоваться кодом:

```
#[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))
```

Полный список параметров конфигурации времени компиляции можно найти по адресу <http://mng.bz/OP7K>.

5.9. Кратко об умных указателях

В табл. 5.2 приведены краткие сведения об основных типах умных указателей и контейнерных типах памяти, способные помочь вам принять решение о том, чем именно воспользоваться. К этой таблице вы можете обращаться по мере накопления знаний о языке Rust, начиная экспериментировать с более совершенными способами управления памятью.

Таблица 5.2. Умные указатели и контейнеры Rust

Тип	Вид	Описание	Когда использовать?	Одно- или много-поточный
Box	Указатель	Умный указатель на память, размещаемую в куче	При каждой необходимости сохранения одного объекта в куче (а не в таком контейнере, как <code>Vec</code>)	Одно

Таблица 5.2 (окончание)

Тип	Вид	Описание	Когда использовать?	Одно- или много-поточный
Cow	Указатель	Умный указатель с клонированием при записи, который можно использовать с находящимися в собственности или с заимствованными данными	Когда нужны данные, размещенные в куче, с функцией клонирования при записи	Одно
Rc	Указатель	Умный указатель на память, размещаемую в куче, с подсчетом ссылок и обеспечением совместного владения	Когда необходимо совместное владение данными, размещенными в куче	Одно
Arc	Указатель	Умный указатель на память, размещаемую в куче, с атомарным подсчетом ссылок и обеспечением совместного владения	Когда необходимо совместное между потоками владение данными, размещенными в куче	Много
Cell	Контейнер	Контейнер памяти, обеспечивающий внутреннюю изменчивость с применением перемещения	Когда необходимо включить внутреннюю изменчивость данных внутри умного указателя с помощью перемещения	Одно
RefCell	Контейнер	Контейнер памяти, обеспечивающий внутреннюю изменчивость с применением ссылки	Когда необходимо включить внутреннюю изменчивость с помощью ссылок	Одно
Mutex	Контейнер	Примитив взаимного исключения, также допускающий внутреннюю изменчивость со ссылкой	Когда необходимо синхронизировать совместное использование данных между потоками	Много
RWLock	Контейнер	Примитив взаимного исключения, гарантирующий различие междучитывающим и записывающим кодом и обеспечивающий внутреннюю изменчивость со ссылкой	Когда нужна блокировка чтения/записи между потоками	Много

Резюме

- ❑ Box и Vec представляют собой методы для выделения памяти в куче. Vec предпочтительнее, когда нужен список элементов, а для одного элемента лучше использовать Box.
- ❑ Для обеспечения глубокого копирования структур данных в Rust можно воспользоваться типажом Clone.

- `Rc` и `Arc` предоставляют умные указатели с подсчетом ссылок с целью обеспечения совместного владения.
- `Cell` и `RefCell` предоставляют лазейку для решения задачи внутренней изменчивости, когда нужно изменить данные внутри неизменяемой структуры, но только для однопоточных приложений.
- `Mutex` и `RwLock` предоставляют примитивы синхронизации, которыми можно воспользоваться с `Arc` для включения внутренней изменчивости.
- API `Allocator` и `GlobalAlloc` предоставляют способ настройки поведения при выделении памяти в Rust.

Часть III

Корректность кода

Создавать качественные программные продукты сложно, и эта сложность проявляется во многих аспектах. Мы часто слышим о важности простоты кода, но меньше склонны задумываться о сестре простоты — правильности кода, его корректности. Написание простого кода — достойная цель, но даже самый красивый в мире и простой код, но некорректный, неправильный, — останется непригодным для использования. Нам свойственно скрывать сложность кода за абстракциями, но сложность есть везде и всегда, даже когда она скрыта, поэтому-то и нужны гарантии сохранения корректности.

Корректность кода бывает как качественной, так и количественной. Причем она зависит от того, насколько хорошо определен API-интерфейс и соответствует ли это определение его реализации. К примеру, вы можете создать функцию сложения, принимающую два параметра и возвращающую их сумму, но она также должна правильно обрабатывать крайние случаи: переполнение, наличие знаков перед числовыми значениями, неверные входные данные и т. п. Чтобы сумматор правильно обрабатывал все эти случаи, их нужно определять и учитывать. Неопределенное поведение — враг корректности.

В главах этой части мы рассмотрим стратегии тестирования, проводимого для получения гарантий корректности кода. Создание и применение тестов для кода позволяет выявить слабые места в спецификациях, найти неоднозначности, а также проверить правильность реализации задуманного.

6

Модульное тестирование

В этой главе:

- объяснение уникальности модульного тестирования в Rust;
- рассмотрение допустимых и недопустимых приемов в модульном тестировании, используемых функций и сред тестирования;
- обсуждение вопросов тестирования с параллельным кодом;
- создание модульных тестов, учитывающих реструктуризацию;
- описание инструментов, содействующих реструктуризации;
- оценка охвата кода тестированием;
- стратегии тестирования для работы с экосистемой Rust.

Модульное тестирование (называемое также юнит-тестированием) — это один из способов улучшения качества кода, позволяющий обнаружить его слабые места и гарантировать перед отправкой кода в производственную среду, что он соответствует предъявляемым к нему требованиям. В Rust встроена структура модульного тестирования, облегчающая разработку правильных программных средств, и в этой главе мы рассмотрим ряд функций, предоставляемых Rust, обсудим некоторые недостатки структуры модульного тестирования Rust и узнаем, как их преодолеть.

6.1. Чем примечательно тестирование в Rust?

Прежде чем перейти к подробному изучению имеющихся в Rust функций модульного тестирования, следует коснуться сути отличий Rust от других языков программирования и уточнить, как эти различия связаны с модульным тестированием. Читатели, имеющие опыт работы с Haskell или Scala, могут заметить, что в вопросах тестирования у Rust имеются некоторые сходства с этими языками. Но по сравнению с большинством других языков существенное отличие Rust от них заключается в том, что разновидности модульных тестов, встречающиеся в этих языках, в Rust просто не нужны.

Пояснить это утверждение можно тем обстоятельством, что в большинстве случаев после прохождения компиляции мы сразу получаем корректный код. Иными словами, компилятор Rust можно рассматривать как автоматический набор тестов, неизменно применяемый к коду. Однако такое положение справедливо лишь в отношении только определенных типов тестов, и в Rust существует множество способов от него отступить.

К наиболее распространенным способам потери части гарантий безопасности, имеющихся в Rust, можно отнести:

- использование ключевого слова `unsafe`;
- переход ошибок в ходе компиляции в ошибки в ходе выполнения программы.

Последняя ситуация может быть стать следствием различных действий, но наиболее распространенные из них связаны с использованием `Option` или `Result` без надлежащей обработки обоих случаев результата. В частности, такая ошибка может быть допущена при вызове для этих типов функции `unwrap()` без обработки случая сбоя. В некоторых случаях это делается преднамеренно — просто потому, что не хочется тратить время на обработку ошибок. Во избежание подобных проблем проще всего обрабатывать все возможные случаи и избегать вызова функций, вызывающих панику в ходе выполнения программы (например, функции `unwrap()`). К сожалению, способа проверки того, что в коде нет паники, в Rust *не предоставляемся*.

Функции и методы, входящие в стандартную библиотеку Rust и вызывающие панику при сбоях, обычно в документации специально отмечаются. Как правило, для любого вида программирования любые функции, выполняющие операции ввода/вывода или неопределенные операции, могут в любое время дать сбой (или вызвать панику), и все случаи сбоя должны обрабатываться соответствующим образом (если, конечно, корректным способом обработки сбоя не является собственно вызов паники).

Пояснение

Паника в Rust означает вызов ошибки и прерывание выполнения программы. Чтобы самому вызвать панику, можно воспользоваться макросом `panic!()`. А чтобы вызвать ошибку времени компиляции, можно воспользоваться макросом `compile_error!()`.

Как уже отмечалось ранее, ошибки программирования могут обнаруживаться Rust-компилятором и без проведения модульных тестов. Но компилятору не под силуправляться с логическими ошибками — Rust-компилятор, к примеру, может выявить ошибки деления на ноль, но не в состоянии обнаружить ошибочное использование деления вместо умножения.

Как правило, самый рациональный способ создания легко тестируемых программ, которые вряд ли будут содержать какие-либо ошибки, связан с разбиением кода на небольшие единицы вычисления (функции), отвечающие в основном следующим параметрам:

1. Функции по возможности не должны иметь состояния.
2. Функции, требующие наличия состояния, должны быть идемпотентными.

3. Функции по возможности должны быть детерминированными — результат выполнения функции должен быть всегда одинаковым для любого заданного набора входных данных.
4. Функции, способные на неудачное завершение, должны возвращать `Result`.
5. Функции, способные не возвращать значение (т. е. возвращать пустое значение), должны возвращать `Option`.

Для соблюдения пунктов 4 и 5 в Rust активно применяется оператор `?` (представляющий собой сокращение кода для раннего возврата с результатом ошибки, если результат не `Ok`), что позволяет сэкономить время на наборе кода. В главе 4 рассматривалось использование `Result` с типажом `From`, существенно упрощающее обработку выдаваемых кодом ошибок. Для любой создаваемой функции, возвращающей `Result`, в целях обработки ошибок с применением оператора `?` требуется всего лишь написать реализацию типажа `From`, подходящую для любых возможных ошибок внутри функции. Но следует учесть, что такое решение срабатывает только в обычных ситуациях и может не подойти в тех случаях, когда обработка ошибок носит для рассматриваемой функции специфический характер.

СОВЕТ

Если нужно вызвать панику при неожиданном результате, следует воспользоваться функцией `expect()`, которая в качестве аргумента принимает сообщение, объясняющее причину паники. Функция `expect()` — более безопасная альтернатива функции `unwrap()` и по большому счету ведет себя так же, как и функция `assert()`.

По действующему соглашению модульные тесты языка Rust хранятся в том же исходном файле, что и тестируемый код. То есть для любой заданной структуры, функции или метода соответствующий модульный тест обычно находится в том же самом исходном файле. При этом тесты обычно располагаются ближе к концу файла. Такое решение имеет полезный побочный эффект, помогающий сохранять код относительно компактным и решающим отдельно взятые проблемы. Попытка упаковать в один файл слишком много логики приводит к его разрастанию, особенно при использовании сложных тестов. Так что как только будет пройдена отметка в 1000 строк, настанет черед задуматься о реструктуризации.

В конечном счете основная часть всех этих советов относится не только к Rust — они применимы ко всем языкам программирования. Что же касается Rust, программа должна всегда обрабатывать возвращаемые значения и избегать, за исключением крайних случаев, использования функции `unwrap()`.

6.2. Встроенные функции тестирования

Rust предоставляет целый ряд базовых функций тестирования (см. табл. 6.1), хотя по сравнению с более совершенными средствами тестирования может оказаться, что встроенных функций вам будет недостаточно. Заметным отличием Rust от других языков программирования является включение основного инструментария и функций тестирования в сам язык без использования дополнительных библиотек или сред тестирования. Во многих языках тестированию отводится второстепенная

роль, и им для надлежащего тестирования кода требуются подключение дополнительных инструментов и библиотек. Функции, не предоставляемые Rust, обычно можно найти в крейтах, но в целом вам уже должно быть понятно, что Rust благодаря строгим языковым гарантиям существенно упрощает тестирование.

Таблица 6.1. Возможности тестирования, имеющиеся в Rust

Функция	Описание
Модульное тестирование	Rust и Cargo предоставляют модульное тестирование непосредственным образом — без использования дополнительных библиотек и с применением в исходных файлах модульных разделов <code>tests</code> , обязательно помечаемых атрибутом <code>#[cfg(test)]</code> . При этом тестовые функции должны быть помечены атрибутом <code>#[test]</code>
Интеграционное тестирование	Rust и Cargo предоставляют интеграционное тестирование, позволяющее тестировать библиотеки и приложения из их открытых интерфейсов. Такие тесты обычно создаются как отдельные приложения, обособленные от основного исходного кода
Тестирование документации	Примеры кода в документации исходного кода с применением <code>rustdoc</code> рассматриваются как модульные тесты, что весьма разумно повышает общее качество документации и тестирования
Интеграция Cargo	Модульные, интеграционные и документационные тесты работают с Cargo автоматически и не требуют никаких дополнительных действий, кроме определения самих тестов. Команда <code>cargo test</code> осуществляет фильтрацию, отображает ошибки утверждений и даже распараллеливает для вас проводимые тесты
Макросы утверждений	Rust предоставляет такие макросы утверждений, как <code>assert!()</code> и <code>assert_eq!()</code> , но они применяются не только для тестов (ими можно воспользоваться где угодно, поскольку это обычные макросы Rust). Однако при запуске модульных тестов Cargo, благодаря им, будет правильно обрабатывать сбои утверждений и предоставлять полезные выходные сообщения

Давайте для демонстрации имеющихся в Rust возможностей тестирования рассмотрим анатомию простой библиотеки с модульным тестом (листинг 6.1), код которой описывает обычный сумматор.

Листинг 6.1. Код базового модульного теста на Rust

```
pub fn add<T: std::ops::Add<Output = T>>(a: T, b: T) -> T {           (1)
    a + b                           (1)
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_add() {                  (2)
        assert_eq!(add(2, 2), 4);   (2)
    }
}
```

Здесь:

- в поз. (1) — функция сложения, принимающая два параметра одного типа и возвращающая результат того же типа. Тип `t` должен иметь реализованный для этого же типа выходных данных типаж `std::ops::Add`;
- в поз. (2) — в наших модульных тестах содержится код тестирования, а атрибут `#[cfg(test)]` сообщает компилятору, что это и есть модульный тест `mod`;
- в поз. (3) — это удобное сокращение, указывающее на включение всего из внешней области видимости нашего модуля. В тестах такое будет встречаться довольно часто;
- в поз. (4) — атрибут `#[test]` сообщает компилятору, что эта функция является модульным тестом.

К нашему удовольствию, тест этот будет пройден. Если запустить команду `cargo test`, на экран будет выведена информация, показанная в листинге 6.2.

Листинг 6.2. Успешный запуск теста

```
$ cargo test
Compiling unit-tests v0.1.0
(/Users/brenden/dev/code-like-a-pro-in-rust/code/c6/6.2/unit-tests)
Finished test [unoptimized + debuginfo] target(s) in 0.95s
    Running unitests (target/debug/deps/unit_tests-c06c761997d04f8f)

running 1 test
test tests::test_add ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests unit-tests

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

6.3. Среды тестирования

В модульном тестировании Rust нет вспомогательных компонентов (хелперов¹), создателей контекстов (фикстур²), тестовой обвязки или же параметризованных функций тестирования, имеющихся в других средах (фреймворках) модульного

¹ Хелперы (helpers) — многократно используемые в приложении фрагменты кода.

² Фикстуры (fixtures) — специальные функции, которые помогают подготовить окружение для тестов и убрать тестовые данные после их выполнения.

тестирования. Чтобы получить их в свое распоряжение, придется либо создавать собственный код, либо обратиться к библиотекам.

Неплохой интерфейс создания тестов для базового параметризованного тестирования предоставляет крейтом `parameterized`³. Еще одну реализацию параметризованного тестирования, характеризующуюся простотой, лаконичностью и удобством в использовании, предоставляет крейт `test-case`⁴. Для фикстур можно воспользоваться крейтом `rstest`⁵. А в крейте `assert2`⁶ содержатся утверждения (`assertions`), вдохновленные `Catch2` — популярной библиотекой C++.

Достойной подробного упоминания библиотекой также является крейт `proptest`⁷, который предоставляет Rust-реализацию `QuickCheck`⁸ — библиотеки Haskell, первоначально выпущенной в 1999 году, с которой вам, возможно, уже приходилось сталкиваться. Крейт `proptest` нельзя признать портированием `QuickCheck` на Rust один к одному — это, скорее, средство предоставления эквивалентной функциональности с некоторыми задокументированными специфичными для Rust отличиями⁹.

Тестирование свойств с помощью крейта `proptest` позволяет сэкономить массу времени за счет генерации случайных тестовых данных, проверки результатов и предоставления отчета с минимальным тестовым примером, необходимым для создания ошибки. Оно существенно экономит время, хотя и не служит обязательной заменой тестирования известных значений (например, при проверке соответствия спецификации).

ПРИМЕЧАНИЕ

Тестирование свойств не похоже на бесплатный обед, поскольку не обходится без компромисса, связанного с дополнительными циклами центрального процессора, затрачиваемыми на тестирование случайных значений, — в отличие от затрат на тестирование значений, отобранных самостоятельно или же заранее известных. Количество случайных значений для тестирования можно отрегулировать, но для данных с большим набором возможных значений тестировать каждый результат зачастую просто не имеет смысла.

Давайте вернемся к сумматору из предыдущего раздела (см. листинг 6.1), но на этот раз попробуем применить к нему `proptest`, предоставляющий тестовые данные для задействованной тестовой функции (листинг 6.3).

Листинг 6.3. Листинг кода сумматора с `proptest`

```
pub fn add<T: std::ops::Add<Output = T>>(a: T, b: T) -> T {
    a + b
}
```

³ См. <https://crates.io/crates/parameterized>.

⁴ См. <https://crates.io/crates/test-case>.

⁵ См. <https://crates.io/crates/rstest>.

⁶ См. (<https://crates.io/crates/assert2>).

⁷ См. <https://lib.rs/crates/proptest>.

⁸ См. <https://github.com/nick8325/quickcheck>.

⁹ См. <http://mng.bz/YRno>.

```

#[cfg(test)]
mod tests {
    use super::*;

    use proptest::prelude::*;

    proptest! {
        #[test]
        fn test_add(a: i64, b: i64) {           (1)
            assert_eq!(add(a, b), a + b);       (2)
        }
    }
}

```

Здесь:

- в поз. (1) — включение библиотеки proptest, в состав которой входит макрос proptest!;
- в поз. (2) — параметры нашей тестовой функции a и b будут предоставляться макросом proptest!;
- в поз. (3) — утверждение, что сумматор действительно возвращает значение a + b.

А теперь давайте снова запустим этот тест, но уже с применением proptest:

```
cargo test
Compiling proptest v0.1.0
(/Users/brenden/dev/code-like-a-pro-in-rust/code/c6/6.2/proptest)
Finished test [unoptimized + debuginfo] target(s) in 0.59s
Running unitests (target/debug/deps/proptest-db846addc2c2f40d)
```

```
running 1 test
test tests::test_add ... FAILED
```

failures:

```
---- tests::test_add stdout ----
# ... snip ...
thread 'tests::test_add' panicked at 'Test failed: attempt to add with overflow;
minimal failing input: a = -2452998745726535882, b = -6770373291128239927
successes: 1
local rejects: 0
global rejects: 0
', src/lib.rs:9:5
```

failures:

```
tests::test_add

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Увы! Похоже, наш сумматор не так уж и хорош. Оказывается, при определенных обстоятельствах он может и «взорваться» (в нашем случае операция сложения привела к переполнению, поскольку производится сложение двух целых чисел конечной длины со знаком). Такого рода сбой не ожидался и, наверное, остался бы незамеченным, если бы для `a` и `b` не была выполнена генерация случайных данных.

Арифметическое переполнение в Rust

Арифметические операции в Rust могут поначалу, особенно при тестировании, обескураживать. Дело в том, что код Rust, скомпилированный в режиме отладки (например, в коде тестов), по умолчанию используется проверенная арифметика. Когда тот же код компилируется в режиме выпуска, он будет использовать непроверенную арифметику. Получается, что мы можем иметь код, не работающий при запуске в режиме отладки, но нормально работающий (т. е. не выдающий ошибку и не приводящий к сбою программы) в рабочей среде.

Принятый в Rust подход может вносить путаницу из-за разницы поведения в зависимости от режима компиляции кода. Обоснование, принятое для этих ситуаций в Rust, утверждает, что тестовый код должен быть строже — чтобы отлавливать больше ошибок, но в целях совместимости код должен вести себя так же, как ведет себя большинство других программ во время выполнения.

Разработчики иногда принимают арифметическое переполнение как должное, потому что большинство языков эмулируют поведение таких языков, как C (которое обычно называют *переносимой* арифметикой — т. е. когда целое число переполняется, оно просто переносится). Для примитивных типов в Rust предоставляется ряд альтернативных арифметических функций, задокументированных в стандартной библиотеке для каждого типа. Например, для `i32` предоставляются `checked_add()`, `unchecked_add()`, `carrying_add()`, `wrapping_add()`, `overflowing_add()` и `saturating_add()`.

Чтобы эмулировать такое же поведение, как и в C, можно воспользоваться структурой `Wrapping`¹⁰ или же вызвать соответствующий метод для каждого типа и операции, — это поведение задокументировано в RFC 560¹¹.

Для исправления кода предыдущего теста есть несколько вариантов, но самым простым является явный перенос переполнения (т. е. последовать поведению C при целочисленном переполнении). Давайте внесем в этот код следующие обновления:

```
extern crate num_traits; (1)
use num_traits::ops::wrapping::WrappingAdd;
```

```
pub fn add<T: WrappingAdd<Output = T>>(a: T, b: T) -> T { (2)
    a.wrapping_add(&b)
}
```

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
```

¹⁰ См. <http://mng.bz/G97M>.

¹¹ См. <http://mng.bz/z01w>.

```
use proptest::prelude::*;

proptest! {
    #[test]
    fn test_add(a: i64, b: i64) {
        assert_eq!(add(a, b), a.wrapping_add(b));           (3)
    }
}
```

Здесь:

- в поз. (1) — ставка на использование крейта `num_traits`, предоставляющего типаж `WrappingAdd`;
- в поз. (2) — изменение привязки типажа `Add` на `WrappingAdd`;
- в поз. (3) — тест также требует обновления, чтобы в нем использовалась функция `wrapping_add()`.

Как можно видеть, в код добавлен крейт `num_traits` — небольшая библиотека, предоставляющая коду типаж `WrappingAdd`. Эквивалентный типаж в стандартной библиотеке Rust отсутствует, а без него создать такую вот универсальную функцию довольно сложно (типажи будут более подробно рассматриваться в главах 8 и 9).

Если запустить обновленный код, тест будет пройден, как, собственно, и ожидалось:

```
cargo test
Compiling wrapping-adder v0.1.0
(/Users/brenden/dev/code-like-a-pro-in-rust/code/c6/6.2/wrapping-adder)
Finished test [unoptimized + debuginfo] target(s) in 0.65s
Running unitests (target/debug/deps/wrapping_adder-5330c09f59045f6a)

running 1 test
test tests::test_add ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.01s

Running unitests (target/debug/deps/wrapping_adder-a198d5a6a64245d9)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests wrapping-adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

6.4. Компилятор лучше вас знает, что не нужно тестировать

Rust — статически типизированный язык, и это дает ему целый ряд важных преимуществ, особенно когда дело касается тестирования. Один из способов осознания разницы между статически и динамически типизированными языками, заключается в понимании того, что компилятор выполняет работу по ограничению набора возможных входных и выходных данных для любого оператора или блока кода, анализируя исходный код до его запуска. Возможный набор входных и выходных данных ограничивается спецификациями типов. То есть строка не может быть целым числом и наоборот. Компилятор проверяет, что типы соответствуют ожидаемым, а ссылки являются допустимыми. И вам уже не нужно беспокоиться о том, что строки и целые числа в ходе выполнения программы будут перепутаны, поскольку компилятор не позволяет такому произойти, тем самым освобождая вас (как разработчика) — при условии правильного использования типов — от беспокойства о целом ряде проблем.

В динамически типизированных языках ошибки типов — одна из наиболее распространенных разновидностей проблем. В интерпретируемых языках сочетание недопустимого синтаксиса и ошибок типов создает реальную возможность возникновения ошибок времени выполнения, которые трудно отлавливаются до отправки кода в производственную среду. Многие динамические языки были дооснащены инструментами статического анализа, но в них нет достаточной строгости или тщательности для выявления распространенных ошибок.

При тестировании в Rust не следует проверять то, что компилятор или средство проверки заимствований проверяет за нас. Например, нет надобности проверять, является ли целое число целым числом или строка строкой. Не требуется проверять и допустимость ссылок или то, что данные изменяются в двух разных потоках (что приводит к состоянию гонки).

Давайте проясним: сказанное не означает, что код вовсе не нужно тестировать, — просто основная часть тестируемого в Rust относится к логике, а не к проверке типов или использованию памяти. Правда, вам по-прежнему следует выполнять преобразования типов, которые могут привести к сбою, но такая обработка — вопрос логики, а эргономика Rust затрудняет обращение с ситуациями, которые могут дать ошибочный сбой.

Проведение надлежащего тестирования в Rust начинается с эффективного использования имеющейся в этом языке системы типов. Чрезмерное применение `Option`, `unwrap()` или небезопасного кода может привести к труднообнаруживаемым ошибкам, особенно если эти функциональные средства используются в качестве способа избежать обработки крайних случаев. Обязательной проверке следует подвергать операции с сохранением состояния и ввода/вывода и вести их обработку соответствующим образом (неплохо было бы ввести в привычку возвращение типа `Result` теми функциями или методами, которые выполняют ввод/вывод).

6.5. Работа с особыми случаями параллельного тестирования и глобального состояния

Когда Cargo запускает модульные тесты, то для ускорения тестирования он одновременно выполняет сразу нескольких тестов, используя параллельные потоки. В большинстве случаев все это работает открыто и не стоит ни малейшего беспокойства. Но иногда для проведения тестов приходится создавать глобальное состояние или фикстуры, что может потребовать совместно используемого состояния.

Для решения этой проблемы в Rust предоставляется несколько возможностей, и одна из них — создание для тестирования собственной функции `main()` (фактически переопределяя встроенную в Rust библиотеку тестирования `libtest`). Но от этого варианта, наверное, всё же больше проблем, чем пользы, поэтому вместо него я направлю вас к альтернативе — удобному крейту `lazy_static`¹².

СОВЕТ

При желании предоставить для тестирования собственную функцию `main()` можно отключить встроенную в Rust функцию `libtest`, указав в настройках цели `harness = false`.

Порядок действий можно найти в документации по `rustc`¹³ и `libtest`¹⁴.

Возможно, знакомство с крейтом `lazy_static` станет для вас приятным сюрпризом. Этот крейт существенно упрощает создание в Rust статических переменных. Создание глобального общего состояния в Rust сопряжено с некоторыми неудобствами, поскольку бывает так, что инициализировать статические структуры приходится в ходе выполнения программы. Но для такой инициализации можно создать статическую ссылку, обновляя ее при первом же обращении, что, собственно, и делает `lazy_static`.

Чтобы проиллюстрировать решение задачи с использованием глобального состояния, рассмотрим код листинга 6.4.

Листинг 6.4. Модульный тест с глобальным подсчетом

```
#[cfg(test)]
mod tests {
    static mut COUNT: i32 = 0;                                (1)

    #[test]
    fn test_count() {
        COUNT += 1;                                         (2)
    }
}
```

¹² См. https://crates.io/crates/lazy_static.

¹³ См. <https://doc.rust-lang.org/rustc/tests/index.html>.

¹⁴ См. <https://doc.rust-lang.org/test/index.html>.

Здесь:

- в поз. (1) — определение статической изменяемой переменной, используемой в качестве счетчика;
- в поз. (2) — увеличение значения счетчика в нашем teste.

Этот код не компилируется, вызывая выдачу следующей ошибки:

```
error[E0133]: use of mutable static is unsafe and requires unsafe function
or block
```

```
--> src/lib.rs:7:9
 |
7 |         COUNT += 1;
|             ^^^^^^^^^^ use of mutable static
|
= note: mutable statics can be mutated by multiple threads: aliasing
violations or data races will cause undefined behavior
```

(1)

()

- Здесь в поз. (1) (перевод) — примечание: изменяемые статические данные могут изменяться несколькими потоками: нарушения псевдонимов или гонка данных приведут к неопределенному поведению.

Компилятор четко отлавливает здесь ошибку. Если бы эквивалентный код создавался на C, то он бы без проблем скомпилировался и запустился (и, вероятно, долго работал бы... пока не перестал).

Для исправления кода есть несколько вариантов. У нас используется простой счетчик, поэтому можно вместо него воспользоваться атомарным целочисленным значением (обладающим потокобезопасностью). Казалось бы, всё просто, но так ли это? Давайте попробуем применить следующий код:

```
#[cfg(test)]
mod tests {
    use std::sync::atomic::{AtomicI32, Ordering};
    static mut COUNT: AtomicI32 = AtomicI32::new(0);
```

(1)

```
    #[test]
    fn test_count() {
        COUNT.fetch_add(1, Ordering::SeqCst);
    }
}
```

Здесь:

- в поз. (1) — использование атомарного целочисленного значения, предоставленного стандартной библиотекой Rust;
- в поз. (2) — выполнение операции выборки и добавления, увеличивающей атомарное целочисленное значение. Ordering::SeqCst сообщает компилятору, как именно следует синхронизировать операции, что отражено в соответствующей документации¹⁵.

¹⁵ См. <http://mng.bz/0lRp>.

При попытке скомпилировать обновленный тест выводится точно такая же ошибка (`use of mutable static — использование изменяемой статической переменной`). Так что же получается? Компилятор `rustc` очень строг — он жалуется на владение переменной `COUNT`, которая сама по себе не имеет реализации типажа `Send`. Чтобы эта реализация была, придется ввести использование `Arc`.

Rust-типажи `Send` и `Sync`

Для обработки общего состояния между потоками в Rust имеются два важных типажа: `Send` и `Sync`. Эти типажи используются компилятором для предоставления в Rust-программе гарантий безопасности потоков, и при работе с многопоточным кодом в Rust и общим состоянием с ними нужно как следует разобраться.

Эти типажи определяются так:

- `Send` отмечает те объекты, которые можно безопасно перемещать между потоками;
- `Sync` отмечает те объекты, которые можно безопасно совместно использовать сразу несколькими потоками.

Например, если требуется переместить переменную из одного потока в другой, ее надо обернуть во что-то, реализующее `Send`. Если нужно иметь общие ссылки на одну и ту же переменную сразу в нескольких потоках, эту переменную следует обернуть во что-то, реализующее `Sync`.

Эти типажи автоматически выводятся компилятором там, где это необходимо. Их непосредственной реализации не требуется, а для гарантии безопасности потоков предпочтительнее воспользоваться комбинацией с `Arc`, `Mutex` и `RwLock` (которые рассматривались в главе 5).

Теперь, когда стало понятно, что нужно воспользоваться `Arc`, давайте обновим код еще раз:

```
#[cfg(test)]
mod tests {
    use std::sync::atomic::{AtomicI32, Ordering};
    use std::sync::Arc;

    static COUNT: Arc<AtomicI32> = Arc::new(AtomicI32::new(0));

    #[test]
    fn test_count() {
        let count = Arc::clone(&COUNT);           (1)
        count.fetch_add(1, Ordering::SeqCst);
    }
}
```

- Здесь в поз. (1) — прежде чем воспользоваться `Arc`, для получения ссылки в контекст этого потока указанный тип следует клонировать.

При попытке скомпилировать этот код нас снова ждет разочарование из-за новой ошибки:

```
error[E0015]: calls in statics are limited to constant functions, tuple
structs and tuple variants
--> src/lib.rs:6:38
|
6 |     static COUNT: Arc<AtomicI32> = Arc::new(AtomicI32::new(0));
|           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

При таком результате, не будь этой книги, у вас могли бы и руки опуститься. А решение здесь весьма простое: `lazy_static`. Компилятор не позволяет создавать глобальные переменные, не являющиеся константами, поэтому для инициализации в ходе выполнения программы нужно либо создать собственный код, либо просто воспользоваться `lazy_static`. Давайте обновим код теста еще раз:

```
#[cfg(test)]
mod tests {
    use lazy_static::lazy_static;
    use std::sync::atomic::{AtomicI32, Ordering};
    use std::sync::Arc;
    lazy_static! {                                     (1)
        static ref COUNT: Arc<AtomicI32> = Arc::new(AtomicI32::new(0)); (2)
    }

    #[test]
    fn test_count() {
        let count = Arc::clone(&COUNT);
        count.fetch_add(1, Ordering::SeqCst);
    }
}
```

Здесь:

- в поз. (1) — макрос `lazy_static!` используется для обертывания наших определений статических переменных;
- в поз. (2) — при инициализации с помощью `lazy_static!` предоставляется блок кода, возвращающий инициализированный объект. В нашем случае всё действие помещается на одной строке, поэтому скобки `{ ... }` опускаются.

Всё! Теперь наш код компилируется и работает безопасно. Все тонкости инициализации данных в ходе выполнения программы берет на себя макрос `lazy_static!`. При первой же доступности переменной она проходит автоматическую инициализацию и оказывается в глобальном доступе. Чтобы разобраться в действиях `lazy_static`, давайте рассмотрим код, сгенерированный макросом с помощью команды `cargo expand` (рассмотренной в главе 3):

```
#[allow(missing_copy_implementations)]
#[allow(non_camel_case_types)]
#[allow(dead_code)]
struct COUNT {
    __private_field: (),
}
```

```

#[doc(hidden)]
static COUNT: COUNT = COUNT {
    __private_field: (),
};

impl ::lazy_static::__Deref for COUNT { (1)
    type Target = Arc<AtomicI32>;
    fn deref(&self) -> &Arc<AtomicI32> {
        #[inline(always)]
        fn __static_ref_initialize() -> Arc<AtomicI32> {
            Arc::new(AtomicI32::new(0)) (2)
        }
        #[inline(always)]
        fn __stability() -> &'static Arc<AtomicI32> {
            static LAZY: ::lazy_static::lazy::Lazy<Arc<AtomicI32>> =
                ::lazy_static::lazy::Lazy::INIT; (3)
            LAZY.get(__static_ref_initialize) (3)
        }
        __stability()
    }
}

impl ::lazy_static::LazyStatic for COUNT {
    fn initialize(lazy: &Self) {
        let _ = &**lazy;
    }
}

```

Здесь:

- в поз. (1) — в `lazy_static` реализован типаж `Deref` (в его коде `__Deref` является псевдонимом для `Deref` из базовой библиотеки);
- в поз. (2) — этот блок будет заменен предоставляемым нами кодом инициализации (в нашем случае всё умещается в одной строке);
- в поз. (3) — внутри кода `lazy_static` используется примитив `std::sync::Once` из базовой библиотеки Rust, инициализируемый на этом этапе.

При изучении исходного кода `lazy_static` становится понятно, что он основан на примитиве `std::sync::Once` (предоставляемом стандартной библиотекой). Добавленный на предыдущем шаге `Arc` можно отбросить, поскольку `lazy_static` предоставляет типаж `Send`. Окончательный результат при использовании `lazy_static` выглядит так:

```

#[cfg(test)]
mod tests {
    use lazy_static::lazy_static;
    use std::sync::atomic::{AtomicI32, Ordering};
    lazy_static! {
        static ref COUNT: AtomicI32 = AtomicI32::new(0);
    }
}

```

```
#[test]
fn test_count() {
    COUNT.fetch_add(1, Ordering::SeqCst);
}
}
```

И хотя `lazy_static` помогает решить задачу совместного использования глобального состояния, он не позволяет синхронизировать сами тесты. Например, если требуется гарантия, что тесты выполняются поочередно, придется либо реализовать для запуска своих тестов собственную функцию `main()`, либо настроить `libtest` на запуск тестов только в одном потоке, либо, как показано в следующем коде, синхронизировать выполнение тестов с помощью мьютекса:

```
#[cfg(test)]
mod tests {
    use lazy_static::lazy_static;
    use std::sync::Mutex;
    lazy_static! {
        static ref MUTEX: Mutex<i32> = Mutex::new(0);
    }
    #[test]
    fn first_test() {
        let _guard = MUTEX.lock().expect("couldn't acquire lock");
        println!("first test is running");
    }
    #[test]
    fn second_test() {
        let _guard = MUTEX.lock().expect("couldn't acquire lock");
        println!("second test is running");
    }
}
```

При многократном запуске этого кода с помощью команды:

```
cargo test --nocapture
```

можно будет заметить, что выводимая информация не всегда появляется в одном и том же порядке. Дело в том, что здесь невозможно гарантировать четкий порядок выполнения (`libtest` всё еще будет пытаться запустить эти тесты в параллельном режиме). Если нужно, чтобы тесты запускались в определенном порядке, следует либо воспользоваться барьерной или условной переменной, либо реализовать для запуска ваших тестов собственную функцию `main()`.

В заключение следует заметить, что модульные тесты не должны требовать синхронизации или общего состояния. Если возникнет необходимость в подобных действиях, то, возможно, вам следует подумать о реструктуризации своего проекта.

6.6. Размышления о реструктуризации

Одно из преимуществ модульного тестирования заключается в возможности выявления *регрессий* — т. е. изменений кода, нарушающих существующие функции или поведение программы до ее выпуска в производственную среду. И если ваши модульные тесты охватывают все функциональные требования, предъявляемые к программному средству, любое вносимое в него изменение, не соответствующее этим требованиям, приведет к сбою тестирования.

Реструктуризация (рефакторинг) кода, которую я определяю здесь как внесение в код изменений, не влияющих на поведение открытых интерфейсов программного средства, является обычной практикой и влечет за собой как преимущества, так и риски. Основной риск реструктуризации — внесение регрессий. Преимущества реструктуризации могут заключаться в сочетании более высокого качества кода, более быстрой компиляции и лучшей производительности.

При написании тестов с целью повышения качества нашего программного продукта можно воспользоваться различными стратегиями. Одна из них — тестирование открытых интерфейсов в дополнение к закрытым или внутренним интерфейсам. Лучше всего это срабатывает при достижении охвата кода, близкого к ста процентам (охват кода мы рассмотрим чуть позже).

В практической разработке программного продукта модульные тесты зачастую способны дать сбой, и их отладка, исправление и сопровождение могут отнимать у разработчика уйму времени. Поэтому сэкономить время и гарантировать такое же или более высокое качество программного продукта поможет, как ни странно, тестирование только того, что действительно нужно тестировать. Определить, что именно нужно протестировать, можно, проанализировав охват кода тестированием, выявив при этом то, что входит в функциональные требования к продукту, и убрав всё, что не должно быть охвачено (при условии, что такие изменения не приведут к сбою теста).

Чрезмерное тестирование отрицательно влияет на гибкость программного продукта и заставляет впустую тратить время на ненужную работу. При правильной же организации тестирования программных средств можно без каких-либо ограничений уверенно заниматься их реструктуризацией. Сочетание таких автоматизированных инструментов тестирования, как тестирование на основе свойств, fuzz-тестирование (которое будет рассмотрено в следующей главе) и анализ охвата кода, приводит к повышению качества и гибкости продукта, не требуя каких-либо суперспособностей.

6.7. Инструменты реструктуризации

Теперь, располагая несколькими отработанными тестами и понятным API, пора навести во всем этом порядок и приступить к улучшению внутреннего содержимого программного продукта. Реструктуризация может иметь разную сложность, но есть целый ряд инструментов, способных сгладить различия.

Прежде чем рассматривать востребованность тех или иных инструментов, процесс реструктуризации следует разбить на ее типы. К наиболее распространенным задачам реструктуризации следует отнести:

- *переформатирование* — более приемлемую расстановку пробелов и перестановку символов для повышения удобства чтения кода;
- *переименование* — изменение имен переменных, обозначений, констант;
- *перемещение* — перемещение кода из одного места в другое в дереве исходного кода, возможно, в другие крейты
- *переписывание* — полное переписывание разделов кода или алгоритмов.

6.7.1. Переформатирование

Для форматирования кода лучше всего воспользоваться `rustfmt` — инструментом, рассмотренным нами в главе 3. Самостоятельно заниматься переформатированием кода на языке Rust вряд ли имеет смысла, поскольку `rustfmt` можно легко настроить под все ваши предпочтения. Как это делается, объясняется в разд. «Поддержка аккуратности кода: инструмент `rustfmt`» главы 3. Использование `rustfmt` сводится к запуску по мере необходимости команды `cargo fmt` или же к непосредственной интеграции этого средства в рабочий редактор или в IDE-среду с помощью `rust-analyzer`.

6.7.2. Переименование

Реализация переименования в ряде сложных ситуаций может оказаться весьма непростой задачей. В редакторах программного кода для внесения изменений обычно имеется инструмент того или иного типа, выполняющий операции поиска и замены (эти же действия можно проделать из командной строки, воспользовавшись командой `sed` или каким-либо ее аналогом), но для масштабной реструктуризации этот способ не всегда оказывается самым лучшим. Весьма эффективным средством может быть применение регулярных выражений, но иногда всё же требуется что-то более контекстно-зависимое.

Так, в VS Code переименовать обозначение можно, просто выделив его курсором и нажав клавишу `<F2>` или выбрав пункт **Rename Symbol** из контекстного меню. Разумным подходом к переименованию обозначений может стать использование инструментального средства `rust-analyzer`, которое также предоставляет механизм структурного поиска и замены¹⁶. Как этим, так и другими его свойствами, можно воспользоваться непосредственно из рабочей IDE-среды или редактора кода.

Задействовать имеющуюся в `rust-analyzer` функцию структурного поиска и замены вы можете либо через его командную палитру, либо путем добавления комментария со строкой замены. По умолчанию замена применяется ко всему рабочему пространству, что упрощает процесс реструктуризации. Чтобы найти совпадения и выполнить замену выражений, типов, путей или элементов таким образом, чтобы не

¹⁶ См. <http://mng.bz/K97P>.

вносить синтаксические ошибки, rust-analyzer проанализирует синтаксическое дерево кода и выполнит замену только при допустимости результата. На рис. 6.1 показано, как в коде ранее показанного в этой главе примера защиты с помощью мьютекса выполнить замену `$m.lock()` на `Mutex::lock(&$m)`.

После применения замены будет получен результат, показанный на рис. 6.2. В этом примере вызовы `MUTEX.lock()` и `Mutex::lock(&MUTEX)` полностью эквивалентны, но кем-то может быть отдано предпочтение последней форме. Поскольку в приведенном

```
lib.rs — mutex-guard
src > lib.rs > () tests Enter request, for example 'Foo($a) ==> Foo::new($a)' (Press 'Enter' to confirm or 'Escape' to cancel)
1 #[cfg(test)]
2 mod tests {
3     use lazy_static::lazy_static;
4     use std::sync::Mutex;
5     lazy_static! {
6         static ref MUTEX: Mutex<i32> = mutex_guard::tests
7             static MUTEX: MUTEX = MUTEX {
8                 __private_field()
9             }
10        let _guard: MutexGuard<i32> = MUTEX.lock().expect(msg: "couldn't acquire lock");
11        println!("first test is running");
12    }
13    #[test]
14        ► Run Test | Debug
15        fn second_test() {
16            let _guard: MutexGuard<i32> = MUTEX.lock().expect(msg: "couldn't acquire lock");
17            println!("second test is running");
18        }
}
```

Рис. 6.1. Структурная замена с помощью rust-analyzer перед ее применением

```
lib.rs — mutex-guard
src > lib.rs > () tests > first_test
1 #[cfg(test)]
2 mod tests {
3     use lazy_static::lazy_static;
4     use std::sync::Mutex;
5     lazy_static! {
6         static ref MUTEX: Mutex<i32> = Mutex::new(0);
7     }
8     #[test]
9         ► Run Test | Debug
10        fn first_test() {
11            let _guard = Mutex.lock(&MUTEX).expect("couldn't acquire lock");
12            println!("first test is running");
13        }
14        #[test]
15        ► Run Test | Debug
16        fn second_test() {
17            let _guard = Mutex.lock(&MUTEX).expect("couldn't acquire lock");
18            println!("second test is running");
19        }
}
```

Рис. 6.2. Структурная замена с помощью rust-analyzer после ее применения

примере вместо выражения `std::sync::Mutex::lock()` указывалось только `Mutex::lock()`, структурный поиск и замена здесь могут считаться контекстными. Благодаря использованию в строке 4 примера инструкции `std::sync::Mutex` средству `rust-analyzer` становится известно, что я запрашиваю `std::sync::Mutex::lock()`.

6.7.3. Перемещение

На момент подготовки книги в инструменте `rust-analyzer` отсутствовала функция перемещения или переноса кода. Поэтому, если, к примеру, вам нужно перенести структуру и ее методы в другой файл или модуль, это придется сделать самостоятельно.

Обычно я не рекомендую обращаться к проектам, не связанным с сообществом Rust, но считаю, что всё же стоит упомянуть здесь Rust-плагин IntelliJ IDE, который предоставляет функцию переноса для перемещения кода¹⁷ (а также множество других функций, сопоставимых с возможностями `rust-analyzer`). Этот плагин приспособлен исключительно под IntelliJ и (насколько мне известно) не может использоваться с другими редакторами, хотя и имеет открытый исходный код.

6.7.4. Переписывание

Когда требуется переписать большие фрагменты кода или отдельные алгоритмы, отличным способом проверки, работает ли новый код так же, как старый, будет использование уже встречавшегося ранее в этой главе крейта `proptest`. Давайте для того, чтобы увидеть, как он работает, рассмотрим сначала показанные в листинге 6.5 код реализации алгоритма FizzBuzz и соответствующий тест.

Листинг 6.5. FizzBuzz с модульным тестом

```
fn fizzbuzz(n: i32) -> Vec<String> {
    let mut result = Vec::new();

    for i in 1..(n + 1) {
        if i % 3 == 0 && i % 5 == 0 {
            result.push("FizzBuzz".into());
        } else if i % 3 == 0 {
            result.push("Fizz".into());
        } else if i % 5 == 0 {
            result.push("Buzz".into());
        } else {
            result.push(i.to_string());
        }
    }

    result
}
```

¹⁷ См. <http://mng.bz/9QJx>.

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_fizzbuzz() {
        assert_eq!(fizzbuzz(3), vec!["1", "2", "Fizz"]);
        assert_eq!(fizzbuzz(5), vec!["1", "2", "Fizz", "4", "Buzz"]);
        assert_eq!(
            fizzbuzz(15),
            vec![
                "1", "2", "Fizz", "4", "Buzz", "Fizz", "7", "8", "Fizz",
                "Buzz", "11", "Fizz",
                "13", "14", "FizzBuzz"
            ]
        )
    }
}
```

Мы уверены, что этот алгоритм работает, но хотим написать другую версию кода. Поэтому нами создается показанная далее новая реализация, использующая `HashMap` (с тем же модульным тестом):

```
fn better_fizzbuzz(n: i32) -> Vec<String> {
    use std::collections::HashMap;
    let mappings = HashMap::from([(3, "Fizz"), (5, "Buzz")]);
    let mut result = vec![String::new(); n as usize];
    let mut keys: Vec<&i32> = mappings.keys().collect();
    keys.sort();
    for i in 0..n {
        for key in keys.iter() {
            if (i + 1) % *key == 0 {
                result[i as usize].push_str(mappings.get(key)
                    .expect("couldn't fetch mapping"));
            }
        }
        if result[i as usize].is_empty() {
            result[i as usize] = (i + 1).to_string();
        }
    }
}

result
```

Эта реализация немного сложнее, и, хотя она проходит все тестовые примеры, полной уверенности в ее работоспособности у нас нет. И тут в дело вступает `proptest` — мы можем просто сгенерировать с его помощью тестовые примеры и сравнить их с исходной реализацией:

```
use proptest::prelude::*;

proptest! {
    #[test]
    fn test_better_fizzbuzz_proptest(n in 1i32..10000) {           (1)
        assert_eq!(fizzbuzz(n), better_fizzbuzz(n))                  (2)
    }
}
```

Здесь:

- в поз. (1) — чтобы тест был не слишком продолжительным, значения для него ограничены диапазоном от 1 до 10 000;
- в поз. (2) — тут просто выполняется сравнение результатов работы наших старых и новых алгоритмов, которые, как ожидается, всегда будут одинаковыми.

6.8. Охват кода

Анализ охвата кода — весьма важный инструмент оценки качества и эффективности как тестов, так и самого кода. Отчеты об охвате кода могут автоматически генерироваться с помощью крейта `tarpaulin`¹⁸, поэтому установите его, выполнив команду:

```
cargo install cargo-tarpaulin
```

Воспользовавшись для примера кодом из предыдущего раздела, можно сгенерировать локальный отчет об охвате кода в формате HTML, запустив на выполнение команду:

```
cargo tarpaulin --out Html
```

результат работы которой приведен на рис. 6.3 и 6.4. В отчете показан 100-процентный охват для кода из файла `lib.rs`, что означает прохождение наших модульных тестов каждой строкой кода.

/Users/brenden/dev/code-like-a-pro-in-rust-book/c06/rewriting-fizzbuzz/src		Covered: 24 of 24 (100.00%)
Path	Coverage	
lib.rs	24 / 24 (100.00%)	

Рис. 6.3. Резюме отчета об охвате

Эти отчеты можно изучать как локально, так и в интеграции с системой CI/CD — для отслеживания охвата кода с течением времени. Такие сервисы, как `Codecov`¹⁹ и `Coveralls`²⁰, предлагают для проектов с открытым исходным кодом бесплатные

¹⁸ См. <https://crates.io/crates/cargo-tarpaulin>.

¹⁹ См. <https://about.codecov.io/>.

²⁰ См. <https://coveralls.io/>.

линии (например, для крейта dryoc используется сервис Codecov²¹). Эти сервисы отслеживают изменения охвата с течением времени, интегрируются с GitHub-запросами на извлечение и упрощают оценку прогресса.

```

fn fizzbuzz(n: i32) -> Vec<String> {
    let mut result = Vec::new();

    for i in 1..(n + 1) {
        if i % 3 == 0 && i % 5 == 0 {
            result.push("FizzBuzz".into());
        } else if i % 3 == 0 {
            result.push("Fizz".into());
        } else if i % 5 == 0 {
            result.push("Buzz".into());
        } else {
            result.push(i.to_string());
        }
    }

    result
}

fn better_fizzbuzz(n: i32) -> Vec<String> {
    use std::collections::HashMap;
    let mappings = HashMap::from([(3, "Fizz"), (5, "Buzz")]);
    let mut result = vec![String::new(); n as usize];
    let mut keys: Vec<&i32> = mappings.keys().collect();
    keys.sort();
    for i in 0..n {
        for key in keys.iter() {
            if (i + 1) % *key == 0 {
                result[i as usize].push_str(
                    mappings.get(key).expect("couldn't fetch mapping"),
                );
            }
        }
        if result[i as usize].is_empty() {
            result[i as usize] = (i + 1).to_string();
        }
    }
}

```

Рис. 6.4. Детальный отчет об охвате для кода из файла lib.rs

И последнее замечание об охвате кода — достижение 100-процентного охвата не должно становиться вашей конечной целью. Дело в том, что практически почти невозможно протестировать каждую строку кода. Данными об охвате можно воспользоваться, чтобы увидеть, улучшается ли со временем состояние дел или, по крайней мере, не ухудшается ли оно, но само численное значение является произвольным показателем, не имеющим качественного наполнения. Как сказал Вольтер: «Лучшее — враг хорошего».

²¹ См. <https://app.codecov.io/gh;brndnmthws/dryoc/>.

6.9. Работа с меняющейся экосистемой

Rust, как сам язык, так и его основные библиотеки, а также все доступные в экосистеме Rust крейты постоянно совершенствуются и обновляются. Всё время находится на переднем крае, конечно, здорово, но это влечет за собой целый ряд проблем. В частности, могут возникать серьезные сложности в поддержке как обратной, так и перспективной совместимости.

Модульное тестирование играет важную роль в непрерывном сопровождении, особенно при работе со смещающимися целями. Может, конечно, возникнуть соблазн просто закрепить версии зависимостей и избегать всяческих обновлений, но в долгосрочной перспективе это принесет больше вреда, чем пользы, тем более что зависимости могут переплетаться между собой. Даже несколько тестов способны сыграть весомую роль в обнаружении регрессий, особенно тех, что возникают в результате обновления сторонних библиотек или даже внесения неожиданных изменений в сам язык.

Резюме

- Присущие языку Rust сильная статическая типизация, строгий компилятор и средство проверки заимствований уменьшают нагрузку на модульное тестирование, поскольку отпадает нужда в проверке ошибок типов, возникающих в ходе выполнения программы, которым подвержены программы на других языках.
- При всей малочисленности встроенных функций тестирования в арсенале языка всё же имеются несколько крейтов для дополнения и автоматизации модульного тестирования.
- Благодаря `libtest` модульные тесты в Rust выполняются в параллельном режиме, обеспечивая неплохое ускорение в не самых сложных ситуациях, но код, чувствительный ко времени выполнения или требующий синхронизации, должен быть обработан особым образом.
- Тестирование свойств может привести к существенному сокращению времени и усилий, затрачиваемых на поддержку модульных тестов, и обеспечить более высокий уровень уверенности в состоятельности программных продуктов.
- Измерение и анализ охвата кода позволяют с течением времени дать количественную оценку эффективности модульных тестов.
- Модульные тесты помогают убедиться в том, что сторонние библиотеки и крейты функционируют после обновлений подобающим образом.



Интеграционное тестирование

В этой главе:

- описание различий между модульным и интеграционным тестированием;
- способы эффективного использования интеграционного тестирования;
- сравнение встроенных механизмов интеграционного тестирования Rust с внешним тестированием;
- изучение библиотек и инструментов для интеграционного тестирования;
- фаззинг создаваемых тестов.

В главе 6 было описано модульное тестирование программ на языке Rust. Здесь мы рассмотрим использование в программах на Rust интеграционного тестирования и сравним его с модульным тестированием. Оба вида тестирования — и модульное, и интеграционное — являются мощными стратегиями повышения качества разрабатываемых программных средств, но при частом их совместном использовании имеются в виду немного разные цели.

Иногда интеграционное тестирование, в зависимости от типа тестируемого программного средства, может даваться сложнее из-за более объемной работы по созданию тестовой обвязки и тестовых сценариев. С модульными тестами мы поэтому встречаемся чаще, чем с интеграционными, однако в Rust имеются все необходимые инструменты для написания эффективных интеграционных тестов, позволяющие экономить время при создании шаблонов и обвязок. В этой главе мы также рассмотрим библиотеки, содействующие ускоренному проведению интеграционного тестирования без лишней дополнительной работы.

7.1. Сравнение интеграционного и модульного тестирования

Интеграционное тестирование заключается в тестировании отдельно взятых модулей или групп из имеющихся в них открытых интерфейсов. Оно отличается от *модульного тестирования*, представляющего собой тестирование наименьших, поддающихся тестированию компонентов программных средств, куда иногда включаются и открытые интерфейсы. *Открытыми* считаются те интерфейсы, которые доступны внешним потребителям программного средства, — например, интерфейсы общедоступной библиотеки или же CLI-команды, когда речь идет о *приложениях командной строки*.

Общего у интеграционных и модульных тестов в Rust немного — учитывая, что, в отличие от модульных тестов, интеграционные тесты выполняются вне основного дерева исходного кода. Интеграционные тесты присутствуют в Rust в качестве отдельных крейтов — т. е. у них имеется доступ только к открыто экспортимуемым функциям и структурам.

Давайте создадим на скорую руку знакомую и любимую многими по курсу информатики простую реализацию алгоритма быстрой сортировки¹, код которой представлен в листинге 7.1.

Листинг 7.1. Быстрая сортировка, реализованная на языке Rust

```
pub fn quicksort<T: std::cmp::PartialOrd + Clone>(slice: &mut [T]) {           (1)
    if slice.len() < 2 {
        return;
    }
    let (left, right) = partition(slice);
    quicksort(left);
    quicksort(right);
}

fn partition<T: std::cmp::PartialOrd + Clone>(
    slice: &mut [T]
) -> (&mut [T], &mut [T]) {                                         (2)
    let pivot_value = slice[slice.len() - 1].clone();
    let mut pivot_index = 0;
    for i in 0..slice.len() {
        if slice[i] <= pivot_value {
            slice.swap(i, pivot_index);
            pivot_index += 1;
        }
    }
}
```

¹ См. <https://en.wikipedia.org/wiki/Quicksort>.

```
if pivot_index < slice.len() - 1 {  
    slice.swap(pivot_index, slice.len() - 1);  
}  
  
slice.split_at_mut(pivot_index - 1)  
}
```

Здесь:

- в поз. (1) — это наша открытая функция `quicksort()`, обозначенная ключевым словом `pub`
- в поз. (2) — наша закрытая функция `partition()`, недоступная за пределами локальной области видимости.

Интеграционные тесты находятся в каталоге `tests` на верхнем уровне исходного дерева и автоматически обнаруживаются Cargo. Пример структуры каталога для небольшой библиотеки (в файле `src/lib.rs`) и одного интеграционного теста имеет следующий вид:

```
$ tree  
.  
└── Cargo.lock  
└── Cargo.toml  
└── src  
    └── lib.rs          ← содержит исходный код нашей библиотеки  
└── tests            ← интеграционные тесты находятся в каталоге tests  
    └── quicksort.rs    ← файл quicksort.rs содержит наши интеграционные тесты
```

2 directories, 4 files

Тестовые функции отмечены атрибутом `#[test]` и запускаются средством Cargo автоматически. При этом вы можете воспользоваться автоматически предоставляемой функцией `main()` из `libtest` или предоставить такую же собственную функцию, как это делалось в модульных тестах. Cargo обрабатывает интеграционные тесты как отдельные крейты. В подкаталогах каталога тестов может быть сразу несколько отдельных наборов крейтов, в каждом из которых будет содержаться свой собственный отдельный интеграционный тест.

Здесь, как и в модульных тестах, для проверки результатов обычно используются макросы утверждений (`assert!()` и `assert_eq!()`). Практический пример кода интеграционного теста показан в листинге 7.2.

Листинг 7.2. Пример кода интеграционного теста для проверки реализации быстрой сортировки

```
use quicksort::quicksort;  
  
#[test]  
fn test_quicksort() {  
    let mut values = vec![12, 1, 5, 0, 6, 2];  
    quicksort::quicksort(&mut values);  
    assert_eq!(values, [0, 1, 2, 5, 6, 12]);  
}
```

```

quicksort(&mut values);
assert_eq!(values, vec![0, 1, 2, 5, 6, 12]);

let mut values = vec![1, 13, 5, 10, 6, 2, 0];
quicksort(&mut values);
assert_eq!(values, vec![0, 1, 2, 5, 6, 10, 13]);
}

```

Ведь правда же, сильно похоже на модульные тесты? Разница в примере почти полностью семантическая — как можно видеть, фактически в этот пример включены модульные тесты, которые выглядят примерно так же, как в листинге 7.3.

Листинг 7.3. Пример кода модульных тестов для проверки реализации быстрой сортировки

```

#[cfg(test)]
mod tests {
    use crate::partition, quicksort;

    #[test]
    fn test_partition() {
        let mut values = vec![0, 1, 2, 3];
        assert_eq!(
            partition(&mut values),
            (vec![0, 1, 2].as_mut_slice(), vec![3].as_mut_slice())
        );
    }

    let mut values = vec![0, 1, 2, 4, 3];
    assert_eq!(
        partition(&mut values),
        (vec![0, 1, 2].as_mut_slice(), vec![3, 4].as_mut_slice())
    );
}

#[test]
fn test_quicksort() {
    let mut values = vec![1, 5, 0, 6, 2];
    quicksort(&mut values);
    assert_eq!(values, vec![0, 1, 2, 5, 6]);

    let mut values = vec![1, 5, 10, 6, 2, 0];
    quicksort(&mut values);
    assert_eq!(values, vec![0, 1, 2, 5, 6, 10]);
}
}

```

Единственное отличие кода этого модульного теста от показанного ранее интеграционного (см. листинг 7.2) заключается в том, что в модульных тестах, кроме всего

прочего, тестируется функция `partition()`, не являющаяся открытой. Тогда может быть и вовсе *не стоит* утруждать себя созданием интеграционных тестов? Нет, всё же стоит. Почему? Да потому что нами создается библиотека с открытым интерфейсом, которая должна быть протестирована, поскольку она предназначена для внешнего пользования. Интеграционные тесты пребывают *вне* тестируемой библиотеки (или приложения), благодаря чему им видны только открытые (и внешние) интерфейсы. Это принуждает к созданию тестов таким образом, будто наша библиотека или приложение являются продуктом для последующих пользователей. Интеграционное тестирование помогает убедиться в том, что открытый API-интерфейс работает с позиции внешних пользователей именно так, как это и было задумано.

7.2. Стратегии интеграционного тестирования

Не так давно популярностью пользовался подход, называемый *разработкой через тестирование* (Test Driven Development, TDD). Он основан на идее написания тестов еще до создания кода самого программного средства. Теоретически TDD был призван ускорить написание качественного кода за счет предшествовавшего ему создания тестов. Сейчас TDD, похоже, уже не в моде, но он все же раскрывает ряд понятий, имеющих особое значение для интеграционного тестирования.

TDD учит тому, что проектирование API не менее важно тестирования. Эргономика программного средства имеет значение, независимо от вида создаваемого продукта — будь то библиотека, приложение командной строки или же веб-, настольное или мобильное приложение. При написании интеграционных тестов проявляется *пользовательское восприятие* (User Experience, UX), заставляющее думать о том, как применяется программный продукт с позиции человека, который им пользуется.

Интеграционное и модульное тестирование не являются взаимоисключающими и могут там, где это уместно, дополнять друг друга. При этом интеграционные тесты не должны совпадать по написанию с модульными тестами, поскольку и в тех и в других тестируются разные сущности. Готовясь к созданию интеграционных тестов, следует брать в расчет не только корректность алгоритма или реализуемой им логики, — интеграционные тесты необходимо воспринимать и как способ проверки работоспособности кода, и как метод тестирования пользовательского восприятия программного продукта. Примеров удачного и неудачного проектирования программных продуктов великое множество, и процесс подготовки интеграционных тестов для вашего собственного программного продукта, дающий вам возможность попробовать на вкус принятые проектировочные решения, поможет вам не ступить на неверный путь. Тем более что при создании программного продукта можно получить туннельное зрение и потерять из виду общую картину, а интеграционные тесты по определению предоставляют целостный взгляд на создаваемый код.

Лично мне не раз приходилось сталкивался с проблемой туннельного зрения. Например, при написании скрипта `dryoc` я немножко увлекся созданием ряда дополнитель-

тельных функций, и только при попытке создания интеграционных тестов понял, что на тот момент спроектировал крайне неудачный интерфейс. Чтобы библиотека стала более простой в использовании, мне пришлось существенно переработать свое проектное решение.

Теперь вопрос: следует ли придерживаться положений TDD, создавая интеграционные тесты *до* написания библиотеки или приложения? Я не следую этой практике, но и не считаю ее непригодной, поэтому рекомендую проверить ее на деле и определить, работает она для вас или нет. Считаю, что интеграционные тесты следует писать в любом случае, чтобы прочувствовать все эмоции конечных пользователей вашего продукта. В каком порядке будут создаваться тесты, решать вам, — в любом случае нужно проявлять гибкость в проектировании и безжалостно реструктурировать код.

Вспоминается высказывание плодовитого архитектора и изобретателя:

Когда я работаю над решением какой-нибудь задачи, то никогда не задумываюсь о красоте, но когда я завершаю работу и вижу, что решение некрасиво, я знаю, что оно неправильное.

R. Бакминстер Фуллер (R. Buckminster Fuller)

Приведенный в предыдущем разделе пример быстрой сортировки иллюстрирует наши возможности совершенствования интерфейса, что проявляется со всей очевидностью при написании тестов для этой библиотеки. На текущий момент у нас имеется автономная функция `quicksort()`, принимающая в качестве входных данных слайс. Всё вроде бы хорошо, но код можно сделать более подходящим для Rust, создав типаж (более подробно типажи рассматриваются в главе 8) и предоставив его реализацию, код которой приведен в листинге 7.4.

Листинг 7.4. Код типажа Quicksort

```
pub trait Quicksort {                                     (1)
    fn quicksort(&mut self) {}                         (2)
}

impl<T: std::cmp::PartialOrd + Clone> Quicksort for [T] { (3)
    fn quicksort(&mut self) {
        quicksort(self);                            (4)
    }
}
```

Здесь:

- в поз. (1) — дается определение нашему открытому типажу `quicksort`;
- в поз. (2) — во избежание конфликтов с существующим для `Vec` и слайсов методом `sort()` вместо него будет использоваться `quicksort()`;
- в поз. (3) — дается определение базовой реализации нашего типажа, работающей для любого типа слайса, в котором будут реализованы типажи `PartialOrd` и `Clone`;

- в поз. (4) — просто выполняется непосредственный вызов нашей реализации quicksort.

Теперь, как показано в листинге 7.5, наши тесты можно обновить.

Листинг 7.5. Код для интеграционного теста типажа quicksort

```
#[test]
fn test_quicksort_trait() {
    use quicksort_trait::Quicksort; (1)

    let mut values = vec![12, 1, 5, 0, 6, 2];
    values.quicksort(); (2)
    assert_eq!(values, vec![0, 1, 2, 5, 6, 12]);

    let mut values = vec![1, 13, 5, 10, 6, 2, 0];
    values.quicksort(); (2)
    assert_eq!(values, vec![0, 1, 2, 5, 6, 10, 13]);
}
```

Здесь:

- в поз. (1) — импортировать нужно только типаж quicksort;
- в поз. (2) — вместо `quicksort(&mut values)` можно просто написать `values.quicksort()`.

Совершенно иным этот код не выглядит, и по большей части в нем для прояснения сути написанного используется простое синтаксическое сокращение. Вызов `arr.quicksort()` вместо `quicksort(&mut arr)` выглядит понятнее и на четыре символа экономнее, поскольку не требует явного указания изменяемого заимствования с помощью `&mut`.

7.3. Сравнение встроенного и внешнего интеграционного тестирования

Наверное, большинству разработчиков вполне хватит встроенного интеграционного тестирования, но оно всё же не панацея. Временами могут понадобиться и внешние инструменты интеграционного тестирования. Например, тестирование HTTP-сервиса в Rust может быть гораздо лучше выполнено с помощью таких простых (и практически вездесущих) инструментов, как curl² или HTTPie³. Конкретно с Rust они не связаны, поскольку являются типовыми инструментами, работающими не на уровне языка, а на уровне системы.

Быстрый поиск в Интернете позволит убедиться в существовании великого множества инструментов тестирования программных средств, особенно для HTTP-

² См. <https://curl.se/>.

³ См. <https://github.com/httpie/httpie>.

сервисов. Если перед вами не стоит задача по созданию собственной среды тестирования, то практически всегда лучше воспользоваться уже существующими инструментами, чем в очередной раз изобретать велосипед.

Написание на Rust интеграционных тестов для созданных на этом же языке приложений командной строки — далеко не лучший подход. Rust разработан для обеспечения высоких уровней безопасности и производительности, а тестовые обвязки обычно не нуждаются ни в безопасности, ни в быстродействии, — они просто должны быть подходящими. В большинстве случаев намного проще писать интеграционные тесты не в виде программы на Rust, а в виде Bash-, Ruby- или Python-сценариев.

Делать все на Rust, конечно, здорово, однако, в зависимости от сложности задачи, разумнее взвесить, во что обойдется по времени написание интеграционных тестов на Rust. Для непрятязательных приложений динамические языки сценариев предлагают множество преимуществ, позволяя, как правило, сделать всё гораздо быстрее и с весьма скромными усилиями даже тем, кто считает себя экспертами по Rust.

И всё же использование для интеграционных тестов *исключительно* языка Rust имеет одно весьма существенное преимущество — такие тесты можно будет запускать на любой платформе, поддерживаемой Rust, не испытывая необходимости в привлечении внешних инструментов и обходясь только набором инструментов самого Rust. От этого можно получить вполне определенный выигрыш, особенно в каких-нибудь ограниченных средах. Кроме того, если Rust для вас является самым продуктивным языком, то нет причин им не воспользоваться.

7.4. Библиотеки и инструменты для проведения интеграционного тестирования

Большинство инструментов и библиотек, используемых для модульного тестирования, также применимы и для выполнения интеграционных тестов. Тем не менее имеются крейты, способные существенно упростить проведение интеграционного тестирования, которые мы здесь и рассмотрим.

7.4.1. Использование `assert_cmd` для тестирования CLI-приложений

Давайте для тестирования приложений командной строки (Command Line Interface, CLI) применим крейт `assert_cmd`⁴, упрощающий запуск команд и проверку результатов их запуска. Чтобы продемонстрировать его применение, мы создадим интерфейс командной строки (листинг 7.6), предназначенный для нашей реализации `quicksort`, выполняющей сортировку целых чисел из состава CLI-аргументов.

⁴ См. https://crates.io/crates/assert_cmd.

Листинг 7.6. CLI-приложение, использующее quicksort

```
use std::env;

fn main() {
    use quicksort_proptest::Quicksort;

    let mut values: Vec<i64> = env::args()
        .skip(1)                                     (1)
        .map(|s| s.parse::<i64>().expect(&format!("{}: bad input: {}", s)))
        .collect();                                 (2)

    values.quicksort();

    println!("{} values: {}", values.len(), values);
}
```

Здесь:

- в поз. (1) — считывание аргументов командной строки, с пропуском первого аргумента, которым всегда является название программы;
- в поз. (2) — парсинг каждого значения (строки) в i64;
- в поз. (3) — сбор значений в Vec.

Работоспособность кода можем проверить, запустив на выполнение команду:

cargo run 5 4 3 2 1

которая выведет на экран:

[1, 2, 3, 4, 5]

А теперь воспользуемся assert_cmd и напишем несколько тестов, приведенных в листинге 7.7.

Листинг 7.7. Интеграционные тесты Quicksort CLI с использованием assert_cmd

```
use assert_cmd::Command;

#[test]
fn test_no_args() -> Result<(), Box <dyn std::error::Error>> {           (1)
    let mut cmd = Command::cargo_bin("quicksort-cli")?;
    cmd.assert().success().stdout("[]\n");

    Ok(())
}

#[test]
fn test_cli_well_known() -> Result<(), Box <dyn std::error::Error>> {      (3)
    let mut cmd = Command::cargo_bin("quicksort-cli")?;

    Ok([1, 2, 3, 4, 5])
}
```

```

cmd.args(&["14", "52", "1", "-195", "1582"])
    .assert()
    .success()
    .stdout("[-195, 1, 14, 52, 1582]\n");

Ok(())
}

```

(4)

Здесь:

- в поз. (1) — тестовыми функциями возвращается `Result`, что позволяет нам воспользоваться оператором `?`;
- в поз. (2) — в конце теста просто возвращается `Ok()`. `()` — это единичный тип, используемый в качестве заполнителя и не имеющий значения. Он может рассматриваться как эквивалент кортежа с нулевыми элементами;
- в поз. (3) — тестовыми функциями возвращается `Result`, что позволяет нам воспользоваться оператором `?`;
- в поз. (4) — в конце теста просто возвращается `Ok()`. `()` — это единичный тип, используемый в качестве заполнителя и не имеющий значения. Он может рассматриваться как эквивалент кортежа с нулевыми элементами;

С тестами все в порядке, но для тестирования с известными значениями их можно улучшить. Вместо жесткой забивки в исходный код для тестирования известных значений, программным способом можно создать несколько простых файловых фикстур.

Сначала создадим в файловой системе простую структуру каталогов для хранения наших тестовых фикстур — эта структура состоит из пронумерованных папок с файлами для аргументов (`args`) и ожидаемого результата (`expected`):

```
$ tree tests/fixtures
tests/fixtures
├── 1
│   ├── args
│   └── expected
├── 2
│   ├── args
│   └── expected
└── 3
    ├── args
    └── expected
```

3 directories, 6 files

После чего создадим тест (листинг 7.8), который проходит по каждому каталогу дерева и считывает аргументы и ожидаемый результат, а затем запустим наш тест и в завершение проверим результат.

Листинг 7.8. Интеграционные тесты Quicksort CLI с файловыми фикстурами

```

#[test]
fn test_cli_fixtures() -> Result<(), Box<dyn std::error::Error>> {
    use std::fs;
    let paths = fs::read_dir("tests/fixtures")?;                                (1)

    for fixture in paths {                                                       (2)
        let mut path = fixture?.path();
        path.push("args");                                                       (3)
        let args: Vec<String> = fs::read_to_string(&path)?;                      (4)
            .trim()                                                               (4)
            .split(' ')
            .map(str::to_owned)                                                    (4)
            .collect();                                                          (4)
        path.pop();                                                               (5)
        path.push("expected");                                                   (6)
        let expected = fs::read_to_string(&path)?;                                 (7)

        let mut cmd = Command::cargo_bin("quicksort-cli")?;                      (8)
        cmd.args(args).assert().success().stdout(expected);                      (8)
    }

    Ok(())
}

```

Здесь:

- в поз. (1) — использование в нашем крейте листинга каталогов, расположенных в каталоге `tests/fixtures`;
- в поз. (2) — последовательный перебор каждого листинга в каталоге;
- в поз. (3) — помещение имени "args" в наш буфер `path`;
- в поз. (4) — считывание содержимого файла `args` в строку и преобразование строки в `Vec`. Метод `trim()` удаляет из файла `args` завершающий символ новой строки, `split(' ')` разбивает содержимое по пробелам, `map(str::to_owned)` преобразует `&str` в принадлежащую `String`, и наконец, `collect()` собирает результаты в `Vec`;
- в поз. (5) — удаление аргументов из буфера `path`;
- в поз. (6) — помещение "expected" в буфер `path`;
- в поз. (7) — считывание ожидаемых значений из файла в строку;
- в поз. (8) — запуск `quicksort CLI`, передача аргументов и проверка ожидаемых результатов.

7.4.2. Использование с интеграционными тестами крейта *proptest*

Далее, как показано в листинге 7.9, для повышения надежности наших тестов к созданной реализации интеграционного теста быстрой сортировки можно добавить крейт *proptest* (рассмотренный в предыдущей главе).

Листинг 7.9. Интеграционный тест quicksort на основе proptest

```
use proptest::prelude::*;

proptest! {
    #[test]
    fn test_quicksort_proptest(
        vec in prop::collection::vec(prop::num::i64::ANY, 0..1000)
    ) {
        use quicksort_proptest::Quicksort;

        let mut vec_sorted = vec.clone();
        vec_sorted.sort();                                     (1)

        let mut vec_quicksorted = vec.clone();
        vec_quicksorted.quicksort();                         (2)

        assert_eq!(vec_quicksorted, vec_sorted);
    }
}
```

Здесь:

- в поз. (1) — предоставление `Vec` случайных целых чисел длиной до 1000 с применением `prop::collection::vec`;
- в поз. (2) — клонирование с последующей сортировкой случайных значений (с использованием встроенного метода сортировки), чтобы воспользоваться ими в качестве средства контроля;
- в поз. (3) — клонирование и сортировка случайных значений с использованием нашей реализации `quicksort`.

Следует заметить, что тестирование с применением инструментов, автоматически генерирующих тестовые данные (к примеру, такого как *proptest*), в том случае, когда ваши тесты имеют внешние побочные действия, — например, выполнение сетевых запросов или запись во внешнюю базу данных, может иметь непредвиденные последствия. Свои тесты нужно стараться разрабатывать с учетом этого обстоятельства: либо подстраивая нужным образом и возвращая в прежнее состояние всю среду до и после каждого теста, либо предоставляя какой-либо другой способ вернуться к известному нужному состоянию до и после запуска тестов. Кроме того, при использовании случайных данных вы можете обнаружить некоторые удивительные пограничные ситуации.

ПРИМЕЧАНИЕ

При запуске в составе интеграционного теста крейт proptest выводит следующее предупреждение:

proptest: FileFailurePersistence::SourceParallel set, but failed to find lib.rs or main.rs.

(proptest: FileFailurePersistence::SourceParallel установлен, но lib.rs или main.rs найти не удалось).

Это предупреждение можно проигнорировать, а дополнительную информацию — найти в статье на GitHub⁵.

7.4.3. Другие инструменты интеграционного тестирования

Для ускорения интеграционных тестов можно применить и некоторые другие крейты, о существовании которых стоит упомянуть:

- `rexpact` — средство автоматизации и тестирования интерактивных CLI-приложений⁶;
- `assert_fs` — средство, предлагающее для приложений фикстуры файловой системы, потребляющие или создающие файлы⁷.

7.5. Fuzz-тестирование

Fuzz-тестирование (или фаззинг⁸) похоже на ранее рассмотренное в этой книге тестирование свойств. Разница между ними заключается в том, что при fuzz-тестировании код тестируется с применением случайно сгенерированных данных, которые могут выходить за рамки допустимости. При тестировании свойств набор входных данных обычно ограничивается значениями, считающимися допустимыми, поскольку зачастую тестировать все возможные входные данные нет никакого смысла, как, впрочем, нет и времени на тестирование всех возможных комбинаций входных данных.

При fuzz-тестировании понятия допустимого и недопустимого отменяются, а в код просто вводятся случайные байты, позволяющие увидеть, что при этом происходит. Fuzz-тестирование особенно популярно в контекстах, чувствительных к безопасности, где нужно понять, что происходит при вводе неправильных данных.

Типичным примером этого является использование данных из общедоступных источников — таких как веб-формы, в которые могут вводиться любые данные, требующие для дальнейшей обработки соответствующего анализа и проверки. Поскольку веб-формы находятся в свободном доступе, ничего не мешает кому-то

⁵ См. <https://github.com/AltSysrq/proptest/issues/233>.

⁶ См. <https://crates.io/crates/rexpact>.

⁷ См. https://crates.io/crates/assert_fs.

⁸ Фаззинг (fuzzing или fuzz testing) — иногда его называют нечетким тестированием (нечетким анализом) — техника тестирования программного обеспечения, часто автоматическая или полуавтоматическая. Заключается в передаче приложению на вход неправильных, неожиданных или случайных данных.

заносить в них случайные данные. Представьте себе, к примеру, форму входа с именем пользователя и паролем, где кто-то (или что-то) в попытке получить доступ к системе может либо перепробовать все комбинации имени пользователя и пароля из списка наиболее распространенных комбинаций, либо внедрить некий «волшебный» набор байтов, вызывающий внутренний сбой кода и дающий возможность обойти систему аутентификации. Как ни удивительно, но такие типы уязвимостей получили широкое распространение, и тестирование методом нечеткого анализа является одной из стратегий для противостояния им.

Основная проблема фаззинга — непомерное количество времени, которое может потребоваться для проверки каждого набора возможных входных данных, но на практике проверять каждую комбинацию входных битов, чтобы найти ошибки, совсем не обязательно. Способности фаззинга к быстрому поиску ошибок в коде, считавшемся до этого непробиваемым, могут вызвать явное изумление.

Для fuzz-тестирования мы воспользуемся библиотекой libFuzzer⁹, являющейся частью LLVM-проекта. libFuzzer можно задействовать непосредственно с FFI-интерфейсом (рассмотренным в главе 4), но мы вместо этого обратимся за помощью к крейту cargo-fuzz, который позаботится о предоставлении библиотеке libFuzzer требуемого Rust API и генерирует для нас соответствующий шаблон.

Прежде чем приступить к изучению примера кода, отметим, как libFuzzer работает на высоком уровне, — библиотека заполняет структуру (предоставленную нами) случайными данными, содержащими аргументы функции, и многократно вызывает функцию нашего кода. Если данные приводят к появлению ошибки, библиотека это обнаруживает и создает тестовый сценарий для запуска условий появления ошибки.

Установите крейт cargo-fuzz, запустив на выполнение команду:

```
cargo install cargo-fuzz
```

и можете приступить к подготовке теста. В листинге 7.10 приведен код относительно простой и на первый взгляд вполне работоспособной функции, которая на самом деле содержит едва заметную ошибку, срабатывающую при определенных условиях.

Листинг 7.10. Функция строчного парсинга с затаившейся ошибкой

```
pub fn parse_integer(s: &str) -> Option<i32> { (1)
    use regex::Regex;
    let re = Regex::new(r"^-?\d{1,10}$").expect("Parsing regex failed"); (2)
    if re.is_match(s) {
        Some(s.parse().expect("Parsing failed"))
    } else {
        None
    }
}
```

⁹ См. <https://llvm.org/docs/LibFuzzer.html>.

Здесь:

- в поз. (1) — проверка с помощью регулярного выражения содержимого строки на наличие в ней только цифр, формирующих также и отрицательные числа;
- в поз. (2) — будет соответствовать строке с цифровой последовательностью длиной от 1 до 10 и с опционально присутствующим перед ней знаком «минус».

В качестве входных данных эта функция примет строку и преобразует ее в целое число формата `i32` — при условии, что длина цифровой последовательности будет в диапазоне от 1 до 10 и опционально иметь префикс `-` (минус). Если входные данные не соответствуют шаблону, функция возвращает `None`. Приводить к сбою программы при недопустимости входных данных эта функция не должна. Но в этом, казалось бы, совершенно безобидном коде кроется серьезная ошибка.

Подобные пограничные ошибки могут вызвать неопределенное поведение, что в контексте безопасности может привести к нехорошим последствиям.

А теперь давайте воспользуемся `cargo-fuzz` и создадим небольшой fuzz-тест. Сначала нужно инициализировать шаблонный код, запустив команду:

```
cargo fuzz init
```

В результате в проекте появится следующая структура:

```
$ tree
.
├── Cargo.lock
├── Cargo.toml
└── fuzz
    ├── Cargo.lock
    ├── Cargo.toml
    └── fuzz_targets
        └── fuzz_target_1.rs
└── src
    └── lib.rs
```

3 directories, 6 files

Здесь показано, что `cargo-fuzz` создал в подкаталоге `fuzz` новый проект, где в файле `fuzz_target_1.rs` будет содержаться собственно тест. Получить список fuzz-целей (или тестов) можно с помощью команды `cargo fuzz list`, в результате выполнения которой на экран будет выведено `fuzz_target_1`.

Далее нужно написать fuzz-тест. Для тестирования мы просто вызовем функцию со случайной строкой, предоставляемой fuzzing-библиотекой, а для получения данных в нужной нам форме — воспользуемся крейтом `Arbitrary`¹⁰. Код fuzz-теста приведен в листинге 7.11.

¹⁰ См. <https://crates.io/crates/arbitrary>.

Листинг 7.11. Fuzz-тест

```

#![no_main]
use arbitrary::Arbitrary;
use libfuzzer_sys::fuzz_target;

#[derive(Arbitrary, Debug)] (1)
struct Input {
    s: String, (2)
}

fuzz_target!({|input: Input| { (3)
    use fuzzme::parse_integer;

    parse_integer(&input.s); (4)
}});

```

Здесь:

- в поз. (1) — использование derive для автоматической генерации типажей Arbitrary и Debug;
- в поз. (2) — наша структура Input не содержит ничего, кроме одной строки. Структура будет заполняться фаззером произвольными данными;
- в поз. (3) — тут используется макрос fuzz_target!, предоставленный cargo-fuzz и определяющий точку входа для нашего fuzz-теста;
- в поз. (4) — тут наша функция вызывается с нашими случайными строковыми данными, предоставляемыми фаззером.

Теперь можно запустить fuzz-тест и посмотреть, что из этого получится. Скорее всего, о наличии ошибки уже известно, поэтому ожидается, что она себя обязательно проявит. Запустите фаззер командой:

```
cargo fuzz run fuzz_target_1
```

и на экран будет выведена информация, похожая на приведенную в листинге 7.12 (который был сокращен, поскольку фаззер генерирует слишком много регистрируемых данных).

Листинг 7.12. Информация, выводимая после запуска команды cargo fuzz run fuzz_target_1

```

cargo fuzz run fuzz_target_1
Compiling fuzzme-fuzz v0.0.0
(/Users/brenden/dev/code-like-a-pro-in-rust/code/c7/7.5/fuzzme/fuzz)
Finished release [optimized] target(s) in 1.07s
Finished release [optimized] target(s) in 0.01s
Running `fuzz/target/x86_64-apple-darwin/release/fuzz_target_1
-artifact_prefix=/Users/brenden/dev/code-like-a-pro-in-rust/
➥ code/c7/7.5/fuzzme/fuzz/artifacts/fuzz_target_1/

```

```
/Users/brenden/dev/code-like-a-pro-in-rust/code/c7/7.5/fuzzme/fuzz/corpus/fuzz_target_1`  
fuzz_target_1(14537,0x10d0a6600) malloc: nano zone abandoned due to  
inability to preallocate reserved vm space.  
INFO: Running with entropic power schedule (0xFF, 100).  
... здесь часть вывода удалена ...
```

Failing input:

```
fuzz/artifacts/fuzz_target_1/  
crash-105eb7135ad863be4e095db6ffe64dc1b9ala466
```

Output of `std::fmt::Debug`:

```
Input {  
    s: "8884844484",  
}
```

Reproduce with:

```
cargo fuzz run fuzz_target_1 fuzz/artifacts/fuzz_target_1/  
crash-105eb7135ad863be4e095db6ffe64dc1b9ala466
```

Minimize test case with:

```
cargo fuzz tmin fuzz_target_1 fuzz/artifacts/fuzz_target_1/  
crash-105eb7135ad863be4e095db6ffe64dc1b9ala466
```

ПРИМЕЧАНИЕ

Работа фаззера даже на быстрых машинах может занять весьма много времени. Хотя этот пример должен сработать довольно быстро (чаще всего в течение 60 секунд), более сложные тесты могут выполняться гораздо дольше. Для неограниченных данных (например, строки без ограничения по длине) фаззер может работать бесконечно долго.

В нижней части `cargo-fuzz` выводит на экран информацию о входных данных, вызвавших сбой. Кроме этого, им для нас создается тестовый сценарий, которым можно будет воспользоваться, чтобы убедиться, что эта ошибка в будущем возникать уже не будет. Для нашего же примера — чтобы еще раз протестировать созданный код с теми же входными данными — мы можем просто запустить команду

```
cargo fuzz run fuzz_target_1 fuzz/artifacts/fuzz_target_1/  
crash-105eb7135ad863be4e095db6ffe64dc1b9ala466
```

что позволит без труда протестировать исправление этой ошибки, не перезапуская фаззер. Поиск вызывающих сбой тестовых сценариев может занять много времени, поэтому использование указанной команды позволит сэкономить время, которое пришлось бы потратить на запуск фаззера.

Попробуйте в качестве упражнения изменить функцию так, чтобы она больше не давала сбоя. Для решения этой задачи есть несколько разных способов, и я дам

подсказку: метод `parse()` уже возвращает нам тип `Result`. Более полную информацию об использовании крейта `cargo-fuzz` можно найти в его документации¹¹.

Резюме

- Интеграционные тесты дополняют своих модульных собратьев, но имеют одно важное отличие — они применяются только к открытым интерфейсам.
- Интеграционные тесты могут использоваться для проверки качества спроектированного API-интерфейса, позволяющей убедиться в его приемлемости и в благосклонности к нему конечного пользователя.
- Встроенная в Rust структура интеграционного тестирования предоставляет минимальные возможности, но для большинства целей ее более чем достаточно.
- В интеграционных тестах Rust, как и в их модульных собратьях, используется библиотека `libtest`, являющаяся частью ядра Rust.
- Для совершенствования интеграционных тестов можно воспользоваться такими крейтами, как `proptest`, `assert_cmd`, `assert_fs` и `rexpect`.
- Крейтом `cargo-fuzz` реализуется интеграция `libFuzzer` и команды `Cargo`, позволяющая настраивать и запускать fuzz-тесты.

¹¹ См. <https://rust-fuzz.github.io/book/>.

Часть IV

Асинхронный Rust

Вам не удастся добиться чего-либо стоящего в программировании, если вы не приобретете опыта решения задач с использованием конкурентности и методов распараллеливания этих задач. Порой можно обойтись и без всего этого, что зачастую является лучшим решением, позволяющим избежать сложности, но современные компьютеры очень хорошо приспособлены к распараллеливанию, и любая достаточно сложная система будет в той или иной степени нуждаться в задании конкурентных вычислений.

Асинхронное программирование, часто сочетающееся с параллелизмом, является популярной технологией работы с конкурентными вычислениями. Во многих отношениях асинхронное программирование может считаться синтаксическим сокращением, позволяющим создавать код, не ломая себе голову понапрасну. Сложности в нем прячутся за абстракциями, высвобождающими наше мышление для решения высокоуровневых задач, что, в свою очередь, предоставляет нам рычаги воздействия на совершенствование своих навыков.

Чтобы асинхронный Rust мог эффективно задействоваться в контексте системного программирования, необходимо понимать всё, что происходит под завесой применяемых абстракций. Но опасаться сложностей не стоит — как только вами будут усвоены присутствующие в асинхронном Rust определения, абстракции и жаргонные выражения, вы поймете, что пользоваться им со всем, что в нем есть, — одно удовольствие.

8

Асинхронное программирование в Rust

В этой главе:

- асинхронное мышление: обзор асинхронного программирования;
- изучение асинхронных сред выполнения Rust;
- обработка результатов асинхронных задач с помощью фьючерсов;
- смешивание синхронного и асинхронного кода;
- использование `async` и `.await`;
- управление конкурентностью и параллелизмом с помощью `async`;
- реализация асинхронного наблюдателя;
- когда не следует применять асинхронное программирование;
- трассировка и отладка асинхронного кода;
- работа с асинхронным программированием при тестировании.

Конкурентность — важная концепция вычислений и один из серьезнейших факторов повышения эффективности использования компьютерной техники. Конкурентные вычисления позволяют работать со входами и выходами в виде данных, сетевых соединений или периферийных устройств значительно быстрее, чем без их использования. И дело тут не всегда в скорости, но и в задержках, издержках и сложностях системы. На рис. 8.1 показано, что в конкурентном режиме могут выполняться тысячи или даже миллионы относительно несложных задач. При конкурентных вычислениях можно создавать множество задач, выполняемых в этом режиме, удалять такие задачи и управлять ходом их выполнения, причем делать всё это с весьма скромными накладными расходами.

Асинхронное программирование задействует конкурентность, чтобы воспользоваться временем простоя между задачами. Некоторые виды задач, такие как ввод/вывод, выполняются намного медленнее обычных инструкций центрального процессора, и после запуска медленной задачи в ожидании ее завершения эту задачу можно отложить, чтобы поработать над выполнением других задач.



Рис. 8.1. Задачи, выполняемые в конкурентном режиме в одном и том же потоке

Конкурентность не следует путать с *параллелизмом* (который здесь я определяю как способность одновременно выполнять сразу несколько задач). Конкурентность отличается от параллелизма тем, что задачи могут выполняться в одном и том же временном отрезке, но не обязательно параллельно. При параллелизме можно одновременно выполнять программный код, либо совместно используя одну и ту же область памяти на хост-машине либо на нескольких центральных процессорах, либо организовав переключение контекста на уровне операционной системы (более подробно эта тема в книге не рассматривается). Два потока, выполняемые параллельно в одно и то же время, показаны на рис. 8.2.

Чтобы провести со всем этим какую-нибудь аналогию, рассмотрим осознанные действия людей — мы просто не в состоянии параллельно выполнять большинство

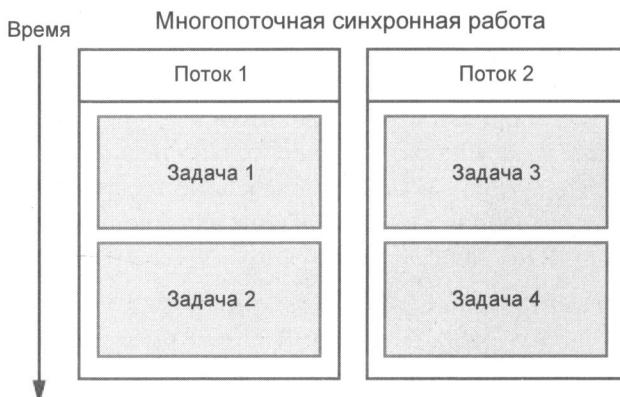


Рис. 8.2. Синхронные задачи, выполняемые параллельно в двух потоках

задач, но множество вещей можем делать одновременно. Попробуйте, к примеру, одновременно вести беседу с двумя или более людьми — это гораздо сложнее, чем кажется. Можно разговаривать со многими людьми одновременно, но вам придется переключаться между ними и делать паузу, переводя свое внимание с одного человека на другого. Люди неплохо справляются с конкурентностью, но пасуют перед параллелизмом.

В этой главе мы рассмотрим имеющуюся в Rust асинхронную конкурентную систему, предусматривающую, в зависимости от потребностей, возможность работы в *обоих* режимах: конкурентном и параллельном. Реализовать параллелизм без асинхронности (используя потоки) относительно просто, а вот реализовать без асинхронности конкурентность довольно сложно. Асинхронность — весьма обширная тема, поэтому мы коснемся только ее основ. А те из вас, кто уже знаком с асинхронным программированием, найдут здесь всё необходимое для эффективной работы с асинхронностью в Rust.

8.1. Среды выполнения

Асинхронность в Rust похожа на то, что встречается в других языках, хотя, в дополнение к заимствованию из других языков многоного того, что хорошо там работает, у нее есть и свои уникальные особенности. У тех, кто знаком с асинхронностью в JavaScript, Python или даже с `std::async` в C++, проблем с адаптацией к асинхронности Rust возникать не должно. У Rust есть одно большое отличие — сам язык не предоставляет и не назначает реализацию асинхронной среды выполнения. Rust предоставляет *только* типаж `Future`, ключевое слово `async` и инструкцию `.await`, а подробности реализации в основном возложены на сторонние библиотеки. На момент подготовки книги существовало три широко используемые реализации асинхронной среды выполнения, представленные в табл. 8.1.

Таблица 8.1. Асинхронные среды выполнения

Название	Загрузки*	Описание
Tokio	144,128,598	Полнофункциональная асинхронная среда выполнения
async-std	18,874,366	Реализация стандартной библиотеки Rust с асинхронностью
Smol	3,604,871	Облегченная среда выполнения, предназначенная для конкурирования с Tokio

* Количество загрузок для каждого крейта указано по состоянию на 26 декабря 2023 года.

Среды `async-std` и `Smol` обеспечивают совместимость со средой выполнения `Tokio`, но смешивать в Rust конкурирующие асинхронные среды выполнения в одном контексте нецелесообразно. И хотя отдельные среды выполнения реализуют один и тот же асинхронный API, вам в большинстве случаев, скорее всего, потребуются функции, специфичные в принятой среде выполнения. Для большинства целей рекомендуется применять `Tokio` как самую зрелую и популярную среду исполнения. Воз-

можно, в будущем смена или взаимозаменяемость сред выполнения упростится, но пока этим заниматься не стоит.

Крейты, предоставляющие асинхронные функции, теоретически могут использовать любую среду выполнения, но на практике это встречается довольно редко, поскольку обычно они лучше всего работают с одной конкретной средой выполнения. Получается, что в асинхронной экосистеме Rust существует некая бифуркация, при которой большинство крейтов разработаны для применения с Tokio, но есть и крейты, приспособленные под Smol или `async-std`. Поэтому в этой книге основное внимание мы уделим Tokio как наиболее предпочтительной асинхронной среде выполнения.

8.2. Асинхронное мышление

Говоря об асинхронном программировании, обычно имеют в виду способ обработки потока управления для любой операции, включающей ожидание завершения задачи, которой зачастую является операция ввода/вывода. Примерами таких задач могут послужить взаимодействия с файловой системой или сокетом, а также медленные операции, наподобие вычисления хеша или ожидания завершения работы таймера.

Синхронный ввод/вывод, который (за исключением некоторых языков, таких как JavaScript) является режимом обработки ввода/вывода по умолчанию, знаком большинству людей. А использование асинхронного программирования (в отличие от синхронного) дает следующие преимущества:

- отсутствие при поддержке конкурентности необходимости в переключении контекста между потоками, что при работе в асинхронном режиме существенно ускоряет выполнение операций ввода/вывода. Переключение контекста, при котором зачастую используется синхронизация или блокировка с помощью мьютексов, может быть связано с невероятно большим объемом издержек;
- использовать программы, написанные с прицелом на асинхронность, гораздо удобнее, поскольку она позволяет избегать многих разновидностей возникновения условий гонки;
- асинхронным задачам свойственна облегченность, позволяющая одновременно без особого труда обрабатывать тысячи или миллионы таких задач.

Когда речь, в частности, заходит об операциях ввода/вывода, количество времени ожидания завершения операций зачастую существенно превышает время, затрачиваемое на обработку результатов таких операций. Благодаря этому в ожидании завершения задач допустимо выполнять другую работу, не выполняя последовательно каждую задачу. Иными словами, с помощью асинхронного программирования осуществляется эффективная разбивка и чередование наших вызовов функций в промежутках, определяемых временем, затрачиваемым на ожидание завершения операции ввода/вывода.

На рис. 8.3 показана разница во времени блокирующих и неблокирующих операций ввода/вывода. Асинхронный ввод/вывод носит неблокирующий характер, а син-

хронный — является блокирующим. Если предположить, что время обработки результата операции ввода/вывода намного меньше времени, затрачиваемого на ожидание завершения ввода/вывода, асинхронный ввод/вывод зачастую будет выполняться гораздо быстрее. Следует также отметить, что с асинхронностью можно использовать многопоточное программирование, но часто быстрее будет просто воспользоваться одним потоком.

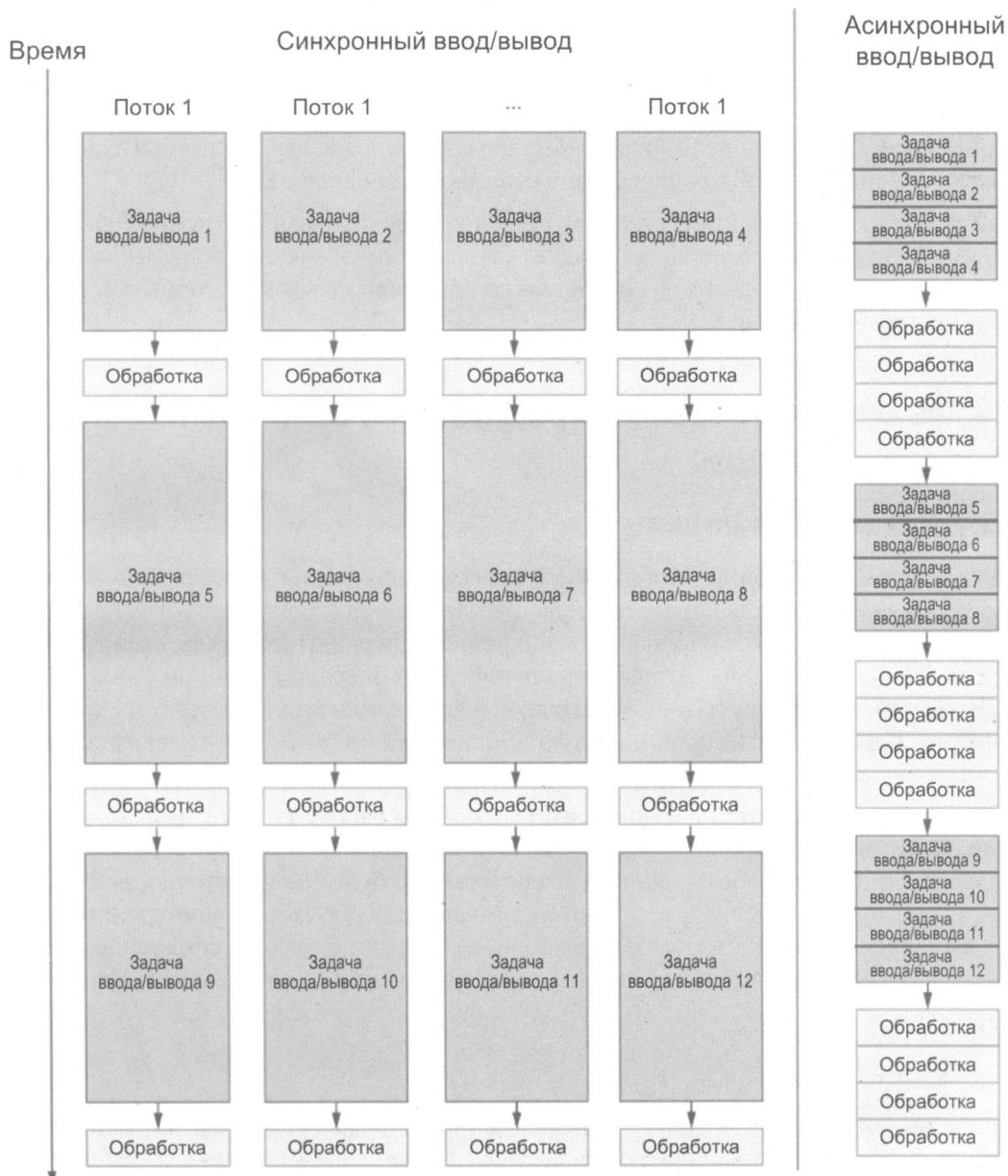


Рис. 8.3. Сравнение синхронного и асинхронного ввода/вывода

Но, как известно, ничего не бывает бесплатным, и если время обработки данных, полученных из операций ввода/вывода, становится больше времени, затраченного на ожидание завершения ввода/вывода, станет заметно ухудшение производительности (если предположить применение однопоточной асинхронности). Получается, асинхронность идеальна не для каждого варианта использования. Успокаивает то, что Tokio проявляет в выборе способа выполнения асинхронных задач вполне разумную гибкость, — например, в подборе количества используемых рабочих потоков.

С асинхронностью может также смешиваться и параллелизм, поэтому сравнивать напрямую асинхронное программирование с синхронным порой не имеет никакого смысла. Асинхронный код способен одновременно параллельно выполнятся сразу в нескольких потоках, что придает ему характер множителя производительности, с которым синхронный код просто не в состоянии соревноваться.

Как только у вас выработается тип мышления, необходимый для понимания асинхронного программирования, оно покажется вам гораздо менее сложным, чем синхронное программирование, — особенно по сравнению с многопоточным синхронным программированием.

8.3. Фьючерсы: обработка результатов выполнения асинхронных задач

8.3.1. Режим бездействия

Большинство асинхронных библиотек и языков основаны на *фьючерсах* — паттернах проектирования для обработки задач, возвращающих результат в будущем (отсюда и название). При выполнении асинхронной операции ее результатом, в отличие от непосредственного возврата значения самой операции (как при синхронном программировании или обычном вызове функции), является фьючерс. Фьючерсы, конечно, представляют собой удобную абстракцию, но требуют от программиста немного больше усилий для их правильной обработки.

Чтобы лучше разобраться в фьючерсах, рассмотрим работу таймера: можно создать (или запустить) асинхронный таймер, возвращающей фьючерс для подачи сигнала о завершении своей работы. Просто создать таймер будет недостаточно, поскольку нам также нужно сообщить исполнителю (являющемуся частью асинхронной среды выполнения) о необходимости выполнения задачи. Когда в синхронном коде надо на 1 секунду войти в режим бездействия, можем просто вызвать функцию `sleep()`.

ПРИМЕЧАНИЕ

Вызвать `sleep()` можно вообще-то и внутри асинхронного кода, но делать это категорически не стоит. Важное правило асинхронного программирования заключается в полном запрете на блокировку основного потока. Хотя вызов `sleep()` не приведет к сбою программы, он фактически дискредитирует саму цель асинхронного программирования и поэтому считается антипаттерном.

Чтобы сравнить работу асинхронного таймера с синхронным, давайте посмотрим, что понадобится для написания небольшой программы на Rust, входящей в режим бездействия на 1 секунду и выводящей на экран строку "Hello, world!". Сначала посмотрим на синхронный код:

```
fn main() {
    use std::thread, time;

    let duration = time::Duration::from_secs(1);

    thread::sleep(duration);

    println!("Hello, world!");
}
```

В нем всё просто и красиво. А теперь рассмотрим асинхронную версию:

```
fn main() {
    use std::time;

    let duration = time::Duration::from_secs(1);

    tokio::runtime::Builder::new_current_thread()
        .enable_time()                                         (1)
        .build()
        .unwrap()
        .block_on(async {                                       (2)
            tokio::time::sleep(duration).await;
            println!("Hello, world!");
        });
}
```

Здесь:

- в поз. (1) — среда выполнения поддерживает `time` или ввод/вывод, которые можно включить по отдельности или все сразу с помощью `enable_all()`;
- в поз. (2) — создание асинхронного блока с ожиданием на фьючерсе, возвращаемом `tokio::time::sleep()`, с последующим выводом на экран "Hello, world!".

Ого! Это же гораздо сложнее. К чему вся эта сложность? Дело в том, что асинхронное программирование требует особого потока управления, который в основном реализуется средой выполнения, но всё равно требует другого стиля программирования. Диспетчер среды выполнения решает, что и когда запускать, но нам нужно предоставить диспетчеру возможность переключаться между задачами. Управление большинством деталей будет в таком случае возложено на среду выполнения, но нам для эффективного использования асинхронности по-прежнему надо будет оставаться в курсе происходящего. В большинстве случаев предоставлять диспетчеру возможность переключения ничуть не сложнее применения инструкции `.await`, рассматриваемого в следующем разделе.

Что понимается под блокировкой основного потока?

Секрет написания качественного асинхронного кода, как уже отмечалось, заключается в исключении блокировки основного потока. Когда говорится о блокировке основного потока, фактически имеется в виду, что среда выполнения не должна быть лишена возможности переключать задачи в течение продолжительных периодов времени. Обычно блокирующей операцией считается ввод/вывод, поскольку время, необходимое для завершения такой операции, зависит от нескольких факторов вне контекста создаваемой программы и ее механизмов управления. Но у вас также могут иметься задачи, строго привязанные к центральному процессору, которые при условии, что их выполнение занимает достаточно много времени, считаются блокирующими.

Слишком продолжительная блокировка основного потока может быть предотвращена путем ввода точек выхода. Под *точкой выхода* понимается любой код, возвращающий управление диспетчеру. Присоединение или ожидание на фьючерсе создает точку выхода с передачей управления по цепочке в среду выполнения.

Вопрос о том, что именно подразумевается под *продолжительным периодом времени*, во многом зависит от контекста, поэтому четкого представления на этот счет дать невозможно. Но можно понять, что подразумевается под быстрыми и медленными операциями, посмотрев, сколько времени обычно занимают операции, привязанные к центральному процессору и вводу/выводу. Чтобы оценить разницу между быстрым и медленным, можно сравнить типичный вызов функции (являющийся быстрой операцией) со временем, уходящем на простую операцию ввода/вывода (медленную операцию).

Давайте прикинем время, необходимое для выполнения обычной функции. Время, уходящее на один тактовый цикл, можно вычислить, взяв величину, обратную частоте работы центрального процессора (предполагая, что одна инструкция выполняется за один тактовый цикл).

Например, для центрального процессора с тактовой частотой 2 ГГц на выполнение инструкции уходит 0,5 нс. Тогда на выполнение операции из 50 инструкций (что приблизительно соответствует обычному вызову функции) уйдет около 25 нс.

Если сравнивать это с небольшой операцией ввода/вывода,читывающей, к примеру, 1024 байта из файла, то она может занять значительно больше времени. Продемонстрирую это запуском небольшого теста на моем ноутбуке:

```
$ dd if=/dev/random of=testfile bs=1k count=1 (1)
```

```
1+0 records in
```

```
1+0 records out
```

```
1024 bytes (1.0 kB, 1.0 KiB) copied, 0.000296943 s, 3.4 MB/s
```

```
$ dd if=testfile of=/dev/null (2)
```

```
2+0 records in
```

```
2+0 records out
```

```
1024 bytes (1.0 kB, 1.0 KiB) copied, 0.000261919 s, 3.9 MB/s
```

Здесь:

- в поз. (1) — запись в тестовый файл 1000 случайных байтов из /dev/random;
- в поз. (2) — считывание содержимого тестового файла и его запись в /dev/null.

В этом teste чтение из небольшого файла занимает порядка 262 мкс, что примерно в 5240 раз больше, чем 50 нс. Сетевые операции, в зависимости от целого ряда факторов, будут, вероятно, выполняться еще на один-два порядка медленнее.

Операции, не относящиеся к вводу/выводу, которые, на ваш взгляд, могут занять относительно много времени, нужно либо рассматривать как блокирующие с применением `tokio::task::spawn_blocking()`, либо разбивать их, вводя по мере необходимости вызовы `.await`, давая тем самым диспетчеру возможность предоставления другим задачам времени на их выполнение. Если не будет полной уверенности в том, что от подобных оптимизаций получится какой-либо выигрыш, код нужно протестировать.

8.3.2. Определение среды выполнения с помощью `#[tokio::main]`

Для обертывания вашей функции `main()` среда Tokio предоставляет специальный макрос, поэтому приведенный в предыдущем разделе код таймера при желании можно упростить, придав ему следующий вид:

```
#[tokio::main]
async fn main() {
    use std::time;

    let duration = time::Duration::from_secs(1);

    tokio::time::sleep(duration).await;
    println!("Hello, world!");
}
```

Благодаря примененному синтаксическому сокращению теперь код выглядит практически так же, как и код синхронной версии. С помощью ключевого слова `async` функция `main()` превращена в асинхронную, а атрибут `#[tokio::main]` занимается шаблоном, необходимым для запуска среды выполнения Tokio и создания нужного нам асинхронного контекста.

Вспомним, что результатом выполнения любой асинхронной задачи является фьючерс, но, прежде чем что-либо произойдет, его нужно выполнить в среде выполнения. В Rust для этого обычно применяется инструкция `.await`, рассматриваемая в следующем разделе.

8.4. Ключевые слова `async` и `.await`: когда и где их использовать?

Ключевые слова `async` и `.await` появились в Rust совсем недавно. Фьючерсы могут применяться и без них, но при первой же возможности лучше все-таки воспользоваться `async` и `.await`, поскольку именно они не в ущерб функциональности обрабатывают основную часть шаблона. Помеченные как `async` функция или блок кода, вернут фьючерс, а `.await` сообщит среде выполнения о нашем желании дождаться результата. Синтаксис `async` и `.await` позволяет создавать асинхронный код, внешне похожий на синхронный, но без особой сложности, возникающей при работе с фьючерсами. Ключевое слово `async` может использоваться с функциями, замыканиями и блоками кода. Блоки с пометкой `async` не выполняются, пока не будут опрошены, что можно сделать с помощью `.await`.

Когда во фьючерсе применяется `.await`, закулисно используется среда выполнения, вызывающая метод `poll()` из типажа `Future` и ожидающая результата выполнения фьючерса. Если обращения к `.await` вообще не будет (или же фьючерс не будет опрошен явным образом), фьючерс никогда не выполнится.

Чтобы воспользоваться `.await`, нужно находиться в асинхронном контексте, который может быть создан путем написания блока кода, помеченного как `async`, и выполнения этого кода в асинхронной среде выполнения. Использовать `async` и `.await` *совсем не обязательно*, но с ними всё гораздо проще, и среда выполнения Tokio (вместе со многими другими асинхронными крейтами) была разработана именно для такого их применения.

Давайте, к примеру, рассмотрим следующую программу, включающую в себя асинхронный блок кода типа «запустил и забыл», созданный с применением метода `tokio::task::spawn()`:

```
[tokio::main]
async fn main() {
    async {
        println!("This line prints first");
    }
    .await;
    let _future = async {
        println!("This line never prints");
    };
    tokio::task::spawn(async {
        println!(
            "This line prints sometimes, depending on how quick it runs"
        )
    });
}

println!("This line always prints, but it may or may not be last");
}
```

Если запустите этот код несколько раз, то он (как ни странно) выведет либо две, либо три строки. Стока в первом макросе `println!()` всегда будет выведена перед другими строками из-за применения инструкции `.await`, ожидающей результата выполнения первого фьючерса. Стока во втором макросе `println!()` не будет выведена никогда, поскольку фьючерс не будет выполнен из-за отсутствия вызова `.await` или же какого-либо иного его порождения в среде выполнения. Третий фьючерс с макросом `println!()` порождается в среде выполнения Tokio, но мы не ждем его завершения, поэтому не получаем никаких гарантий его запуска и не знаем, запустится ли он до или после последнего макроса `println!()`, который всегда будет выводить свою строку.

Почему третий в этой последовательности макрос `println!()` не выводит строку? Возможно, программа завершает работу еще до того, как диспетчер среды выполнения Tokio получает возможность выполнения кода. Если нужно гарантировать

запуск кода до выхода из программы, надо дождаться завершения выполнения фьючерса, возвращаемого `tokio::task::spawn()`.

Функция `tokio::task::spawn()` обладает еще одной важной особенностью — она позволяет запускать асинхронную задачу в асинхронной среде выполнения извне асинхронного контекста. Она также возвращает фьючерс (в частности, `tokio::task::JoinHandle`), который можно будет передать как любой другой объект. Дескрипторы соединения Tokio также позволяют по нашему желанию прерывать задачи. Давайте рассмотрим пример, показанный в листинге 8.1.

Листинг 8.1. Порождение задачи с помощью `tokio::task::spawn()`

```
use tokio::task::JoinHandle;

fn not_an_async_function() -> JoinHandle<()> {           (1)
    tokio::task::spawn(async {                                (2)
        println!("Second print statement");
    })
}

#[tokio::main]
async fn main() {
    println!("First print statement");
    not_an_async_function().await.ok();                      (3)
}
```

Здесь:

- в поз. (1) — обычная функция, возвращающая `JoinHandle` (где реализован типаж `Future`);
- в поз. (2) — наша задача `println!()` создается во время выполнения программы;
- в поз. (3) — ключевое слово `.await` используется для фьючерса, возвращаемого нашей функцией с целью дождаться его выполнения.

В коде этого листинга создается обычная функция, возвращающая `JoinHandle` (являющийся просто типом фьючерса). Метод `tokio::task::spawn()` возвращает `JoinHandle`, позволяющий присоединиться к задаче (т. е. получить результат выполнения нашего блока кода, который является просто единичной составляющей этого примера).

А что будет, если захочется воспользоваться `.await` вне асинхронного контекста? Отвечу коротко — этого делать нельзя. Но вы можете заблокировать результат выполнения фьючерса, чтобы можно было его когда-либо дождаться, воспользовавшись методом `tokio::runtime::Handle::block_on()`. Для этого надо будет получить дескриптор для среды выполнения и переместить его в поток, где нужно выполнить блокировку. Как показано в листинге 8.2, дескрипторы можно клонировать и совместно использовать, предоставив доступ к асинхронной среде выполнения извне асинхронного контекста.

Листинг 8.2. Использование Tokio Handle для порождения задач

```
use tokio::runtime::Handle;

fn not_an_async_function(handle: Handle) {
    handle.block_on(async {                                     (1)
        println!("Second print statement");
    })
}

#[tokio::main]
async fn main() {
    println!("First print statement");

    let handle = Handle::current();                         (2)
    std::thread::spawn(move || {                           (3)
        not_an_async_function(handle);                   (4)
    });
}
```

Здесь:

- в поз. (1) — порождение блокирующей асинхронной задачи в нашей среде выполнения путем использования дескриптора среды выполнения;
- в поз. (2) — получение дескриптора среды выполнения для текущего контекста среды выполнения;
- в поз. (3) — порождение нового потока путем захвата переменных с помощью перемещения;
- в поз. (4) — вызов нашей неасинхронной функции в отдельном потоке с передачей дескриптора асинхронной среды выполнения.

Пусть этот код и не слишком красиво выглядит, но он работает. Бывают обстоятельства, речь о которых пойдет в следующем разделе, когда такой подход может вам пригодиться, но в большинстве случаев следует по возможности попытаться воспользоваться `async` и `.await`.

Коротко говоря, когда нужно выполнить асинхронные задачи или вернуть фьючерс, оберните блоки кода (включая функции и замыкания) в `async`, а когда надо дождаться выполнения асинхронной задачи, воспользуйтесь `.await`. Создание асинхронного блока не приводит к выполнению фьючерса — его всё равно нужно выполнять (или порождать с помощью `tokio::task::spawn()`) в ходе выполнения программы.

8.5. Конкурентность и параллелизм с `async`

В начале главы мы уже рассматривали различия между конкурентностью и параллелизмом. С `async` получить бесплатно конкурентность или параллелизм невозможно. Чтобы воспользоваться возможностями этих режимов вычислений, придется всё же задуматься о структурировании кода.

В Tokio нет явно выраженного контроля над параллелизмом (кроме запуска блокирующей задачи с помощью `tokio::task::spawn_blocking()`, которая всегда выполняется в отдельном потоке). И хотя у нас есть явный контроль над concurrentностью, мы не в состоянии контролировать параллельное выполнение отдельно взятых задач, поскольку такой контроль оставлен на усмотрение среды выполнения. Tokio позволяет задавать количество рабочих потоков, но какие именно потоки использовать для каждой задачи, будет решать среда выполнения.

Введение concurrentности в ваш код может быть выполнено одним из трех способов:

- порождением задач с помощью `tokio::task::spawn()`;
- объединением нескольких фьючерсов с помощью макроса `tokio::join!(...)` или метода `futures::future::join_all()`;
- использованием макроса `tokio::select! { ... }`, позволяющего выстраивать ожидание из нескольких concurrentных ветвей кода (смоделированных по образцу системного вызова UNIX `select()`).

Чтобы ввести в код *параллелизм*, следует воспользоваться методом `tokio::task::spawn()`, но получить таким образом явный параллелизм не получится. Мы просто при порождении задачи сообщаем Tokio, что она может быть выполнена в любом потоке, но решение, какой именно поток использовать, всё же остается за Tokio. Если, к примеру, запустить среду выполнения Tokio только с одним рабочим потоком, все задачи, даже если будет использован метод `tokio::task::spawn()`, будут выполняться в этом потоке. Такое поведение можно продемонстрировать на примере, код которого приведен в листинге 8.3.

Листинг 8.3. Демонстрация асинхронной concurrentности и параллелизма

```
async fn sleep_ls_blocking(task: &str) {
    use std::{thread, time::Duration};
    println!("Entering sleep_ls_blocking({task})");
    thread::sleep(Duration::from_secs(1));                                (1)
    println!("Returning from sleep_ls_blocking({task})");
}

#[tokio::main(flavor = "multi_thread", worker_threads = 2)]           (2)
async fn main() {
    println!("Test 1: Run 2 async tasks sequentially");
    sleep_ls_blocking("Task 1").await;                                     (3)
    sleep_ls_blocking("Task 2").await;

    println!("Test 2: Run 2 async tasks concurrently (same thread)");
    tokio::join!(                                         (4)
        sleep_ls_blocking("Task 3"),
        sleep_ls_blocking("Task 4")
    );
}
```

```

println!("Test 3: Run 2 async tasks in parallel");
tokio::join!()                                (5)
    tokio::spawn(sleep_1s_blocking("Task 5")),
    tokio::spawn(sleep_1s_blocking("Task 6"))
);
}

```

Здесь:

- в поз. (1) — чтобы продемонстрировать параллелизм, тут намеренно используется блокирующий метод `std::thread::sleep()`;
- в поз. (2) — Tokio в явном виде настраивается на два рабочих потока, что позволяет выполнять задачи в параллельном режиме;
- в поз. (3) — тут дважды последовательно вызывается наша функция `sleep_1s()` без каких-либо конкурентности или параллелизма;
- в поз. (4) — тут для введения конкурентности `sleep_1s()` вызывается дважды, с применением `tokio::join!();`
- в поз. (5) — нами наконец порождается `sleep_1s()`, после чего результаты объединяются, чем и реализуется параллелизм.

Выполнение этого кода приведет к выводу на экран следующей информации:

```

Test 1: Run 2 async tasks sequentially
Entering sleep_1s_blocking(Task 1)
Returning from sleep_1s_blocking(Task 1)
Entering sleep_1s_blocking(Task 2)
Returning from sleep_1s_blocking(Task 2)
Test 2: Run 2 async tasks concurrently (same thread)
Entering sleep_1s_blocking(Task 3)
Returning from sleep_1s_blocking(Task 3)
Entering sleep_1s_blocking(Task 4)
Returning from sleep_1s_blocking(Task 4)
Test 3: Run 2 async tasks in parallel
Entering sleep_1s_blocking(Task 5)          (1)
Entering sleep_1s_blocking(Task 6)
Returning from sleep_1s_blocking(Task 5)
Returning from sleep_1s_blocking(Task 6)

```

- Здесь в поз. (1) — в третьем тесте видно, что наша функция `sleep_1s()` выполняется параллельно, поскольку вход в обе функции происходит до возвращения.

Из информации, выведенной на экран, понятно, что только в третьем teste, где каждая задача запускается с помощью `tokio::spawn()` (что эквивалентно `tokio::task::spawn()`), код выполняется параллельно. Утверждать, что он выполняется параллельно, можно потому, что оба выражения `Entering ...` показаны до выражений `Returning ...`. Последовательность событий хорошо иллюстрирует диаграмма, показанная на рис. 8.4.

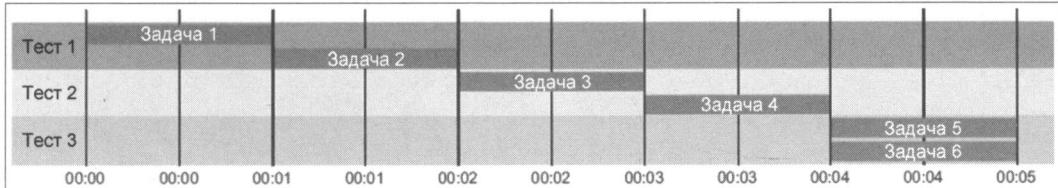


Рис. 8.4. Диаграмма, показывающая последовательность событий при блокирующем бездействии (`sleep_1s_blocking`)

Обратите внимание, что хотя второй тест, конечно же, выполняется в конкурентном режиме, задачи в нем выполняются не параллельно, а последовательно, поскольку в коде листинга 8.3 нами используется блокирующее бездействие. Давайте обновим код, добавив в него неблокирующую бездействие:

```
async fn sleep_1s_nonblocking(task: &str) {
    use tokio::time::{sleep, Duration};
    println!("Entering sleep_1s_nonblocking({task})");
    sleep(Duration::from_secs(1)).await;
    println!("Returning from sleep_1s_nonblocking({task})");
}
```

После обновления `main()` для добавления трех тестов с неблокирующими бездействиями будет получен следующий вывод на экран:

```
Test 4: Run 2 async tasks sequentially (non-blocking)
Entering sleep_1s_nonblocking(Task 7)
Returning from sleep_1s_nonblocking(Task 7)
Entering sleep_1s_nonblocking(Task 8)
Returning from sleep_1s_nonblocking(Task 8)
Test 5: Run 2 async tasks concurrently (same thread, non-blocking)
Entering sleep_1s_nonblocking(Task 9) (1)
Entering sleep_1s_nonblocking(Task 10)
Returning from sleep_1s_nonblocking(Task 10)
Returning from sleep_1s_nonblocking(Task 9)
Test 6: Run 2 async tasks in parallel (non-blocking)
Entering sleep_1s_nonblocking(Task 11)
Entering sleep_1s_nonblocking(Task 12)
Returning from sleep_1s_nonblocking(Task 12)
Returning from sleep_1s_nonblocking(Task 11)
```

□ Здесь в поз. (1) — тут видно, что теперь, когда функция бездействия заменена на неблокирующую, бездействие происходит в конкурентном режиме.

На рис. 8.5 показано, что оба теста: 5 и 6 — по-видимому, выполняются в параллельном режиме, хотя в действительности параллельно выполняются только задачи теста 6, а задачи теста 5 выполняются в конкурентном режиме. Если обновить настройки Tokio еще раз, указав `worker_threads = 1`, а затем перезапустить тест, станет видно, что в версии блокирующего бездействия все задачи выполняются последовательно, а в совпадающей по времени неблокирующей версии они, даже с одним потоком, по-прежнему выполняются в конкурентном режиме.

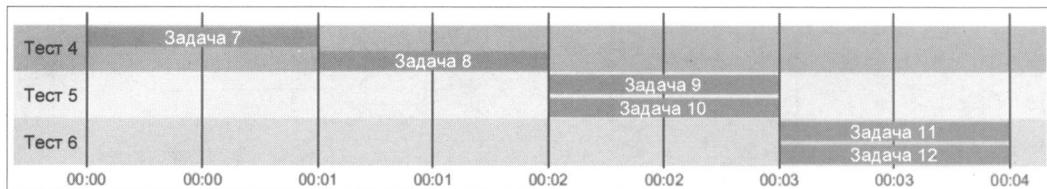


Рис. 8.5. Диаграмма, показывающая последовательность событий при неблокирующем бездействии (`sleep_1s_nonblocking`)

Разобраться в конкурентности и параллелизме в асинхронном Rust можно не сразу, так что не волнуйтесь, если поначалу всё это покажется вам слишком запутанным. Чтобы лучше понять, что происходит, я рекомендую поработать с этим примером самостоятельно и поэкспериментировать с выбором различных параметров.

8.6. Реализация асинхронного наблюдателя

Давайте рассмотрим реализацию паттерна наблюдателя в асинхронном Rust. Особая польза от этого паттерна проявилась именно в асинхронном программировании, поэтому сейчас мы узнаем, что нужно сделать, чтобы заставить его работать с асинхронностью.

ПРИМЕЧАНИЕ

Ожидается, что асинхронные типажи будут добавлены в Rust в предстоящем выпуске, но на момент подготовки книги они еще не были доступны.

Итак, до сих пор еще сохраняется одно весьма существенное ограничение поддержки асинхронности в Rust — невозможность использования типажей с асинхронными методами. Например, следующий код не станет работать:

```
trait MyAsyncTrait {
    async fn do_thing();
}
```

Из-за этого при реализации паттерна наблюдателя с асинхронностью приходится идти на хитрость. Обойти существующую проблему можно несколькими путями, но здесь я показываю решение, дающее ко всему прочему еще и некоторое представление о том, как в Rust реализуется синтаксическое сокращение `async fn`.

Как уже ранее упоминалось, функции `async` и `.await` — просто удобный синтаксис для работы с фьючерсами. При объявлении `async`-функции или блока кода компилятор для нас создает для этого кода обертку из фьючерса. Получается, что мы всё же можем создать эквивалент `async`-функции с типажами, но делать это придется явным образом (без синтаксических сокращений).

Типаж наблюдателя выглядит следующим образом:

```
pub trait Observer {
    type Subject;
    fn observe(&self, subject: &Self::Subject);
}
```

Чтобы преобразовать метод `observe()` в асинхронную функцию, его в первую очередь нужно заставить возвращать `Future`. В качестве первого шага можно попытаться сделать примерно так:

```
pub trait Observer {
    type Subject;
    type Output: Future<Output = ()>; (1)
    fn observe(&self, subject: &Self::Subject) -> Self::Output; (2)
}
```

Здесь:

- в поз. (1) — за счет возвращения `()` определяется ассоциированный тип с привязкой к типажу `Future`;
- в поз. (2) — теперь наш метод `observe()` возвращает ассоциированный тип `Output`.

На первый взгляд кажется, что это должно сработать, и код проходит компиляцию. Но, как только будет предпринята попытка реализации типажа, обнаружится целый ряд проблем. Во-первых, поскольку `Future` — это всего лишь типаж (а не конкретный тип), мы не знаем, какой тип нужно указать для `Output`. Получается, что использовать таким образом ассоциированный тип не представляется возможным. Вместо этого нужно воспользоваться объектом типажа. Для этого наш фьючерс надо вернуть в `Box`. Давайте обновим типаж:

```
pub trait Observer {
    type Subject;
    type Output;
    fn observe(
        &self,
        subject: &Self::Subject,
    ) -> Box<dyn Future<Output = Self::Output>>; (1)
} (2)
```

Здесь:

- в поз. (1) — для возвращаемого типа сохраняется ассоциированный тип, что добавляет некоторую долю гибкости;
- в поз. (2) — теперь вместо прежнего варианта возвращается `Box <dyn Future>`.

Давайте попробуем собрать всё воедино, реализовав для `MyObserver` наш новый асинхронный наблюдатель:

```
struct Subject;
struct MyObserver;

impl Observer for MyObserver {
    type Subject = Subject;
    type Output = ();
    fn observe(
        &self,
        _subject: &Self::Subject,
```

```

) -> Box<dyn Future<Output = Self::Output>> {
    Box::new(async {           ← обратите внимание, что возвращаемый фьючерс пришлось поместить в Box
        // do some async stuff here!
        use tokio::time::(sleep, Duration);
        sleep(Duration::from_millis(100)).await;
    })
}
}

```

Пока все идет неплохо! Компилятор тоже доволен. А что будет, если попытаться всё это протестировать? Давайте создадим небольшой тест:

```

#[tokio::main]
async fn main() {
    let subject = Subject;
    let observer = MyObserver;
    observer.observe(&subject).await;
}

```

И тут мы натыкаемся на следующую загвоздку — попытка скомпилировать этот код приводит к следующей ошибке:

```

error[E0277]: `dyn Future<Output = ()>` cannot be unpinned
--> src/main.rs:29:31
|
29 |     observer.observe(&subject).await;
|             ^^^^^^ the trait `Unpin` is not
|                   → implemented for `dyn Future<Output = ()>`
|
= note: consider using `Box::pin`
= note: required because of the requirements on the impl of `Future`
       → for `Box<dyn Future<Output = ()>>`
= note: required because of the requirements on the impl of `IntoFuture`
       → for `Box<dyn Future<Output = ()>>`
help: remove the `.`await`'
|
29 -     observer.observe(&subject).await;
29 +     observer.observe(&subject);
|
For more information about this error, try `rustc --explain E0277`.

```

Что происходит? Чтобы понять, нужно взглянуть на типаж Future из стандартной библиотеки Rust:

```

pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}

```

Заметьте, что метод `poll()` принимает свой параметр `self` как тип `Pin<&mut Self>`. Иными словами, прежде чем появится возможность опроса фьючерса (чем, собственно, и занимается `.await`), его нужно закрепить. В Rust закрепленный указатель представляет собой особый вид указателя, который нельзя перемещать (пока он не откреплен). К счастью для нас, получить закрепленный указатель совсем нетрудно — просто нужно еще раз обновить наш типаж `Observer`, придав ему следующий вид:

```
pub trait Observer {
    type Subject;
    type Output;
    fn observe(
        &self,
        subject: &Self::Subject,
    ) -> Pin<Box<dyn Future<Output = Self::Output>>>;   ← теперь Box обертывается в Pin,
                                                               что дает нам закрепленный блок
}
```

Далее обновим нашу реализацию, придав ей такой вид:

```
impl Observer for MyObserver {
    type Subject = Subject;
    type Output = ();
    fn observe(
        &self,
        _subject: &Self::Subject,
    ) -> Pin<Box<dyn Future<Output = Self::Output>>> {           (1)
        Box::pin(async {                                              (2)
            // do some async stuff here!
            use tokio::time::{sleep, Duration};
            sleep(Duration::from_millis(100)).await;
        })
    }
}
```

Здесь:

- в поз. (1) — теперь возвращается `Pin<Box<...>>>`;
- в поз. (2) — `Box::pin()` просто возвращает нам закрепленный блок.

На этом шаге код скомпилируется и заработает. Можно подумать, что все напасти миновали, но, к сожалению, это не так. Реализовать типаж `Observable` будет еще сложнее. Давайте рассмотрим исходный код типажа:

```
pub trait Observable {
    type Observer;
    fn update(&self);
    fn attach(&mut self, observer: Self::Observer);
    fn detach(&mut self, observer: Self::Observer);
}
```

Метод `update()` из `Observable` нужно сделать асинхронным, но это окажется труднее прежнего, потому что внутри `update()` приходится передавать `self` каждому из наблюдателей. Передача ссылки `self` внутри асинхронного метода без указания времени жизни для этой ссылки работать не станет. Кроме того, надо, чтобы каждый экземпляр `Observer` реализовал как `Send`, так и `Sync`, потому что наблюдать за обновлениями нам понадобится одновременно, для чего требуется, чтобы наши наблюдатели могли перемещаться между потоками. Окончательный вид кода нашего типажа `Observer` показан в листинге 8.4.

Листинг 8.4. Реализация асинхронного свойства `Observer`

```
pub trait Observer: Send + Sync { (1)
    type Subject;
    type Output;
    fn observe<'a>(
        &'a self, (2)
        subject: &'a Self::Subject, (3)
    ) -> Pin<Box<dyn Future<Output = Self::Output> + 'a + Send>>; (4)
} (5)
```

Здесь:

- в поз. (1) — добавление супертипов `Send + Sync`, гарантирующее возможность конкурентного использования наших наблюдателей в разных потоках;
- в поз. (2) — время жизни '`a` позволяет нам передавать `self` и `subject` в качестве ссылок;
- в поз. (3) — тут '`a` применяется к ссылке `self`;
- в поз. (4) — тут '`a` применяется к ссылке `subject`.
- в поз. (5) — '`a + Send` добавляется к границам типажа, чтобы разрешить перемещение между потоками и гарантировать, что возвращаемый фьючерс не переживет ни одной захваченной ссылки, использующей '`a`.

Код нашего обновленного типажа `Observable` показан в листинге 8.5.

Листинг 8.5. Реализация асинхронного типажа `Observable`

```
pub trait Observable {
    type Observer;
    fn update<'a>(          ← по аналогии с Observer для ссылок необходимо добавить время жизни
        &'a self,
    ) -> Pin<Box<dyn Future<Output = ()> + 'a + Send>>;
    fn attach(&mut self, observer: Self::Observer);
    fn detach(&mut self, observer: Self::Observer);
}
```

А теперь реализуем `Observable` для нашего `Subject` (листинг 8.6).

Листинг 8.6. Реализация асинхронного типажа Observable для Subject

```

pub struct Subject {
    observers: Vec<Weak<dyn Observer<Subject = Self, Output = ()>>>,
    state: String,
}

impl Subject {
    pub fn new(state: &str) -> Self {
        Self {
            observers: vec![],
            state: state.into(),
        }
    }

    pub fn state(&self) -> &str {
        self.state.as_ref()
    }
}

impl Observable for Subject {
    type Observer =
        Arc<dyn Observer<Subject = Self, Output = ()>>;
    fn update<'a>(&'a self) -> Pin<Box<dyn Future<Output = ()> + 'a + Send>>
        let observers: Vec<_> =
            self.observers.iter().flat_map(|o| o.upgrade()).collect();      (1)

        Box::pin(async move {                                              (2)
            futures::future::join_all(
                observers.iter().map(|o| o.observe(self)),           (3)
            )
            .await;                                              (4)
        })                                              (5)
    }

    fn attach(&mut self, observer: Self::Observer) {
        self.observers.push(Arc::downgrade(&observer));
    }
    fn detach(&mut self, observer: Self::Observer) {
        self.observers
            .retain(|f| !f.ptr_eq(&Arc::downgrade(&observer)));
    }
}

```

Здесь:

- в поз. (1) — составление списка наблюдателей для уведомления за пределами асинхронного контекста и сбор его в новый Vec;

- в поз. (2) — применение move к нашему асинхронному блоку для перемещения собранного списка наблюдателей в асинхронный блок;
- в поз. (3) — использование в этом месте `join_all()` обеспечивает конкурентность между нашими наблюдателями;
- в поз. (4) — функция `observe` каждого наблюдателя вызывается с одной и той же `self-ссылкой`;
- в поз. (5) — с помощью `.await` в нашем асинхронном блоке включается ожидание операции `join`.

Теперь наконец-то можно протестировать наш паттерн асинхронного наблюдателя (листинг 8.7).

Листинг 8.7. Тестирование нашего паттерна асинхронного наблюдателя

```
#[tokio::main]
async fn main() {
    let mut subject = Subject::new("some subject state");

    let observer1 = MyObserver::new("observer1");
    let observer2 = MyObserver::new("observer2");

    subject.attach(observer1.clone());
    subject.attach(observer2.clone());

    // ... do something here ...

    subject.update().await;
}
```

Выполнение этого кода приведет к выводу на экран следующего результата:

```
observed subject with state="some subject state" in observer1
observed subject with state="some subject state" in observer2
```

8.7. Смешивание синхронного и асинхронного кода

Экосистема асинхронного Rust быстро развивается, и `async` и `.await` уже поддерживаются многими библиотеками. И всё же бывают моменты, когда может потребоваться совместная работа синхронного и асинхронного кода. Два таких примера уже были показаны в предыдущем разделе, но давайте рассмотрим подобную ситуацию более подробно.

СОВЕТ

Как правило, смешивания синхронного и асинхронного кода следует избегать. Но в некоторых случаях имеет смысл добавить поддержку асинхронности, если она отсутствует, или же обновить код, использующий старые версии Tokio, не работающие с новым синтаксисом `async` и `.await`.

Наиболее распространенным сценарием, требующим смешения синхронного и асинхронного кода, является использование крейта, не поддерживающего асинхронность, — например, движка базы данных или сетевой библиотеки. Так, если нужно создать HTTP-сервис, использующий крейт `rocket`¹ с асинхронностью, может понадобиться чтение или запись в базу данных, которая всё еще не поддерживает асинхронность. Впрочем, добавление поддержки асинхронности в слишком сложные библиотеки может оказаться не самой лучшей тратой вашего времени, даже если это благое дело.

Для вызова синхронного кода из асинхронного контекста предпочтение следует отдавать использованию функции:

```
tokio::task::spawn_blocking()
```

которая принимает функцию и возвращает фьючерс. Вызов `spawn_blocking()` приводит к выполнению функции, предоставленной в очереди потоков, управляемой Tokio (что поддается настройке). Затем, как это обычно делается с любым асинхронным кодом, для фьючерса, возвращаемого `spawn_blocking()`, можно будет воспользоваться `.await`.

Давайте рассмотрим пример `spawn_blocking()` в действии, создав код, записывающий в файл в асинхронном режиме, а затемчитывающий записанное обратно в синхронном режиме:

```
use tokio::io::{self, AsyncWriteExt};  
  
async fn write_file(filename: &str) -> io::Result<()> {  
    let mut f = tokio::fs::File::create(filename).await?;  
    f.write(b"Hello, file!").await?;  
    f.flush().await?;  
  
    Ok(())  
}  
  
fn read_file(filename: &str) -> io::Result<String> {  
    std::fs::read_to_string(filename)  
}  
  
#[tokio::main]  
async fn main() -> io::Result<()> {  
    let filename = "mixed-sync-async.txt";  
    write_file(filename).await?;  
  
    let contents = tokio::task::spawn_blocking(|| read_file(filename)).await??; (3)  
  
    println!("File contents: {}", contents);
```

¹ См. <https://crates.io/crates/rocket/>.

```
tokio::fs::remove_file(filename).await?;
Ok(())
}
```

Здесь:

- в поз. (1) — запись "Hello, file!" в наш файл в асинхронном режиме;
- в поз. (2) — считывание содержимого нашего файла в строку, с ее возвращением в синхронном режиме;
- в поз. (3) — обратите внимание на наличие сдвоенного знака ??, обусловленного тем, что результат возвращает как `spawn_blocking()`, так и `read_file()`.

В этом коде в функции, вызываемой `spawn_blocking()`, выполняется синхронный ввод/вывод. Результата можно дожидаться точно так же, как и в любом другом обычном асинхронном блоке, разве что выполнение фактически происходит в отдельном блокирующем потоке, управляемом Tokio. Подробности реализации нас волновать не должны, за исключением того, что Tokio должно быть выделено достаточно большое количество потоков. В приведенном примере просто используются значения по умолчанию, но количество блокирующих потоков можно изменить с помощью построителя среды выполнения Tokio².

Синхронизация асинхронного кода

Иногда нужно синхронизировать асинхронный код — например, когда требуется осуществить обмен сообщениями между различными объектами. Поскольку блоки асинхронного кода могут работать в отдельных потоках выполнения, обмен данными между ними может усложняться, поскольку попытка получения доступа к данным ненадлежащим образом может привести к возникновению условий гонки (следует добавить, что язык Rust этого просто не допустит). В качестве легкого способа обмена данными можно действовать совместно используемое состояние за мьютексом, но Tokio предоставляет несколько более удачных методов совместного использования состояния.

В модуле `sync` Tokio содержится несколько инструментов для синхронизации асинхронного кода. В этой связи нельзя не упомянуть о канале `mpsc` — с несколькими производителями и одним потребителем (`multi-producer, single-consumer`), найти который можно в модуле `tokio::sync::mpsc`. Канал `mpsc` позволяет безопасно передавать сообщения от нескольких производителей одному потребителю в асинхронном контексте без необходимости применения явной блокировки (т. е. без введения мьютексов). Tokio предоставляет и другие типы каналов, включая `broadcast`, `oneshot` и `watch`.

С помощью каналов `mpsc` в асинхронном Rust можно создавать масштабируемые и конкурентные передающие сообщения без явной блокировки. Канал `mpsc` может быть неограниченным или ограниченным с фиксированной длиной, гарантуя тем самым противодействие производителям.

Каналы Tokio похожи на те, что можно найти в других фреймворках для работы с акторами или для обработки событий, за исключением того, что они относительно низкоуровневые и практически универсального назначения. Их создание больше похоже на программирование сокетов, чем на то, что может быть найдено в библиотеках акторов

² См. <http://mng.bz/j1WV>.

более высокого уровня. Подробнее об инструментарии синхронизации, имеющемся в Tokio, можно узнать из описания модуля `sync`³.

Для обратного сценария — использования асинхронного кода в синхронном коде, можно, как показано в предыдущем разделе, воспользоваться дескриптором среды выполнения с `block_on()`. Но всё же это весьма редкий вариант сценария, и его применения лучше избегать. Для более полного знакомства с этой темой обратитесь к документации по Tokio⁴.

8.8. Когда не стоит применять асинхронность?

Асинхронное программирование отлично подходит для приложений с большим объемом ввода/вывода — например, для сетевых сервисов. Ими могут быть HTTP-серверы, другие пользовательские сетевые сервисы или даже программы, инициирующие множество сетевых запросов (в отличие от ответов на запросы). Асинхронное программирование по причинам, рассмотренным в этой главе, привносит целый ряд сложностей, обычно не присущих синхронному программированию.

Смысл использования асинхронного программирования в качестве общей нормы приобретает только тех случаях, когда нужна конкурентность. Многие задачи программирования конкурентности не требуют, и для них лучше всего подходит синхронное программирование. В качестве примеров можно привести простой CLI-инструмент, выполняющий чтение из файла и запись в него или стандартный ввод/вывод, или простой HTTP-клиент, выполняющий несколько последовательных HTTP-запросов, — такой как `curl`. Если же у вас имеется инструмент наподобие `curl`, но от него требуется выполнение тысячи одновременных HTTP-запросов, тогда, конечно, следует воспользоваться асинхронным программированием.

Следует заметить, что добавление асинхронности на поздних стадиях подготовки проекта дается гораздо сложнее упреждающего создания программного средства с поддержкой асинхронности, поэтому следует тщательно продумать вопрос надобности в асинхронном программировании в каждом конкретном случае. Если брать в расчет исключительно фактор производительности, то между использованием и неиспользованием асинхронности для простых последовательных и неконкурентных задач практически нет никакой разницы, но Tokio всё же вносит небольшие, но заметные накладные расходы, которые вряд ли будут существенными для большинства намечаемых целей.

8.9. Трассировка и отладка асинхронного кода

Для любого достаточно сложного сетевого приложения очень важно оснащать код инструментальными средствами с целью измерения его производительности и отладки, позволяющей устранять возникающие проблемы. Проект Tokio предостав-

³ См. <https://docs.rs/tokio/latest/tokio/sync/index.html>.

⁴ См. <https://tokio.rs/tokio/topics/bridging>.

ляет для этих целей трассировочный крейт `tracing5`, поддерживающий стандарт `OpenTelemetry6`, позволяющий объединять его с рядом популярных сторонних инструментов трассировки и телеметрии, а также выдавать трассировку в виде журналов.

Включение трассировки в Tokio также приводит к разблокировке `tokio-console7` — CLI-инструмента, похожего на программу `top`, с которой вы, возможно, знакомы по большинству UNIX-систем. Инструмент `tokio-console` позволяет в режиме реального времени анализировать асинхронный Rust-код, созданный на базе Tokio. Замечательно! Но при всех своих удобствах, `tokio-console` не обладает широким спектром возможностей и используется в основном в качестве средства отладки, а вам в большинстве сред, скорее всего, понадобится выводить трассировку в журналы или задействовать `OpenTelemetry`. Инструмент `tokio-console` также нельзя подключить к программе, которая не была скомпилирована для этого инструмента заранее.

Включение трассировки требует настройки подписчика, которому будут отправляться данные трассировки. Кроме того, для эффективного использования трассировки, надо в тех точках, где желательно проводить измерения, оснастить функции соответствующим инструментарием. Сделать это можно с помощью макроса `#[tracing::instrument]`. Трассировка может производиться на разных уровнях и с различными опциями, подробное описание которых можно найти в документации по трассировке⁸.

Давайте для демонстрации трассировки с помощью инструмента `tokio-console`, требующего определенной настройки и шаблона, напишем небольшую программу. В этой программе мы предусмотрим три разные функции бездействия, каждая из которых будет оснащена инструментарием, и все они станут одновременно работать в бесконечном цикле:

```
use tokio::time::{sleep, Duration};

#[tracing::instrument]
async fn sleep_1s() {
    sleep(Duration::from_secs(1)).await;
}

#[tracing::instrument] (1)
async fn sleep_2s() {
    sleep(Duration::from_secs(2)).await;
}
```

⁵ См. <https://crates.io/crates/tracing>.

⁶ См. <https://opentelemetry.io/>.

⁷ См. <https://github.com/tokio-rs/console>.

⁸ См. <https://docs.rs/tracing/latest/tracing/index.html>.

```

#[tracing::instrument]
async fn sleep_3s() {
    sleep(Duration::from_secs(3)).await;
}

#[tokio::main]
async fn main() {
    console_subscriber::init();                                (2)

    loop {
        tokio::spawn(sleep_1s());                            (3)
        tokio::spawn(sleep_2s());
        sleep_3s().await;                                    (4)
    }
}

```

Здесь:

- в поз. (1) — применение для оснащения трех наших функций бездействия инструментарием макрояа `instrument` из крейта `tracing`;
- в поз. (2) — инициализация подписчика консоли в нашей функции `main()` с целью отправки трассировок;
- в поз. (3) — мы активируем `sleep 1` и `sleep 2`, а затем забудем о них, после чего заблокируем `sleep 3`;
- в поз. (4) — `sleep 3` блокируется до истечения 3 секунд, после чего процесс бесконечно повторяется.

Чтобы включить в Tokio функцию трассировки, необходимо к нашим зависимостям добавить следующий код:

```
[dependencies]
tokio = { version = "1", features = ["full", "tracing"] }          (1)
tracing = "0.1"
console-subscriber = "0.1"
```

- Здесь в поз. (1) — при выборе режима `"full"` функция трассировки в Tokio не включается, и ее необходимо включать явным образом.

Чтобы скомпилировать нашу программу и запустить ее на выполнение, предварительно нужно установить `tokio-console` с помощью команды:

```
cargo install tokio-console
```

Надо также учесть, что для включения в Tokio нестабильных функций трассировки, предназначенных для `tokio-console`, компиляцию следует проводить с настройкой:

```
RUSTFLAGS="--cfg tokio_unstable"
```

Это может быть сделано путем запуска программы с этой настройкой непосредственно из Cargo:

```
RUSTFLAGS="--cfg tokio_unstable" cargo run
```

А запустив программу, можно будет запустить и tokio-console, что приведет к выводу на экран информации, показанной на рис. 8.6. Кроме того, в дополнение к мониторингу задач могут отслеживаться и ресурсы (рис. 8.7). Вы сможете также углубляться в отдельные задачи и даже увидеть гистограмму времени опроса (рис. 8.8).

Warn	ID	State	Name	Total	Busy	Idle	Polls	Target	Location	Fields
	5	idle		2.0039s	497.2910µs	2.0034s	2	tokio::task	src/main.rs:24:9	kind=task
	4	idle		2.0035s	727.5410µs	2.0027s	2	tokio::task	src/main.rs:24:9	kind=task
	7	idle		2.0029s	557.2490µs	2.0023s	2	tokio::task	src/main.rs:24:9	kind=task
	9	idle		2.0024s	425.3740µs	2.0019s	2	tokio::task	src/main.rs:24:9	kind=task
	12	idle		2.0023s	494.2080µs	2.0018s	2	tokio::task	src/main.rs:24:9	kind=task
	2	idle		2.0022s	540.9580µs	2.0017s	2	tokio::task	src/main.rs:24:9	kind=task
	1	idle		1.0051s	1.6637ms	1.0034s	2	tokio::task	src/main.rs:23:9	kind=task
	3	idle		1.0036s	1.2935ms	1.0024s	2	tokio::task	src/main.rs:23:9	kind=task
	6	idle		1.0035s	507.4160µs	1.0030s	2	tokio::task	src/main.rs:23:9	kind=task
	8	idle		1.0027s	270.2910µs	1.0024s	2	tokio::task	src/main.rs:23:9	kind=task
	11	idle		1.0025s	618.1660µs	1.0019s	2	tokio::task	src/main.rs:23:9	kind=task
	10	idle		1.0021s	281.5830µs	1.0019s	2	tokio::task	src/main.rs:23:9	kind=task
	13	idle		989.1582ms	213.0000µs	988.9452ms	1	tokio::task	src/main.rs:23:9	kind=task
	14	idle		988.9660ms	123.1250µs	988.8429ms	1	tokio::task	src/main.rs:24:9	kind=task

Рис. 8.6. Показываемые в tokio-console запущенные задачи

ID	Parent	Kind	Total	Target	Type	Vis	Location	Attributes
33	n/a	Timer	977.8489ms	tokio::time::driver::sleep	Sleep	✓	src/main.rs:15:5	duration=3001ms
32	n/a	Timer	977.8025ms	tokio::time::driver::sleep	Sleep	✓	src/main.rs:10:5	duration=2001ms
31	n/a	Timer	977.8845ms	tokio::time::driver::sleep	Sleep	✓	src/main.rs:5:5	duration=1001ms
30	n/a	Timer	1.0024s	tokio::time::driver::sleep	Sleep	✓	src/main.rs:5:5	duration=1001ms
29	n/a	Timer	3.0027s	tokio::time::driver::sleep	Sleep	✓	src/main.rs:15:5	duration=3001ms
28	n/a	Timer	2.0013s	tokio::time::driver::sleep	Sleep	✓	src/main.rs:10:5	duration=2001ms
27	n/a	Timer	2.0017s	tokio::time::driver::sleep	Sleep	✓	src/main.rs:10:5	duration=2001ms
26	n/a	Timer	3.0017s	tokio::time::driver::sleep	Sleep	✓	src/main.rs:15:5	duration=3001ms
25	n/a	Timer	1.0022s	tokio::time::driver::sleep	Sleep	✓	src/main.rs:5:5	duration=1001ms
24	n/a	Timer	2.0007s	tokio::time::driver::sleep	Sleep	✓	src/main.rs:10:5	duration=2001ms
23	n/a	Timer	1.0015s	tokio::time::driver::sleep	Sleep	✓	src/main.rs:5:5	duration=1001ms
22	n/a	Timer	3.0022s	tokio::time::driver::sleep	Sleep	✓	src/main.rs:15:5	duration=3001ms
21	n/a	Timer	2.0026s	tokio::time::driver::sleep	Sleep	✓	src/main.rs:10:5	duration=2001ms
20	n/a	Timer	3.0027s	tokio::time::driver::sleep	Sleep	✓	src/main.rs:15:5	duration=3001ms
19	n/a	Timer	1.0015s	tokio::time::driver::sleep	Sleep	✓	src/main.rs:5:5	duration=1001ms
18	n/a	Timer	1.0023s	tokio::time::driver::sleep	Sleep	✓	src/main.rs:5:5	duration=1001ms
17	n/a	Timer	3.0021s	tokio::time::driver::sleep	Sleep	✓	src/main.rs:15:5	duration=3001ms
16	n/a	Timer	2.0020s	tokio::time::driver::sleep	Sleep	✓	src/main.rs:10:5	duration=2001ms
15	n/a	Timer	3.0027s	tokio::time::driver::sleep	Sleep	✓	src/main.rs:15:5	duration=3001ms
14	n/a	Timer	2.0022s	tokio::time::driver::sleep	Sleep	✓	src/main.rs:10:5	duration=2001ms
13	n/a	Timer	1.0019s	tokio::time::driver::sleep	Sleep	✓	src/main.rs:5:5	duration=1001ms

Рис. 8.7. Показанное в tokio-console использование ресурсов

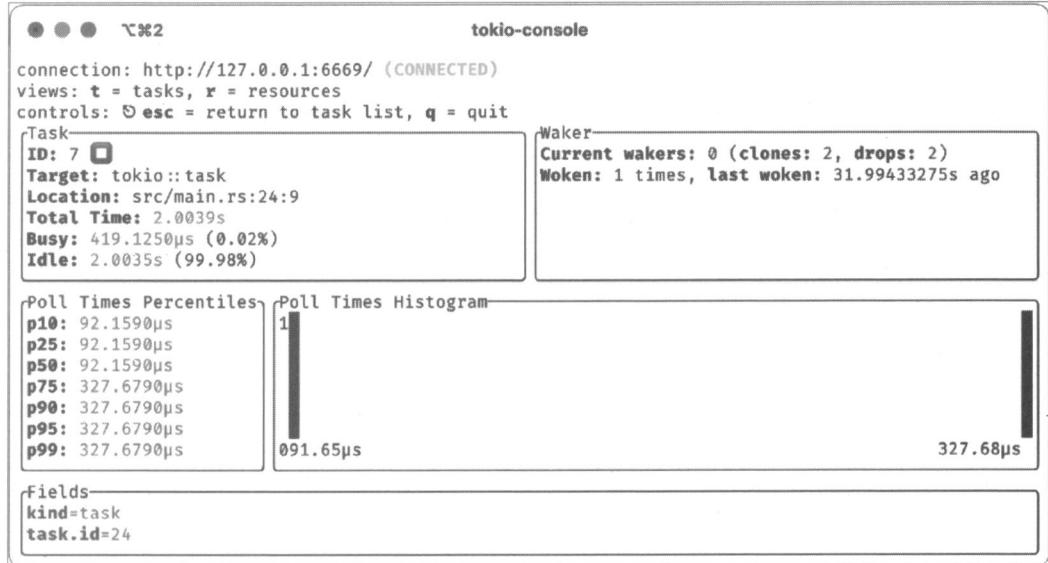


Рис. 8.8. Отдельно взятая задача с гистограммой времени опроса

Таким образом, используя инструмент `tokio-console`, мы можем в реальном времени видеть состояние задач, множество связанных с каждой из них показателей, включенные дополнительные метаданные и даже местоположение исходных файлов. При этом `tokio-console` позволяет просматривать по отдельности как реализованные нами задачи, так и ресурсы Tokio. Все эти данные также будут доступны в трассировках, отправленных в другой приемник, — например, в файл журнала или в коллектор OpenTelemetry.

8.10. Работа с асинхронностью при тестировании

И последний вопрос, рассматриваемый в этой главе, — тестирование асинхронного кода. При создании модульных или интеграционных тестов для асинхронного кода используются две стратегии:

- создание и разрушение асинхронной среды выполнения для каждого отдельно взятого теста;
- повторное использование одной или нескольких асинхронных сред выполнения в отдельно взятых тестах.

Чаще всего предпочтительнее создавать и разрушать среду выполнения для каждого теста, но из этого правила есть исключения, когда целесообразнее использовать среду выполнения повторно. В частности, повторное использование среды выполнения имеет смысл, при наличии множества (т. е. сотен или тысяч) тестов.

Для повторного использования в тестах среды выполнения можно воспользоваться крейтом `lazy_static`, который весьма подробно рассматривался в главе 6. Имеющаяся в Rust среда тестирования запускает тесты параллельно в потоках, требующих,

как уже было показано в этой главе ранее, корректной обработки с применением `tokio::runtime::Handle`.

Иногда можно просто применить макрос `#[tokio::test]`, работающий точно так же, как и `#[test]`, — за исключением того, что он предназначен для асинхронных функций. Имеющийся в Tokio тестовый макрос берет на себя настройку среды выполнения теста, поэтому свой асинхронный модульный или интеграционный тест вы можете написать точно так же, как это обычно делается при написании любого другого теста. Чтобы продемонстрировать такой подход, рассмотрим следующую функцию бездействия в течение 1 секунды:

```
async fn sleep_1s() {
    sleep(Duration::from_secs(1)).await;
}
```

Тест можно написать, применив макрос `#[tokio::test]`, управляющий созданием для нас среды выполнения:

```
#[tokio::test]
async fn sleep_test() {
    let start_time = Instant::now();
    sleep(Duration::from_secs(1)).await;
    let end_time = Instant::now();
    let seconds = end_time
        .checked_duration_since(start_time)
        .unwrap()
        .as_secs();
    assert_eq!(seconds, 1);
}
```

Этот тест выполняется обычным образом, как и любой другой тест, разве что его выполнение проходит в асинхронном контексте. Допускается, конечно, и самостоятельное управление средой выполнения, но, как уже ранее упоминалось, при этом не следует забывать, что тесты Rust будут выполняться параллельно.

И наконец, средой выполнения Tokio предоставляется крейт `tokio-test`⁹, включаемый добавлением функциональной особенности "test-util" (которая не включается флагом функциональной особенности "full"). Этот крейт содержит целый ряд вспомогательных инструментов для имитации асинхронных задач, а также несколько удобных макросов, предназначенных для использования с Tokio.

Резюме

- Для программ на языке Rust предоставляется сразу несколько реализаций асинхронной среды выполнения, но для решения большинства задач наиболее подходящей из них является Tokio.

⁹ См. https://docs.rs/tokio-test/latest/tokio_test/.

- Асинхронное программирование требует особого потока управления, и наш код должен подчиняться планировщику среды выполнения, чтобы дать ему возможность переключать задачи. Этой уступки можно добиться, воспользовавшись `.await` или непосредственно порождая фьючерсы с помощью `tokio::task::spawn()`.
- Асинхронные блоки кода (например, функции) обозначаются ключевым словом `async` и всегда возвращают фьючерсы.
- Фьючерс можно выполнить с помощью инструкции `.await`, но только лишь в асинхронном контексте (т. е. в асинхронном блоке кода).
- Асинхронные блоки ленивы и не будут выполняться, пока не будет выполнен вызов `.await` или пока они не будут порождены явным образом. Это является особым отличием от большинства других асинхронных реализаций.
- Использование `tokio::select! {}` или `tokio::join!()` позволяет вводить явно заданную конкурентность.
- Порождение задач с помощью `tokio::task::spawn()` дает возможность вводить конкурентность и параллелизм.
- При желании выполнять блокирующие операции их следует порождать с помощью `tokio::task::spawn_blocking()`.
- Крейт `tracing` предоставляет простой способ оснащения трассировки инструментарием и отправки телеметрии в журналы или в коллекторы OpenTelemetry.
- Для отладки асинхронных программ можно воспользоваться `tokio-console` с трассировкой.
- Tokio для проведения модульных и интеграционных тестов предоставляет макросы тестирования, обеспечивающие необходимую для тестирования среду выполнения. Инструменты имитации и утверждения, используемые с Tokio, предоставляются крейтом `tokio_test`, включаемым с помощью выставления в Tokio флага функциональной особенности "test-util".

9

Создание сервиса HTTP REST API

В этой главе:

- выбор используемого веб-фреймворка;
- разработка API;
- моделирование наших данных;
- реализация API;
- надлежащая обработка ошибок.

В процессе создания веб-сервиса с применением асинхронного Rust мы практически воспользуемся здесь многими знаниями, усвоенными нами при изучении предыдущих глав. А для полноты картины в главе 10 рассмотрим создание API клиента.

Основное внимание мы уделим окончательному коду, а не обсуждению синтаксиса, шаблона и альтернативных реализаций. Я уверен, что максимальную пользу можно извлечь из знакомства только с полностью работоспособным примером. Различные описания «как сделать», встречающиеся в Интернете (и в других местах), имеют тенденцию опускать многие детали полной реализации и замалчивать основные сложности, поэтому я сделаю всё возможное, чтобы показать, чего не хватает в этом примере, и куда можно двигаться дальше. Я не стану подробно обсуждать темы развертывания, балансировки нагрузки, управления состоянием, управления кластером или высокой доступности, поскольку они не вписываются в тематику этой книги и не связаны напрямую с языком Rust.

Таким образом, к концу главы мы создадим веб-сервис API, использующий для управления состоянием базу данных и обеспечивающий тем самым критически важные функции практически всех существующих веб-сервисов: создание, чтение, обновление, удаление и перечисление элементов в базе данных. Мы также смоделируем CRUD-приложение todo («что сделать»), поскольку они довольно часто используются в качестве примеров в учебных целях. После этого вы сможете воспользоваться его кодом в качестве шаблона или стартового проекта для своей будущей разработки. А теперь за работу!

9.1. Выбор веб-фреймворка

При работе над книгой стало заметно, что асинхронный ландшафт Rust довольно сильно изменился — особенно в отношении инструментов и библиотек, доступных для работы с асинхронным Rust. В основном это были позитивные изменения, и, в частности, я был весьма впечатлен прогрессом Tokio и связанных с этим средством проектов.

Для написания веб-сервиса я рекомендую воспользоваться фреймворком `axum`, представляющим собой часть более крупного проекта Tokio. Фреймворк `axum` по сравнению с другими фреймворками не может похвастаться богатой функциональностью, но благодаря гибкому API и реализации, преимущественно обходящейся без применения макросов, обладает большим потенциалом. Он относительно просто поддается интеграции с другими библиотеками и инструментами, а его простой API позволяет быстро и легко приступить к работе. Фреймворк `axum` основан на `Tower`¹ — библиотеке, предоставляющей абстракции для создания сетевых сервисов, и на `hyper`² — библиотеке, предоставляющей реализацию HTTP-клиента и сервера как для HTTP/1, так и для HTTP/2 (погрузиться в специфику HTTP/2 позволяет книга «HTTP/2 в действии» Барри Полларда³).

Лучшее, чем может похвастаться `axum`, — отсутствием навязывания с его стороны многочисленных шаблонов или приемов использования. Детальное освоение этого фреймворка предполагает изучение библиотеки `Tower`, но для простых задач виникать в подробности необязательно, — базовый веб-сервис можно быстро настроить без предварительных существенных затрат времени на изучение собственно веб-фреймворка. Для создания производственных сервисов в `axum` включена поддержка трассировки и контрольных показателей, для использования которых требуется всего лишь небольшая настройка.

Стоит упомянуть...

Нельзя не сказать несколько слов и о двух других фреймворках: о `Rocket`⁴ — веб-фреймворке, пытающемся стать для Rust чем-то вроде Ruby on Rails, и о `Actix`⁵ — одном из самых ранних веб-фреймворков для Rust. И `Rocket`, и `Actix` характеризуются одним и тем же недостатком — в них, чтобы скрыть детали реализации, слишком сильно ориентация на макросы. А вот в `axum` для его основного API макросы не применяются, что (по моему скромному мнению) делает его намного приятнее для работы и проще для понимания.

К чести и `Rocket`, и `Actix`, надо отметить, что они активно применялись в Rust до появления в нем стабильной версии типажа `Future` и синтаксиса `async/await`, почему и требовалось использование макросов. Следует упомянуть и о том, что оба этих фреймворка в своих более поздних версиях успешно снизили свою зависимость от макросов.

¹ См. <https://github.com/tower-rs/tower>.

² См. <https://hyper.rs/>.

³ См. <https://dmkpress.com/catalog/computer/web/978-5-97060-925-5/>.

⁴ См. <https://rocket.rs/>.

⁵ См. <https://actix.rs/>.

9.2. Построение архитектуры

Для нашего веб-сервиса мы выстроим типичную архитектуру веб-уровня, состоящую как минимум из трех компонентов: балансировщика нагрузки, самого веб-сервиса и сервиса с сохранением состояния (т. е. базы данных). Реализацией балансировщика нагрузки мы заниматься не станем (предположим, что он уже имеется или предоставлен), а для базы данных воспользуемся SQLite, — впрочем, при серьезном практическом применении вам, скорее всего, захочется воспользоваться другой базой данных SQL — например, PostgreSQL. Выстраиваемая архитектура веб-сервиса показана на рис. 9.1.



Рис. 9.1. Архитектура веб-сервиса

Как можно здесь видеть, наш API-сервис способен масштабироваться по горизонтали путем простого добавления дополнительных экземпляров сервиса. Каждый экземпляр API-сервиса получает запросы от балансировщика нагрузки и независимо взаимодействует с базой данных для сохранения и извлечения состояния.

Разрабатываемое нами приложение должно принимать свою конфигурацию из среды, поэтому параметры конфигурации мы передадим ему с применением переменных среды. Вместо этого можно было бы воспользоваться синтаксическим анализом содержимого командной строки или файлом конфигурации, но переменные среды очень удобны — особенно при развертывании в таких контекстах, как системы оркестровки кластера. В нашем случае будет задействована только пара параметров конфигурации: один — для указания базы данных, а другой — для настройки ведения журнала. Эти параметры мы рассмотрим чуть позже.

Конфигурация для каждого экземпляра нашего API-сервиса в большинстве случаев будет идентичной, хотя могут складываться и особые обстоятельства, при которых потребуется указывать параметры, уникальные для каждого экземпляра сервиса, — например, информацию о местоположении или IP-адрес для привязки. На практике

обычно осуществляется привязка к адресу 0.0.0.0, который относится ко всем интерфейсам и эффективно делегирует обработку всех подробностей сетевому стеку операционной системы (с возможностью настройки по мере необходимости).

9.3. Проектирование API

Для нашего сервиса мы смоделируем обычное приложение todo («что сделать»). Скорее всего, с таким приложением вы уже знакомы, и поэтому для todo-записей нами будут реализованы только конечные точки CRUD: создания (create), чтения (read), обновления (update) и удаления (delete), а также конечная точка для вывода полного списка дел. К этому мы добавим еще и конечные точки проверки работоспособности и готовности. Как показано в табл. 9.1, API-маршруты мы разместим под путем /v1.

Таблица 9.1. Маршруты API-сервиса

Путь	HTTP-метод	Действие	Тело запроса	Ответ
v1/todos	GET	Вывод списка	Не указывается	Список всех дел (todo)
v1/todos	POST	Создание	Новый todo-объект	Только что созданный todo-объект
v1/todos/:id	GET	Чтение	Не указывается	Только что созданный todo-объект
v1/todos/:id	PUT	Обновление	Обновленный todo-объект	Только что созданный todo-объект
v1/todos/:id	DELETE	Удаление	Новый todo-объект	Только что созданный todo-объект

Для путей чтения, обновления и удаления мы используем параметр пути к идентификатору каждого todo-объекта, обозначенный в указанных в табл. 9.1 путях токеном :id. Для проверки работоспособности и готовности добавим конечные точки, показанные в табл. 9.2.

Таблица 9.2. Конечные точки состояния API-сервиса

Путь	HTTP-метод	Ответ
/alive	GET	Возвращение 200 с ok в случае успеха
/ready	GET	Возвращение 200 с ok в случае успеха

Теперь, составив описание API, давайте рассмотрим инструменты и библиотеки, которые будут использоваться для его создания, чему и посвящен следующий раздел.

9.4. Библиотеки и инструменты

Для выполнения основной и самой трудной части работы по созданию сервиса мы воспользуемся уже существующими крейтами. Нам вообще не потребуется создавать большой объем программного кода — в основном наша работа в ходе создания сервиса будет заключаться в интеграции уже существующих компонентов. Естественно, при этом мы должны уделять пристальное внимание тому, как именно объединяются различные компоненты, но, к счастью для нас, система типов Rust упрощает эту задачу, сообщая с помощью выдаваемых компилятором ошибок, когда что-то идет не так.

Инициализацию проекта можно выполнить с помощью команды:

```
cargo new api-server
```

после чего вы сможете приступить к добавлению необходимых нам крейтов с использованием команды:

```
cargo add ...
```

Перечень таких крейтов (зависимостей API-сервиса) и их функций приведен в табл. 9.3.

Таблица 9.3. Зависимости API-сервиса

Название	Функции	Описание
axum	Стандартные	Веб-фреймворк
chrono	serde	Библиотека даты/времени с функцией <code>serde</code>
serde	derive	Библиотека сериализации/десериализации с функцией <code>#[derive(...)]</code>
serde_json	Стандартные	JSON-сериализация/десериализация для крейта <code>serde</code>
sqlx	runtime-tokio-rustls, sqlite, chrono, macros	Асинхронный набор инструментов SQL для SQLite, MySQL и PostgreSQL
tokio	macros, rt-multi-thread	Асинхронная среда выполнения, используемая с <code>axum</code> и <code>sqlx</code>
tower-http	trace, cors	Предоставляет промежуточное программное обеспечение HTTP для <code>axum</code> , в частности трассировку и CORS.
tracing	Стандартные	Библиотека асинхронной трассировки
tracing-subscriber	env-filter	Позволяет подписываться на трассировку данных в крейтах, использующих трассировку

ПРИМЕЧАНИЕ

Версии зависимостей в табл. 9.3 не указаны. Их можно найти в файле `Cargo.toml` из листингов исходного кода для этой книги.

Для зависимостей с функциональными особенностями можно при использовании команды `cargo add` воспользоваться флагом `--feature`. Например, чтобы добавить `axum` со стандартными функциями, запускается команда:

```
cargo add axum
```

а для библиотеки `SQLx` — команда:

```
cargo add sqlx --features runtime-tokio-rustls,sqlite,chrono,macros
```

Для нашего проекта можно просто скопировать файл `Cargo.toml` из исходного кода, сопровождающего книгу.

ПРИМЕЧАНИЕ

Напомню, что код примеров, включенных в книгу, доступен для загрузки с веб-сайта издательства Manning⁶, а также с GitHub⁷. Копия этого кода может быть также скачана вами на свой компьютер с помощью следующей Git-команды:

```
$ git clone https://github.com/brndnmthws/code-like-a-pro-in-rust-book
```

Кроме того, вы можете попробовать применить инструмент `sqlx-cli`⁸, устанавливаемый с помощью команды:

```
cargo install sqlx-cli
```

Этот инструмент позволяет создавать базы данных, запускать миграции и удалять базы данных. Для получения дополнительных сведений по этому инструменту нужно после установки запустить команду:

```
sqlx --help
```

Для запуска кода этот инструмент не понадобится, но при желании добиться от `SQLx` чего-то большего, он, конечно же, пригодится.

Для удобства всё, указанное в табл. 9.3, может быть установлено в один прием запуском следующих команд:

```
cargo add axum
cargo add chrono --features serde
cargo add serde --features derive
cargo add serde_json
cargo add sqlx --features runtime-tokio-rustls,sqlite,chrono,macros
cargo add tokio --features macros,rt-multi-thread
cargo add tower-http --features trace,cors
cargo add tracing
cargo add tracing-subscriber --features env-filter
cargo install sqlx-cli
```

После их выполнения ваш файл `Cargo.toml` приобретет вид, показанный в листинге 9.1.

⁶ См. <https://www.manning.com/books/code-like-a-pro-in-rust>.

⁷ См. <https://github.com/brndnmthws/code-like-a-pro-in-rust-book>.

⁸ См. <https://crates.io/crates/sqlx-cli>.

Листинг 9.1. Файл Cargo.toml API-сервиса

```
[package]
name = "api-service"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/
➥ reference/manifest.html

[dependencies]
axum = "0.6.18"
chrono = { version = "0.4.26", features = ["serde"] }
serde = { version = "1.0.164", features = ["derive"] }
serde_json = "1.0.99"
sqlx = { version = "0.6.3", features = ["runtime-tokio-rustls", "sqlite",
➥ "chrono", "macros"] }
tokio = { version = "1.28.2", features = ["macros", "rt-multi-thread"] }
tower-http = { version = "0.4.1", features = ["trace", "cors"] }
tracing = "0.1.37"
tracing-subscriber = { version = "0.3.17", features = ["env-filter"] }
```

Настроив зависимости, можно приступать к написанию кода.

ПРИМЕЧАНИЕ

На практике зависимости, скорее всего, будут добавляться и изменяться в процессе разработки, так что не стоит воспринимать всё здесь сказанное как предложение заранее настраивать все зависимости. Как мне нравится говорить, *software is soft* (программы — вещь податливая), поэтому всегда подгоняйте их под свой вкус (не исключая и приводимые здесь примеры).

9.5. Создание шаблонов приложений

Точка входа в наше приложение, описанная файле `main.rs`, содержит небольшой объем шаблонного кода и необходимые настройки. Здесь мы сделаем следующее:

- объявим нашу основную точку входа;
- инициализируем трассировку и ведение журнала;
- создадим и инициализируем наше подключение к базе данных;
- запустим все необходимые миграции базы данных;
- определим маршруты для нашего API;
- запустим сам сервис.

9.5.1. Функция `main()`

Давайте начнем с рассмотрения функции `main()` (листинг 9.2).

Листинг 9.2. Функция main() из файла src/main.rs API-сервиса

```

#[tokio::main]
async fn main() {
    init_tracing();                                     (1)

    let dbpool = init_dbpool().await
        .expect("couldn't initialize DB pool");          (2)

    let router = create_router(dbpool).await;            (3)

    let bind_addr = std::env::var("BIND_ADDR")
        .unwrap_or_else(|| "127.0.0.1:3000".to_string()); (4)

    axum::Server::bind(&bind_addr.parse().unwrap())
        .serve(router.into_make_service())                (5)
        .await
        .expect("unable to start server")                 (6)
}

```

Здесь:

- в поз. (1) — инициализация трассировки и ведение журнала для нашего сервиса и его зависимостей;
- в поз. (2) — инициализация пула базы данных;
- в поз. (3) — создание основного сервиса приложения и его маршрутов;
- в поз. (4) — извлечение адреса привязки из переменной среды BIND_ADDR или использование значения по умолчанию 127.0.0.1:3000;
- в поз. (5) — разбор адреса привязки для превращения его в адрес сокета;
- в поз. (6) — создание сервиса и запуск HTTP-сервера.

Содержимого в нашей функции `main()` немного, но чтобы понять, что происходит, в нем нужно как следует разобраться. Однако сначала следует заметить, что здесь для инициализации среды выполнения Tokio используется принадлежащий этому средству макрос `tokio::main`, скрывающий от нас сложность происходящего, — например, установку количества рабочих потоков.

СОВЕТ

Tokio прочитает значение переменной среды `TOKIO_WORKER_THREADS` и, если оно указано, установит соответствующее количество рабочих потоков.

Для более сложных сценариев может потребоваться самостоятельное создание экземпляра среды выполнения Tokio и его соответствующая настройка с помощью `tokio::runtime::Builder`.

9.5.2. Инициализация трассировки: `init_tracing()`

А теперь рассмотрим инициализацию трассировки в `init_tracing()` (листинг 9.3).

Листинг 9.3. Функция `init_tracing()` из файла `src/main.rs` API-сервиса

```
fn init_tracing() {
    use tracing_subscriber::{
        filter::LevelFilter, fmt, prelude::*, EnvFilter
    };

    let rust_log = std::env::var(EnvFilter::DEFAULT_ENV)           (1)
        .unwrap_or_else(|| "sqlx=info,tower_http=debug,info".to_string());

    tracing_subscriber::registry()                                (2)
        .with(fmt::layer())                                       (3)
        .with(
            EnvFilter::builder()                                 (4)
                .with_default_directive(LevelFilter::INFO.into())
                .parse_lossy(rust_log),
        )
        .init();
}
```

Здесь:

- в поз. (1) — извлечение переменной среды `RUST_LOG` и, если она не определена, предоставление значения по умолчанию;
- в поз. (2) — возвращение исходного глобального реестра;
- в поз. (3) — объявление уровня форматирования, обеспечивающего удобное для восприятия форматирование трассировки;
- в поз. (4) — создание фильтра среды с исходным уровнем журналирования, установленным в `info`, или же использование значения, предоставленного `RUST_LOG`.

Если нужно увидеть в журнале полезные сообщения, то без инициализации трассировки будет просто не обойтись. Скорее всего, вам не захочется включать все сообщения трассировки, и вы решите оставлять только полезные, поэтому здесь явным образом включаются сообщения уровня отладки для `tower_http::*` и сообщения уровня информации для `sqlx::*`. Можно было бы также добавить и свои собственные трассировки, но для наших нужд более чем достаточно тех, что включены в используемые нами крейты.

Определить, какие именно трассировки могут быть включены, бывает нелегко, но установкой `RUST_LOG=trace` можно включить сразу все. Это может привести к генерации слишком большого объема выходных данных журнала, поэтому если в них нет особой потребности, то не стоит даже пытаться давать такую установку в производственных средах. Фильтр среды `EnvFilter` совместим с `env_logger`, используемым

многими другими крейтами Rust, что позволяет поддерживать совместимость и узнаваемость в экосистеме Rust.

9.5.3. Инициализация пула базы данных: `init_dbpool()`

Для управления состоянием мы воспользуемся пулом базы данных, позволяющим к ней подключаться. Пул подключений дает возможность получать и повторно использовать подключения к базе данных, не создавая новых подключений для каждого запроса, что обеспечивает нам право провести небольшую, но весьма полезную оптимизацию. Параметры пула подключений зависят от базы данных и могут быть настроены по мере необходимости, но для нашего примера мы будем придерживаться параметров по умолчанию. Стоит также заметить, что при всей полезности этого пула особой необходимости в нем мы не испытываем, поскольку задействуем базу SQLite, работающую (в отличие от подключений по сети к таким базам данных, как MySQL или PostgreSQL) в том же процессе в фоновых потоках, управляемых библиотекой SQLite. Давайте рассмотрим код `init_dbpool()`, показанный в листинге 9.4.

Листинг 9.4. Функция `init_dbpool()` из файла `src/main.rs` API-сервиса

```
async fn init_dbpool() -> Result<sqlx::Pool<sqlx::Sqlite>, sqlx::Error> {
    use sqlx::sqlite::{SqliteConnectOptions, SqlitePoolOptions};
    use std::str::FromStr;

    let db_connection_str =
        std::env::var("DATABASE_URL")
            .unwrap_or_else(|| "sqlite:db.sqlite".to_string()); (1)

    let dbpool = SqlitePoolOptions::new()
        .connect_with(SqliteConnectOptions::from_str(&db_connection_str)?
        .create_if_missing(true)) (2)
        .await
        .expect("can't connect to database");

    sqlx::migrate!() (3)
        .run(&dbpool) (4)
        .await
        .expect("database migration failed");

    Ok(dbpool)
}
```

Здесь:

- в поз. (1) — попытка чтения значения переменной среды `DATABASE_URL` или значения по умолчанию `sqlite:db.sqlite`, если значение переменной не определено (что приводит к открытию файла с именем `db.sqlite` в текущем рабочем каталоге);

- в поз. (2) — требования к движку базы данных создать эту базу при подключении, если она еще не была создана;
- в поз. (3) — выполнение всех миграций, необходимых после подключения к базе данных;
- в поз. (4) — недавно созданный пул базы данных может быть непосредственно передан в ту SQLx, которая получает подключения из пула.

Базы данных — непростая тема, которая не вписывается в формат нашей книги, но я всё же опишу то, что происходит в показанном здесь коде:

- строка подключения извлекается из переменной среды `DATABASE_URL`, принимая значение по умолчанию `sqlite:db.sqlite`, что приводит к открытию файла `db.sqlite` в текущем рабочем каталоге. Можно было бы (теоретически) поддерживать несколько движков баз данных, но для этого — в зависимости от движка, указанного в `DATABASE_URL`, — придется тщательно настроить используемые SQL-инструкции. При практическом же применении следует просто выбрать одну базу данных и убедиться, что код с ней работает, поскольку у каждой базы данных есть свои особенности и различия, даже если они, выражаясь технически, используют один и тот же язык SQL;
- движку SQLite при установке для `SqliteConnectOptions` опции `create_if_missing(true)` разрешается при подключении создавать базу данных. Если база данных еще не создана, SQLx генерирует ее для нас с помощью `CREATE DATABASE IF NOT EXISTS ...`, поэтому беспокоиться о создании базы данных нет никаких причин. Такая возможность предоставляется для удобства и должна быть относительно безопасной, но может быть востребована не во всех контекстах;
- SQLx предоставляет API миграции, слишком подробного рассказа о котором здесь не последует, но если вам уже приходилось использовать какие-либо другие веб-фреймворки, то там, вероятно, можно было увидеть что-либо подобное. Написание миграций возлагается на вас самих, но SQLx может применить их и за вас. Вам же следует убедиться, что они корректны, идемпотентны и, если требуется включить прямые и обратные миграции, предоставляют (опционально) как миграции вверх (создание), так и миграции вниз (уничтожение);
- создание базы данных и запуск миграций — относятся к деструктивным операциям с сохранением состояния. База данных подвергается мутации, и если допустить ошибку или опечатку, волшебной кнопки отмены у вас не будет (если только не спроектировать ее самостоятельно). Но беспокоиться здесь особо не стоит — просто нужно иметь это в виду, поскольку вряд ли когда-либо вам захочется, чтобы миграция ненароком была применена к той базе данных, к которой вы ее применять не намерены (например, при тестировании миграций на рабочей базе данных еще до того, как код будет готов к выпуску в производственную среду). В следующем разделе мы рассмотрим модель данных и порядок взаимодействия с базой данных.

9.6. Моделирование данных

Не станем усложнять наш сервис и смоделируем лишь один вид данных — элемент todo. Нашим todo-элементам будут нужны только два поля: тело (т. е. сам элемент todo), представляющее собой простую текстовую строку, и логическое поле для обозначения того, выполнена намеченная работа или нет. Можно было бы удалять todo-элемент после выполнения намеченной в нем работы, но в общем-то имеет смысл сохранять уже выполненные todo-работы — на случай, если потребуется просмотреть историю старых (выполненных) todo-работ. Поэтому мы включим для них временную метку, определяющую дату создания и время последнего обновления todo-элемента. Могла бы также понадобиться и третья временная метка, содержащая время выполнения работы, указанной в элементе, но мы не станем сильно усложнять пример.

9.6.1. SQL-схема

Составим для нашей таблицы с todo-элементами схему SQL (листинг 9.5).

Листинг 9.5. Схема SQL из файла migrations/20230701202642_todos.sql API-сервиса

```
CREATE TABLE IF NOT EXISTS todos (
    id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    body TEXT NOT NULL,
    completed BOOLEAN NOT NULL DEFAULT FALSE,
    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

Вдаваться в подробности этой схемы мы не станем, поскольку сам код SQL вполне понятен, но всё же нельзя не отметить ряд деталей:

- предоставление первичного ключа и автоматического увеличения идентификатора (ID) при создании новой записи возлагается нами на SQLite, поэтому возможность повторного использования какого-либо идентификатора можно исключить. Мы, конечно, могли бы также воспользоваться чем-то вроде UUID, но это добавило бы еще один уровень сложности для проверки факта уникальности такого любого созданного нами универсального уникального идентификатора. Первичные ключи UUID поддерживаются в последних версиях PostgreSQL и в некоторых версиях баз данных, совместимых с MySQL;
- мы не допускаем, чтобы какое-либо поле было пустым или неопределенным;
- мы предоставляем значение по умолчанию для каждого столбца, кроме текстового тела todo. Это означает, что мы можем создать новый todo только с одним фрагментом данных — текстовым телом todo;
- столбец `updated_at` будет обновляться не с помощью триггера SQL (или какого-либо другого метода), а создаваемым нами кодом Rust. Можно, конечно, пред-

почесть здесь использование триггера, и я оставлю задачу создания такого варианта в качестве упражнения для читателя. Главное преимущество использования триггера заключается в возможности выполнения простых SQL-запросов, при которых столбец `updated_at` будет обновляться соответствующим образом.

Наша инструкция `CREATE TABLE ...` добавляется в качестве миграции, поэтому таблица будет создана при первой инициализации базы данных и выполнении миграций. Для создания миграции мы задействуем CLI-команду `SQLx`:

```
$ sqlx migrate add todos
# ...
```

Эта команда создает файл с именем `migrations/20230701202642_todos.sql`, который заполняется нами кодом SQL из листинга 9.5.

9.6.2. Взаимодействие с нашими данными

Теперь можно приступить к написанию кода Rust для моделирования наших todo-элементов на языке Rust и организации взаимодействия с базой данных. При этом мы реализуем поддержку пяти операций: создание, чтение, обновление, удаление и вывод списка. Это стандартные, присущие таблицам CRUD-операции, которые обычно поставляются в готовом виде и с которыми придется неоднократно сталкиваться, если проводить много времени, работая с веб-фреймворками.

Структура нашего `Todo` представлена в листинге 9.6.

Листинг 9.6. Структура Todo из файла `src/todo.rs` API-сервиса

```
#[derive(Serialize, Clone, sqlx::FromRow)]                                     (1)
pub struct Todo {
    id: i64,
    body: String,
    completed: bool,
    created_at: NaiveDateTime,                                                 (2)
    updated_at: NaiveDateTime,
}
```

Здесь:

- в поз. (1) — мы выводим типаж `Serialize` из крейтов `serde` и `sqlx::FromRow`, позволяющих получать `Todo` из SQLx-запроса;
- в поз. (2) — использование типа `chrono::NaiveDateTime` для отображения временных меток SQL на Rust-объекты.

О самой структуре `Todo` говорить особо нечего, поэтому давайте перейдем к блокам `impl`, вызывающим гораздо больший интерес. Рассмотрим сначала код вывода списка и чтения записей (листинг 9.7).

Листинг 9.7. impl-блок чтения из файла src/todo.rs, относящийся к структуре Todo API-сервиса

```
impl Todo {
    pub async fn list(dbpool: SqlitePool) -> Result<Vec<Todo>, Error> {
        query_as("select * from todos")                                (1)
            .fetch_all(&dbpool)
            .await
            .map_err(Into::into)
    }

    pub async fn read(dbpool: SqlitePool, id: i64) -> Result<Todo, Error> {
        query_as("select * from todos where id = ?")                  (2)
            .bind(id)
            .fetch_one(&dbpool)
            .await
            .map_err(Into::into)
    }
}
```

Здесь:

- в поз. (1) — выбор всех todo-записей из таблицы todos;
- в поз. (2) — выбор из таблицы todos одной todo-записи с соответствующим значением в поле идентификатора.

В этом программном коде задействуются два метода: `list()` и `read()`. А в каждом методе применяется ожидаемое действие, совершающееся путем выполнения запроса к базе данных. Единственное реальное различие между `list()` и `read()` заключается в количестве возвращаемых записей и в том, что при чтении одной записи ее следует выбирать по идентификатору. Теперь посмотрим на листинг 9.8, содержащий код `impl`-блока записи.

Листинг 9.8. impl-блок записи из файла src/todo.rs, относящийся к структуре Todo API-сервиса

```
impl Todo {
    pub async fn create(
        dbpool: SqlitePool,
        new_todo: CreateTodo,                                (1)
    ) -> Result<Todo, Error> {
        query_as("insert into todos (body) values (?) returning *") (2)
            .bind(new_todo.body())
            .fetch_one(&dbpool)                                (3)
            .await
            .map_err(Into::into)
    }

    pub async fn update(
        dbpool: SqlitePool,
        id: i64,
        updated_todo: UpdateTodo,                          (4)
    ) -> Result<Todo, Error> {
        query_as("update todos set body = ? where id = ?")
            .bind(updated_todo.body())
            .bind(id)
            .execute(&dbpool)
            .await
            .map_err(Into::into)
    }
}
```

```
) -> Result<Todo, Error> {
    query_as(
        "update todos set body = ?, completed = ?, \
         updated_at = datetime('now') where id = ? returning *", (5)
    )
    .bind(updated_todo.body())
    .bind(updated_todo.completed())
    .bind(id)
    .fetch_one(&dbpool) (7)
    .await
    .map_err(Into::into)
}

pub async fn delete(dbpool: SqlitePool, id: i64) -> Result<(), Error> {
    query("delete from todos where id = ?") (8)
    .bind(id)
    .execute(&dbpool)
    .await?;
    Ok(())
} (10)
}
```

Здесь:

- в поз. (1) — добавление нового, до этого еще не определенного типа `CreateTodo`. В нем содержится тело todo-записи, необходимое для ее создания;
- в поз. (2) — использование возвращающего SQL-выражения `*` для извлечения записи сразу после ее вставки;
- в поз. (3) — выполнение запроса с помощью `fetch_one()`, поскольку ожидается, что будет возвращена одна строка;
- в поз. (4) — добавление еще одного нового типа `UpdateTodo`, содержащего два поля, разрешенных нами к обновлению;
- в поз. (5) — повторное использование возвращающего SQL-выражения `*` для немедленного получения обновленной записи. Обратите внимание на установку для поля `updated_at` текущих даты и времени;
- в поз. (6) — каждое значение привязывается в порядке, объявленном в SQL-инструкции, при этом для привязки значений используется токен `?`. В зависимости от конкретной реализации SQL синтаксис может различаться;
- в поз. (7) — при выполнении этого запроса ожидается извлечение одной строки;
- в поз. (8) — удаление является деструктивной операцией (в случае успеха возвращать будет нечего);
- в поз. (9) — тут для выполнения запроса используется метод `execute()`, применяемый для запросов, не возвращающих записи;
- в поз. (10) — возвращение единичного типа в случае успеха (т. е. в случае отсутствия ошибок при выполнении предыдущего кода).

Для каждого действия используется похожий код, поэтому давайте рассмотрим некоторые общие черты его поведения:

- предполагается, что любые ошибки при выполнении запроса приведут к возврату результата `sqlx::Error`, отображаемого на наш собственный тип ошибки с помощью типажа `From`, применяемого путем вызова `.map_err(Into::into)`. Типажи `From` и `Into` являются взаимно обратными, поэтому метод типажа `Into::into` для ошибки можно вызывать только при использовании `map_err()`;
- каждый запрос, за исключением действия удаления, возвращает одну или несколько записей, и поскольку мы получили для `Todo` (как было ранее упомянуто) `sqlx::FromRow`, у нас есть возможность позволить `SQLx` отобразить тип для нас;
- для выполнения каждой операции в пул базы данных нужно передать дескриптор (подключение можно было бы передать и напрямую);
- при использовании для привязки значений к SQL-инструкции метода `bind()` нужно обратить внимание на порядок следования значений, поскольку они призываются в том порядке, в котором были указаны. Некоторые движки SQL позволяют для привязки значений использовать идентификаторы, но `SQLite` этого не делает.

Рассмотрим подробнее структуры `CreateTodo` (листинг 9.9) и `UpdateTodo` (листинг 9.10), упомянутые в листинге 9.8.

Листинг 9.9. Структура `CreateTodo` из файла `src/todo.rs` API-сервиса

```
#[derive(Deserialize)]
pub struct CreateTodo {
    body: String,
}

impl CreateTodo {
    pub fn body(&self) -> &str {
        self.body.as_ref()
    }
}
```

Следует заметить, что единственным предоставляемым методом здесь является аксессор для поля `body`. Дело в том, что для создания структуры, выведенной в верхней части, мы полагаемся на `Deserialize`. То есть создавать структуру `CreateTodo` не требуется — ее при получении в вызове API нужно просто десериализовать.

Листинг 9.10. Структура `UpdateTodo` из файла `src/todo.rs` API-сервиса

```
#[derive(Deserialize)]
pub struct UpdateTodo {
    body: String,
```

```
completed: bool,  
}  
  
impl UpdateTodo {  
    pub fn body(&self) -> &str {  
        self.body.as_ref()  
    }  
    pub fn completed(&self) -> bool {  
        self.completed  
    }  
}
```

Структура `UpdateTodo` является почти такой же, как и `CreateTodo`, за исключением наличия двух полей: `body` и `complete`. И снова для создания для нас объекта мы полагаемся на библиотеку `serde`.

На этом рассмотрение модели данных завершено. А в следующем разделе мы перейдем к определению API-маршрутов.

9.7. Объявление API-маршрутов

Итак, наш API мы уже разработали, поэтому нам остается лишь объявить маршруты, воспользовавшись принадлежащим ахум маршрутизатором `Router`. Если вы уже пользовались какими-либо другими веб-фреймворками, то этот код покажется вам знакомым, поскольку он состоит из тех же компонентов: пути запроса (с опциональными параметрами), метода запроса, обработчика запроса, состояния, требующегося для обработчиков, и любых дополнительных уровней для нашего сервиса.

Так что продолжим работу и рассмотрим код листинга 9.11 из файла `router.rs`, определяющий сервис и его маршрутизатор.

Листинг 9.11. Маршрутизатор из файла `src/router.rs` API-сервиса

```
pub async fn create_router(  
    dbpool: sqlx::Pool<sqlx::Sqlite>,  
) -> axum::Router {  
    use crate::api::  
        ping, todo_create, todo_delete, todo_list, todo_read, todo_update,  
    };  
    use axum::{routing::get, Router};  
    use tower_http::cors::(Any, CorsLayer);  
    use tower_http::trace::TraceLayer;  
  
    Router::new()  
        .route("/alive", get(|| async { "ok" }))  
        .route("/ready", get(ping))  
}
```

```

    .nest(
        "/v1",
        Router::new()
            .route("/todos", get(todo_list).post(todo_create))          (4)      (6)
            .route(
                "/todos/:id",
                get(todo_read).put(todo_update).delete(todo_delete),    (6)
            ),
        )
    .with_state(dbpool)                                         (7)
    .layer(CorsLayer::new().allow_methods(Any).allow_origin(Any)) (8)
    .layer(TraceLayer::new_for_http())                           (9)
}

```

Здесь:

- в поз. (1) — передача пула базы данных маршрутизатору, который становится его владельцем;
- в поз. (2) — наша проверка жизнеспособности и готовности, просто возвращающая статус 200 с наполнением `ok`;
- в поз. (3) — выполнение нашей проверки готовности GET-запроса с обработчиком `ping()`;
- в поз. (4) — API-маршруты вложены в путь `/v1`;
- в поз. (5) — тут для пути `/v1/todos` разрешаются два метода: либо `GET`, либо `POST`, вызывающие соответственно обработчики `todo_list()` и `todo_create()`. Мы можем изменять методы вместе, воспользовавшись удобным и понятным интерфейсом;
- в поз. (6) — параметр пути `:id` отображается на `todo`-идентификатор. Методы `GET`, `PUT` или `DELETE` для `/v1/todos/:id` отображаются соответственно на `todo_read()`, `todo_update()` и `todo_delete`;
- в поз. (7) — пул подключений к базе данных передается маршрутизатору, чтобы он попал в обработчики в виде состояния;
- в поз. (8) — добавление CORS-уровня с целью демонстрации способа применения CORS-заголовков;
- в поз. (9) — чтобы получить трассировку запросов, нужно добавить уровень HTTP-трассировки из `tower_http`.

`axum::Router` является основной абстракцией веб-фреймворка `axum`, позволяющей объявлять маршруты с их обработчиками, а также подмешивать уровни из других сервисов, — например, из `tower_http`. Пусть показанный здесь пример и не самый сложный, но на основе всего, что в нем продемонстрировано, можно пойти гораздо дальше, поскольку он охватывает существенную часть вариантов использования. Для решения же сугубо практических задач вам лучше обратиться к документации

ахим⁹, что позволит изучить фреймворк и его функции более подробно. А теперь давайте перейдем к реализации обработчиков API-маршрутов.

9.8. Реализация API-маршрутов

Последний фрагмент пазла, который мы здесь собираем, — это обработчики API-маршрутов. И начнем мы с рассмотрения самого основного обработчика `ping()`, предназначенного для проверки готовности (листинг 9.12).

Листинг 9.12. Ping-обработчик из файла `src/api.rs` API-сервиса

```
pub async fn ping(                                     (1)
    State(dbpool): State<SqlitePool>,
) -> Result<String, Error> {
    use sqlx::Connection;

    let mut conn = dbpool.acquire().await?;           (2)
    conn.ping()                                       (3)
        .await
        .map(|_| "ok".to_string())                   (4)
        .map_err(Into::into)                         (5)
}
```

Здесь:

- в поз. (1) — экстрактор состояний предоставляет пул подключений к базе данных, полученный из состояния `axum`;
- в поз. (2) — сначала из пула базы данных нужно получить подключение;
- в поз. (3) — метод `ping()` проверит, в порядке ли подключение к базе данных. В варианте применения SQLite всё сводится к проверке активности фоновых потоков SQLite;
- в поз. (4) — в случае успеха `ping()` возвращает единичный тип, поэтому он просто отображается на строку `ok`, возвращаемую в качестве нашего ответа;
- в поз. (5) — для отображения `sqlx::Error` на наши собственные типы ошибок используется типаж `From`.

В `ping()` я ввел новое понятие из фреймворка `axum` — **экстрактор**. Коротко говоря, экстрактор — это всё, что реализует типаж `axum::extract::FromRequest` или же типаж `axum::extract::FromRequestParts`, но можно также воспользоваться одним из следующих экстракторов, предоставляемых `axum`:

- `axum::extract::State` — извлекает глобальное состояние приложения, предоставляемое маршрутизатору с помощью `.with_state()`, что и было показано в листинге 9.11 для пула базы данных;

⁹ См. <https://docs.rs/axum/>.

- `axum::extract::Path` — извлекает такие параметры пути, как параметр `id`, включенный нами в маршруты;
- `axum::extract::Json` — извлекает тело запроса в виде JSON-объекта и выполняет его десериализацию с помощью крейта `serde`.

Фреймворк `axum` предоставляет также ряд других экстракторов, а кроме этого, можно также создавать свои собственные экстракторы, реализовав их типажи.

А теперь мы перейдем к самым важным частям — todo-обработчикам API-маршрутов (листинг 9.13).

Листинг 9.13. todo-обработчики из файла `src/api.rs` API-сервиса

```
pub async fn todo_list(
    State(dbpool): State<SqlitePool>,
) -> Result<Json<Vec<Todo>>, Error> { (1)
    Todo::list(dbpool).await.map(Json::from)
}

pub async fn todo_read(
    State(dbpool): State<SqlitePool>,
    Path(id): Path<i64>, (3)
) -> Result<Json<Todo>, Error> {
    Todo::read(dbpool, id).await.map(Json::from)
}

pub async fn todo_create(
    State(dbpool): State<SqlitePool>,
    Json(new_todo): Json<CreateTodo>, (4)
) -> Result<Json<Todo>, Error> {
    Todo::create(dbpool, new_todo).await.map(Json::from)
}

pub async fn todo_update(
    State(dbpool): State<SqlitePool>,
    Path(id): Path<i64>,
    Json(updated_todo): Json<UpdateTodo>, (5)
) -> Result<Json<Todo>, Error> {
    Todo::update(dbpool, id, updated_todo).await.map(Json::from)
}

pub async fn todo_delete(
    State(dbpool): State<SqlitePool>,
    Path(id): Path<i64>,
) -> Result<(), Error> {
    Todo::delete(dbpool, id).await
}
```

Здесь:

- в поз. (1) — заметьте, что возвращается JSON-объект `Vec<Todo>` или же, возможно, ошибка;
- в поз. (2) — метод `Todo::list()` возвращает простой `Vec<Todo>`, поэтому он отображается на JSON-объект, для чего используется `Json::from`, зависящий от типажа `Serialize`, полученного для `Todo`;
- в поз. (3) — параметр пути, доступ к которому мы получаем с помощью экстрактора `Path`. Отображение идентификатора из пути маршрутизатора `/v1/todos/:id` на именованный параметр с использованием при этом безопасного для типов способа фреймворк `axum` берет на себя;
- в поз. (4) — тут нами вводится структура `CreateTodo`, получаемая из тела запроса с помощью экстрактора `Json`, задействующего реализацию `Deserialize`, полученную при использовании крейта `serde`;
- в поз. (5) — структура `UpdateTodo`, получаемая из тела запроса с применением экстрактора `Json`, задействующего реализацию `Deserialize`, полученную при использовании крейта `serde`.

Код для наших обработчиков API совсем небольшой. Поскольку основная часть всей тяжелой работы уже проделана, на этом шаге речь идет только об определении входов и выходов для каждого из наших обработчиков. Фреймворк `axum` будет сопоставлять запросы только с обработчиками, имеющими допустимые экстракторы с учетом заданного пути и метода запроса, и делать это безопасным для типов способом, поэтому слишком сильно задумываться о работоспособности обработчиков после успешной компиляции кода нам не придется. В этом, собственно, и заключается красота Rust и обеспечиваемая им безопасность типов.

ПРИМЕЧАНИЕ

Возвращаясь с небес на землю, следует отметить, что наш API разработан с жестко заданными параметрами. Например, в нем ни в одной из конечных точек не допускаются опциональные поля — предоставлять можно только обязательные, иначе сервис вернет ошибку. В большинстве случаев это вполне подходящий вариант, но в качестве упражнения мои читатели могут, к примеру, попробовать в запросах `PATCH` или обновления сделать поле `completed` опциональным. Если необходимо изменить лишь одно конкретное поле, то представляется вполне разумным, что API будет корректно обрабатывать только указанные поля, ведь так же?

Теперь у нас имеется полностью функционирующий API-сервис с завершенными основными конечными точками CRUD. Осталось рассмотреть еще одну тему — обработку ошибок, после чего можно будет запустить несколько тестов и посмотреть, как работает наше свежеиспеченное изделие.

Но прежде чем перейти к обработке ошибок, давайте коротко рассмотрим порядок обработки ответов, заведенный в `axum`. В исходном готовом варианте `axum` будет конвертировать базовые типы (единичный тип, `String`, `Json` и `axum::http::StatusCode`) в HTTP-ответы. Фреймворк это делает, предоставляемый для наиболее распространен-

ных типов ответов реализацию типажа `axum::response::IntoResponse`. Если потребуется конвертировать в ответ собственный тип, его нужно будет преобразовать во что-либо, реализующее `IntoResponse`, или же самостоятельно реализовать `IntoResponse`, что и будет продемонстрировано в следующем разделе.

9.9. Обработка ошибок

Обработка ошибок организована у нас очень просто — в файле `error.rs` определяется одно перечисление с именем `Error` (листинг 9.14).

Листинг 9.14. Перечисление Error из файла src/error.rs API-сервиса

```
#[derive(Debug)]
pub enum Error {
    Sqlx.StatusCode, String), (1)
    NotFound, (2)
}
```

Здесь:

- в поз. (1) — ошибки из `sqlx::Error` будут конвертироваться в код состояния HTTP и сообщение;
- в поз. (2) — `Error::NotFound` будет использоваться нами для удобного отображения ответов на HTTP-состояние 404.

ПРИМЕЧАНИЕ

Тут код 404 (не найдено) рассматривается в качестве ошибки, но 404 также является нормальным ответом HTTP, который не всегда указывает на ошибку. Однако — для удобства — у нас всё, не являющееся кодом состояния 200, считается ошибкой.

Теперь для метода `sqlx::Error`, конвертирующего ошибки SQLx в наш тип ошибки, нужно определить типаж `From` (листинг 9.15).

Листинг 9.15. Реализация From для sqlx::Error из файла src/error.rs API-сервиса

```
impl From<sqlx::Error> for Error {
    fn from(err: sqlx::Error) -> Error {
        match err {
            sqlx::Error::RowNotFound => Error::NotFound, (1)
            _ => Error::Sqlx(
                StatusCode::INTERNAL_SERVER_ERROR, (2)
                err.to_string(), (3)
            ),
        }
    }
}
```

Здесь:

- в поз. (1) — для запросов, не завершившихся нахождением соответствующих строк, возвращается HTTP 404;
- в поз. (2) — для всех остальных SQLx-ошибок возвращается HTTP 500;
- в поз. (3) — строка, возвращаемая SQLx-ошибкой, включается в тело ответа наших 500-х состояний.

Создаваемая нами реализация `From<sqlx::Error> for Error` довольно проста — в качестве особого обрабатывается только один случай: `NotFound`. Для этого он отображается на HTTP 404, что полезнее возврата обобщенной ошибки 500.

Далее нужно сделать так, чтобы `axum` мог использовать наш тип ошибки в качестве ответа, поэтому для `Error` реализуется `IntoResponse` (листинг 9.16).

Листинг 9.16. Реализация `IntoResponse` для `sqlx::Error` из файла `src/error.rs` API-сервиса

```
impl IntoResponse for Error {
    fn into_response(self) -> Response {
        match self {
            Error::Sqlx(code, body) => (code, body).into_response(), (1)
            Error::NotFound => StatusCode::NOT_FOUND.into_response(), (2)
        }
    }
}
```

Здесь:

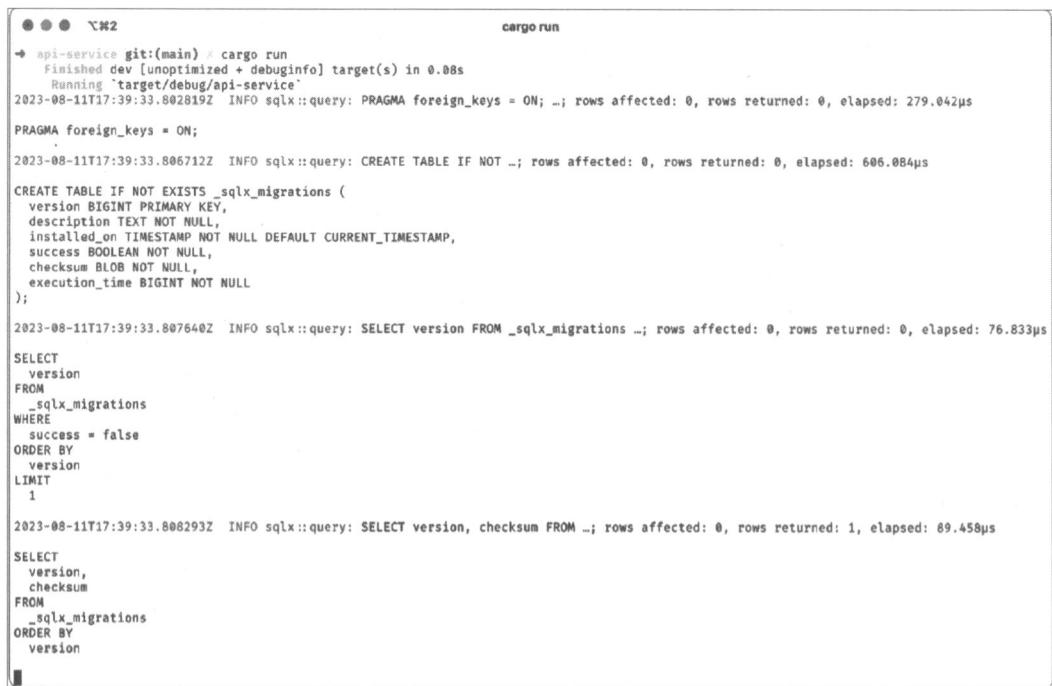
- в поз. (1) — извлечение кода состояния и тела ответа с последующим вызовом `into_response()` для кортежа (`StatusCode, String`), поскольку реализация `IntoResponse` предоставляется нам фреймворком `axum`;
- в поз. (2) — вызов `into_response()` для `StatusCode::NOT_FOUND`, в результате чего получается пустой ответ HTTP 404.

Можно заметить, что в этом коде мы даже не пытаемся выстроить `Response`, как того требует `IntoResponse`. Благодаря реализациям, предоставляемым `axum`, выстраивание ответа с использованием существующей реализации `IntoResponse` делегируется `axum` — весьма изящный прием, требующий минимальных усилий с нашей стороны. К нему не захочется прибегать лишь в том случае, когда исходная реализация включает чересчур затратную конвертацию и у вас достаточно информации, чтобы найти более подходящий способ оптимизации.

9.10. Запуск сервиса

Давайте запустим сервис и убедимся, что он ведет себя ожидаемым образом. Выполните команду `cargo run`, и на экран будет выведена информация, похожая на ту, что показана на рис. 9.2. Содержащиеся здесь регистрационные записи демонстри-

рутут выполняемые при запуске запросы из SQLx, включая запуск миграций. Нам не нужен автоматический запуск миграций, но он удобен для тестирования. В рабочем сервисе автоматического запуска миграций, скорее всего, не будет.



```

● ● ● ✘cargo run
→ api-service git:(main) ✘ cargo run
   Finished dev [unoptimized + debuginfo] target(s) in 0.08s
     Running `target/debug/api-service`
2023-08-11T17:39:33.802892Z INFO sqlx::query: PRAGMA foreign_keys = ON; ... rows affected: 0, rows returned: 0, elapsed: 279.042µs
PRAGMA foreign_keys = ON;

2023-08-11T17:39:33.806712Z INFO sqlx::query: CREATE TABLE IF NOT ...; rows affected: 0, rows returned: 0, elapsed: 606.084µs

CREATE TABLE IF NOT EXISTS _sqlx_migrations (
    version BIGINT PRIMARY KEY,
    description TEXT NOT NULL,
    installed_on TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    success BOOLEAN NOT NULL,
    checksum BLOB NOT NULL,
    execution_time BIGINT NOT NULL
);

2023-08-11T17:39:33.807640Z INFO sqlx::query: SELECT version FROM _sqlx_migrations ...; rows affected: 0, rows returned: 0, elapsed: 76.833µs

SELECT
    version
FROM
    _sqlx_migrations
WHERE
    success = false
ORDER BY
    version
LIMIT
    1

2023-08-11T17:39:33.808293Z INFO sqlx::query: SELECT version, checksum FROM ...; rows affected: 0, rows returned: 1, elapsed: 89.458µs

SELECT
    version,
    checksum
FROM
    _sqlx_migrations
ORDER BY
    version

```

Рис. 9.2. Запуск API-сервиса

API-сервис нужно протестировать, но сначала следует убедиться в надлежащей работе конечных точек проверки работоспособности. Для этих тестов мы воспользуемся инструментом HTTPie¹⁰, но можно было бы также легко применить curl или любой другой CLI-клиент HTTP.

Давайте запустим комманду:

`http 127.0.0.1:3000/alive`

а затем комманду:

`http 127.0.0.1:3000/ready`

которые сгенерируют GET-запрос HTTP к каждой конечной точке, с результатом, приведенным на рис. 9.3. Здесь на экране слева показан вывод журнала нашего сервиса, а справа — вывод HTTPie. Пока нас здесь всё устраивает: код состояния HTTP для каждого запроса равен 200, в теле запроса просто `ok`, а CORS-заголовки, как и ожидалось, присутствуют.

¹⁰ См. <https://httpie.io/>.

```

● ● ● ℰ%2 brenden@fruit-computer:~ 
X cargo (api-service)
updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);

2023-08-11T17:58:19.704713Z INFO sqlx::query: INSERT INTO _sqlx_migr
ations ( ...; rows affected: 1, rows returned: 0, elapsed: 169.666µs

INSERT INTO
    _sqlx_migrations (
        version,
        description,
        success,
        checksum,
        execution_time
    )
VALUES
    (?1, ?2, TRUE, ?3, -1)

2023-08-11T17:58:19.705968Z INFO sqlx::query: UPDATE _sqlx_migration
s SET execution_time ...; rows affected: 1, rows returned: 0, elapsed:
324.833µs

UPDATE
    _sqlx_migrations
SET
    execution_time = ?1
WHERE
    version = ?2

2023-08-11T17:58:37.773384Z DEBUG request{method=GET uri=/alive versi
on=HTTP/1.1}: tower_http::trace::on_request: started processing requ
est
2023-08-11T17:58:37.773500Z DEBUG request{method=GET uri=/alive versi
on=HTTP/1.1}: tower_http::trace::on_response: finished processing req
uest latency=0 ms status=200
2023-08-11T17:58:45.350770Z DEBUG request{method=GET uri=/ready versi
on=HTTP/1.1}: tower_http::trace::on_request: started processing requ
est
2023-08-11T17:58:45.351088Z DEBUG request{method=GET uri=/ready versi
on=HTTP/1.1}: tower_http::trace::on_response: finished processing req
uest latency=0 ms status=200
⇒ ~ (-zsh)
⇒ ~ http 127.0.0.1:3000/alive
HTTP/1.1 200 OK
access-control-allow-origin: *
content-length: 2
content-type: text/plain; charset=utf-8
date: Fri, 11 Aug 2023 17:58:37 GMT
vary: origin
vary: access-control-request-method
vary: access-control-request-headers
ok

⇒ ~ http 127.0.0.1:3000/ready
HTTP/1.1 200 OK
access-control-allow-origin: *
content-length: 2
content-type: text/plain; charset=utf-8
date: Fri, 11 Aug 2023 17:58:45 GMT
vary: origin
vary: access-control-request-method
vary: access-control-request-headers
ok
⇒ ~

```

Рис. 9.3. Проверка работоспособности сервиса

Теперь пришло время создать todo. Для этого, как показано на рис. 9.4, с помощью команды:

```
http post 127.0.0.1:3000/v1/todos body='wash the dishes'
```

мы выполним POST-запрос HTTP.

Теперь, как показано на рис. 9.5, с помощью команд:

- http 127.0.0.1:3000/v1/todos/1 — для чтения;
- http 127.0.0.1:3000/v1/todos — для списка

протестируем GET-методы HTTP для конечных точек чтения и вывода списка.

Заметьте, что первый запрос (для конкретного элемента данных) возвращает только объект todo, а второй запрос (для перечисления всех элементов данных) — список объектов. Пока всё приемлемо. Далее, как показано на рис. 9.6, с помощью команды:

```
http put 127.0.0.1:3000/v1/todos/1 body='wash the dishes' done:=true
```

мы протестируем PUT-метод для обновления нашей todo-записи, пометив ее выполненной.

```
● ● ● ℰ 322 brenden@fruit-computer:~ 
X cargo (api-service)
2023-08-11T18:03:15.309076Z INFO sqlx::query: INSERT INTO _sqlx_migrations ( ; rows affected: 1, rows returned: 0, elapsed: 146.042µs
INSERT INTO
    _sqlx_migrations (
        version,
        description,
        success,
        checksum,
        execution_time
    )
VALUES
    (?1, ?2, TRUE, ?3, -1)

2023-08-11T18:03:15.310202Z INFO sqlx::query: UPDATE _sqlx_migration
s SET execution_time = ?; rows affected: 1, rows returned: 0, elapsed: 342.750µs
UPDATE
    _sqlx_migrations
SET
    execution_time = ?1
WHERE
    version = ?2

2023-08-11T18:03:17.201422Z DEBUG request{method=POST uri=/v1/todos v
ersion=HTTP/1.1}: tower_http::trace::on_request: started processing r
equest
2023-08-11T18:03:17.202317Z DEBUG request{method=POST uri=/v1/todos v
ersion=HTTP/1.1}: tower_http::trace::on_response: finished processing
request latency=0 ms status=200
2023-08-11T18:03:17.202790Z INFO sqlx::query: insert into todos (body) ; rows affected: 1, rows returned: 1, elapsed: 707.917µs
insert into
    todos (body)
values
    (?) returning *
X ~ (-zsh)
⇒ ~ http post 127.0.0.1:3000/v1/todos body='wash the dishes'
HTTP/1.1 200 OK
access-control-allow-origin: *
content-length: 121
content-type: application/json
date: Fri, 11 Aug 2023 18:03:17 GMT
vary: origin
vary: access-control-request-method
vary: access-control-request-headers
{
    "body": "wash the dishes",
    "completed": false,
    "created_at": "2023-08-11T18:03:17",
    "id": 1,
    "updated_at": "2023-08-11T18:03:17"
}
⇒ ~
```

Рис. 9.4. Создание todo-записи с помощью POST-запроса HTTP

```
● ● ● ℰ 322 brenden@fruit-computer:~ 
X cargo (api-service)
version = ?2

2023-08-11T18:03:17.201422Z DEBUG request{method=POST uri=/v1/todos v
ersion=HTTP/1.1}: tower_http::trace::on_request: started processing r
equest
2023-08-11T18:03:17.202317Z DEBUG request{method=POST uri=/v1/todos v
ersion=HTTP/1.1}: tower_http::trace::on_response: finished processing
request latency=0 ms status=200
2023-08-11T18:03:17.202790Z INFO sqlx::query: insert into todos (body) ; rows affected: 1, rows returned: 1, elapsed: 707.917µs
insert into
    todos (body)
values
    (?) returning *
2023-08-11T18:03:44.604387Z DEBUG request{method=GET uri=/v1/todos/1 v
ersion=HTTP/1.1}: tower_http::trace::on_request: started processing
request
2023-08-11T18:03:44.604961Z DEBUG request{method=GET uri=/v1/todos/1 v
ersion=HTTP/1.1}: tower_http::trace::on_response: finished processin
g request latency=0 ms status=200
2023-08-11T18:03:44.605273Z INFO sqlx::query: select * from todos -; rows affected: 1, rows returned: 1, elapsed: 135.458µs
select
    *
from
    todos
where
    id = ?
2023-08-11T18:03:51.777124Z DEBUG request{method=GET uri=/v1/todos ve
rsion=HTTP/1.1}: tower_http::trace::on_request: started processing re
quest
2023-08-11T18:03:51.777644Z INFO sqlx::query: select * from todos; r
ows affected: 1, rows returned: 1, elapsed: 111.833µs
2023-08-11T18:03:51.777617Z DEBUG request{method=GET uri=/v1/todos ve
rsion=HTTP/1.1}: tower_http::trace::on_response: finished processing
request latency=0 ms status=200
X ~ (-zsh)
⇒ ~ http 127.0.0.1:3000/v1/todos/1
HTTP/1.1 200 OK
access-control-allow-origin: *
content-length: 121
content-type: application/json
date: Fri, 11 Aug 2023 18:03:44 GMT
vary: origin
vary: access-control-request-method
vary: access-control-request-headers
{
    "body": "wash the dishes",
    "completed": false,
    "created_at": "2023-08-11T18:03:17",
    "id": 1,
    "updated_at": "2023-08-11T18:03:17"
}
⇒ ~ http 127.0.0.1:3000/v1/todos
HTTP/1.1 200 OK
access-control-allow-origin: *
content-length: 123
content-type: application/json
date: Fri, 11 Aug 2023 18:03:51 GMT
vary: origin
vary: access-control-request-method
vary: access-control-request-headers
[
    {
        "body": "wash the dishes",
        "completed": false,
        "created_at": "2023-08-11T18:03:17",
        "id": 1,
        "updated_at": "2023-08-11T18:03:17"
    }
]
⇒ ~
```

Рис. 9.5. Чтение todo-записей с помощью GET-запросов HTTP

```

● ● ● ℰ 22
brenden@fruit-computer:~          X ~ (-zsh)
X cargo (api-service)
request
2023-08-11T18:03:44.604961Z DEBUG request{method=GET uri=/v1/todos/1
version=HTTP/1.1}: tower_http::trace::on_response: finished processing
g request latency=0 ms status=200
2023-08-11T18:03:44.605273Z INFO sqlx::query: select * from todos ..
rows affected: 1, rows returned: 1, elapsed: 135.458µs

select
*
from
todos
where
id = ?

2023-08-11T18:03:51.777124Z DEBUG request{method=GET uri=/v1/todos ve
rsion=HTTP/1.1}: tower_http::trace::on_request: started processing re
quest
2023-08-11T18:03:51.777449Z INFO sqlx::query: select * from todos; r
ows affected: 1, rows returned: 1, elapsed: 111.833µs
2023-08-11T18:03:51.777617Z DEBUG request{method=PUT uri=/v1/todos ve
rsion=HTTP/1.1}: tower_http::trace::on_response: finished processing
request latency=0 ms status=200
2023-08-11T18:04:15.756046Z DEBUG request{method=PUT uri=/v1/todos/1
version=HTTP/1.1}: tower_http::trace::on_request: started processing
request
2023-08-11T18:04:15.756714Z DEBUG request{method=PUT uri=/v1/todos/1
version=HTTP/1.1}: tower_http::trace::on_response: finished processin
g request latency=0 ms status=200
2023-08-11T18:04:15.757837Z INFO sqlx::query: update todos set body
..; rows affected: 1, rows returned: 1, elapsed: 687.750µs

update
todos
set
body = ?,
completed = ?,
updated_at = datetime('now')
where
id = ? returning *

```

Рис. 9.6. Обновление todo-записи с помощью PUT-запроса HTTP

```

● ● ● ℰ 22
brenden@fruit-computer:~          X ~ (-zsh)
X cargo (api-service)
request
2023-08-11T18:03:51.777124Z DEBUG request{method=GET uri=/v1/todos ve
rsion=HTTP/1.1}: tower_http::trace::on_request: started processing re
quest
2023-08-11T18:03:51.777449Z INFO sqlx::query: select * from todos; r
ows affected: 1, rows returned: 1, elapsed: 111.833µs
2023-08-11T18:03:51.777617Z DEBUG request{method=DELETE uri=/v1/todos/1
version=HTTP/1.1}: tower_http::trace::on_response: finished processing
request latency=0 ms status=200
2023-08-11T18:04:15.756046Z DEBUG request{method=PUT uri=/v1/todos/1
version=HTTP/1.1}: tower_http::trace::on_request: started processing
request
2023-08-11T18:04:15.756714Z DEBUG request{method=PUT uri=/v1/todos/1
version=HTTP/1.1}: tower_http::trace::on_response: finished processin
g request latency=0 ms status=200
2023-08-11T18:04:15.757837Z INFO sqlx::query: update todos set body
..; rows affected: 1, rows returned: 1, elapsed: 687.750µs

update
todos
set
body = ?,
completed = ?,
updated_at = datetime('now')
where
id = ? returning *

2023-08-11T18:05:10.683983Z DEBUG request{method=DELETE uri=/v1/todos
/1 version=HTTP/1.1}: tower_http::trace::on_request: started processi
ng request
2023-08-11T18:05:10.685106Z INFO sqlx::query: delete from todos wher
e ..; rows affected: 1, rows returned: 0, elapsed: 625.542µs

delete from
todos
where
id = ?

2023-08-11T18:05:10.685198Z DEBUG request{method=DELETE uri=/v1/todos
/1 version=HTTP/1.1}: tower_http::trace::on_response: finished proces
sing request latency=1 ms status=200

```

Рис. 9.7. Удаление todo-записи с помощью DELETE-запроса HTTP

Досадно, что приходится указывать не только поле `complete`, но и поле `body`. Было бы неплохо при обновлении записи выборочно обрабатывать только обязательные поля, но решение этой задачи я оставлю в качестве упражнения для читателей. А в завершение, как показано на рис. 9.7, мы проверим возможность удаления нашей `todo`-записи с помощью команды:

```
http delete 127.0.0.1:3000/v1/todos/1
```

Ура! Похоже, всё работает. А читателям в качестве упражнения я предлагаю запустить еще несколько тестов и поэкспериментировать с некоторыми из следующих вариантов:

- добавление нескольких записей;
- вывод списка из нескольких записей;
- разрешение опциональных полей (как уже ранее предлагалось);
- попытка использования другого типа первичного ключа (например, UUID);
- изменение ответа на POST-запрос для включения URL-адреса ресурса или 3xx-перенаправления — паттерна, иногда используемого в RESTful API-интерфейсах.

Резюме

- `axum` — веб-фреймворк, предоставляющий все необходимое для создания в Rust API-интерфейсов веб-сервисов с использованием асинхронной среды выполнения Tokio. Имеются, конечно, и другие веб-фреймворки для Rust, но `axum` предоставляет типобезопасный API без потребности в макросах.
- Чтобы получить всё, что нужно в веб-фреймворке, `axum` может использоваться вместе с целым рядом других крейтов, но сам крейт `axum` имеет весьма скромный размер и предоставляет всего лишь несколько ключевых абстракций: маршрутизатор, экстракторы, ответы, обработку ошибок и интеграцию с крейтами `tower` и `tower-http`.
- `axum` через крейт `hyper` поддерживает HTTP/1 и HTTP/2, способен с помощью `rustls` обрабатывать TLS-завершение и построен на невероятно быстрой асинхронной среде выполнения Tokio со стабильной поддержкой Rust для синтаксиса `async/await`.
- Основная часть работы по созданию API-сервиса включает проектирование модели данных, выбор управления состоянием, добавление трассировки и ведения журнала, выбор способа визуализации данных на стороне клиента (HTML, JSON и т. д.) и выбор архитектуры, соответствующей вашим потребностям.
- Стандартная архитектура веб-уровня будет соответствовать потребностям значительной части приложений — она относительно надежна, хорошо масштабируется, вполне понятна и поддерживается такими известными стандартами, как HTTP.

10

Создание CLI-инструмента HTTP REST API

В этой главе:

- выбор используемых инструментов и библиотек;
- проектирование CLI-инструмента;
- объявление команд;
- реализация команд;
- реализация запросов;
- надлежащая обработка ошибок;
- тестирование нашего CLI-инструмента.

Продолжая работу, проделанную в предыдущей главе, мы напишем здесь инструмент интерфейса командной строки (Command Line Interface, CLI) для созданного нами API-сервиса. Используя этот инструмент, мы покажем еще один способ асинхронного взаимодействия в Rust, связанный с выполнением HTTP-запросов к отдельно взятому сервису (который нами уже реализован ранее). Созданный CLI-инструмент предоставит нам удобный вариант взаимодействия с серверной частью нашего приложения todo (с API-сервисом). Таким образом, опираясь на возможности асинхронного Rust, мы рассмотрим здесь основы написания асинхронных запросов на языке Rust со стороны клиента в клиент-серверных отношениях.

Создание CLI-инструментов является одним из способов использования программных средств для решения возникающих проблем, а применение созданных инструментов позволяет избегать повторений, ошибок и излишней траты времени на те задачи, для выполнения которых лучше всего подходят компьютеры. В большинстве версий изложения философии UNIX (имеющей несколько разновидностей) включен постулат «делай одно дело и делай его хорошо», который мы и применим к нашему CLI-инструменту. Мы также упростим передачу выходных данных из нашего CLI-инструмента в другой инструмент (еще один постулат философии UNIX), что позволит объединять команды в цепочки.

10.1. Выбор используемых инструментов и библиотек

Мы продолжим здесь работать со средой выполнения Tokio, а для создания HTTP-запросов снова воспользуемся библиотекой `hyper`, обеспечивающей реализацию HTTP-протокола (как для серверов, так и для клиентов). Кроме того, мы задействуем также новый крейт — `clap`¹, предлагающий структурированный и безопасный в отношении типов парсинг командной строки.

Следует упомянуть и о существовании клиентской библиотеки HTTP более высокого уровня — `reqwest`², которая похожа на Python-библиотеку `Requests`, но реализована для Rust, однако мы тем не менее остановимся здесь на `hyper`, поскольку она более низкоуровневая и позволяет при ее непосредственном использовании узнать немного больше о том, как всё работает, и откажемся от обращения к `reqwest`, представляющей собой лишь обертку для библиотеки `hyper`. Впрочем, на практике, наверное, всё же лучше задействовать `reqwest` (которая имеет более удобный и дружественный к пользователю API-интерфейс).

Перечень всех используемых нами здесь крейтов (зависимостей API-сервиса) и их функций приведен в табл. 10.1.

Таблица 10.1. Зависимости API-сервиса

Название	Функции	Описание
<code>clap</code>	<code>derive</code>	Фреймворк командной строки
<code>colored_json</code>	<i>Стандартные</i>	Красивый вывод данных в формате JSON
<code>hyper</code>	<code>client, http1, tcp, stream</code>	Клиент-серверный API-интерфейс HTTP
<code>serde</code>	<i>Стандартные</i>	Библиотека сериализации/десериализации
<code>serde_json</code>	<i>Стандартные</i>	JSON-сериализация/десериализация для крейта <code>serde</code>
<code>tokio</code>	<code>macros, rt-multi-thread, io-util, io-std</code>	Асинхронная среда выполнения, используемая с <code>hyper</code>
<code>yansi</code>	<i>Стандартные</i>	Цветовой вывод ANSI

Удобнее всего установку всего, что показано в табл. 10.1, можно выполнить с помощью следующего набора команд:

```
cargo add clap --features derive
cargo add colored_json
cargo add hyper --features client,http1,tcp,stream
cargo add serde
cargo add serde_json
cargo add tokio --features macros,rt-multi-thread,io-util,io-std
cargo add yansi
```

¹ См. <https://crates.io/crates/clap>.

² См. <https://crates.io/crates/reqwest>.

После выполнения всех этих команд ваш файл Cargo.toml приобретет вид, показанный в листинге 10.1.

Листинг 10.1. Файл Cargo.toml API-клиента

```
[package]
name = "api-client"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/
➥ reference/manifest.html

[dependencies]
clap = { version = "4.3.10", features = ["derive"] }
colored_json = "3.2.0"
hyper = { version = "0.14.27", features = ["client", "http1", "tcp", "stream"] }
serde = "1.0.166"
serde_json = "1.0.100"
tokio = { version = "1.29.1", features = ["macros", "rt-multi-thread", "io-util", "io-std"] }
yansi = "0.5.1"
```

Определив зависимости, мы можем перейти к рассмотрению структуры проектируемого нами интерфейса командной строки (CLI).

10.2. Проектирование CLI

Наш интерфейс командной строки (CLI) будет очень простым — в нем мы собираемся отобразить наши пять CRUD-команд и команду вывода списка на CLI-команды, которые, как показано в табл. 10.2, будут в точности совершать ожидаемые действия.

Таблица 10.2. CLI-команды

Команда	Действие	Метод	Путь
create	Создание задачи (todo)	POST	/v1/todos
read	Считывание задачи по ID	GET	/v1/todos/:id
update	Обновление задачи по ID	PUT	/v1/todos/:id
delete	Удаление задачи по ID	DELETE	/v1/todos/:id
list	Вывод списка всех задач	GET	/v1/todos

Мы вернем ответ непосредственно из API, выводя его на стандартное устройство вывода, а ответам в формате JSON придадим для удобства чтения красивый вид. Такой подход позволит нам передать команду другому инструменту (например, jq), а также сделать вывод удобочитаемым.

Библиотека `clap` позволяет создавать командные CLI-интерфейсы с позиционными аргументами или с опциональными параметрами. Она автоматически генерирует справку (которую можно получить с помощью команды `help`), а параметры могут относиться как к команде верхнего уровня, так и к одной из подкоманд. `Clap` возьмет на себя парсинг аргументов при правильно определенных нами их типах и обработку ошибок аргументов в случае их некорректности или недействительности. После завершения выполнения `clap` парсинга будет оставаться определяемая нами структура, содержащая все значения, полученные в результате парсинга из аргументов командной строки.

Итак, давайте приступим к разбору кода и посмотрим, как с помощью `clap` определяется интерфейс.

10.3. Объявление команд

API `clap` для объявления интерфейса в дополнение к некоторым процедурным макросам задействует макрос `derive`. Как показано в листинге 10.2, где мы определяем наш CLI, мы собираемся воспользоваться командным интерфейсом, который может быть включен с помощью `clap` при использовании макроса `#[command]`.

Листинг 10.2. Определение высокогоревневого CLI для `clap` из файла `src/main.rs`

```
#[derive(Parser)]                                     (1)
struct Cli {
    /// Base URL of API service                   (2)
    url: hyper::Uri,                            (3)
    #[command(subcommand)]                      (4)
    command: Commands,                         (5)
}
```

Здесь:

- в поз. (1) — создание для нашего CLI `clap::Parser`, позволяющего выполнять парсинг аргументов командной строки с использованием структуры `Cli`;
- в поз. (2) — обратите внимание на применение тройного слеша `///`, который `clap` воспринимает как строку справки для этого аргумента (являющегося URL-адресом API-сервиса);
- в поз. (3) — для первого аргумента парсингу непосредственно подвергается структура `hyper::Uri`, поскольку в ней реализуется используемый `clap` типаж `FromStr`;
- в поз. (4) — наш второй аргумент является подкомандой, обозначаемой макроподкомандой `#[command()]`;
- в поз. (5) — сюда входит и наша подкоманда (второй аргумент), которая будет определена далее.

Для CLI верхнего уровня определены два позиционных аргумента: базовый URL нашего API-сервиса и подкоманда (одна из следующих: `create`, `read`, `update`, `delete` или `list`). Команды еще не определены, поэтому нам нужен код листинга 10.3.

Листинг 10.3. Определение подкоманд CLI из файла `src/main.rs`

```
#[derive(Subcommand, Debug)] (1)
enum Commands {
    /// List all todos
    List,
    /// Create a new todo
    Create {
        /// The todo body
        body: String,
    },
    /// Read a todo
    Read {
        /// The todo ID
        id: i64,
    },
    /// Update a todo
    Update {
        /// The todo ID
        id: i64,
        /// The todo body
        body: String,
        /// Mark todo as completed
        #[arg(short, long)] (3)
        completed: bool,
    },
    /// Delete a todo
    Delete {
        /// The todo ID
        id: i64,
    },
}
```

Здесь:

- в поз. (1) — тут для использования в качестве подкоманды извлекается `clap::Subcommand`;
- в поз. (2) — заметьте, что тут используется перечисление, поскольку за раз можно выбрать только одну команду;
- в поз. (3) — этот логический аргумент будет сделан необязательным путем использования вместо позиционного аргумента переключателя аргументов.

Обратите внимание, что мы ввели аргументы для наших команд, а самый первый вариант List никаких аргументов не требует. После реализации в следующем разделе функции main() наш CLI может быть запущен с помощью команды:

```
cargo run --help
```

которая выведет справочное сообщение, похожее на показанное далее (обратите внимание, что при использовании cargo run аргументы должны следовать за двойным тире: --):

```
Usage: api-client <URL> <COMMAND>
```

Commands:

List	List all todos
create	Create a new todo
read	Read a todo
update	Update a todo
delete	Delete a todo
help	Print this message or the help of the given subcommand(s)

Arguments:

<URL>	Base URL of API service
-------	-------------------------

Options:

-h, --help	Print help
------------	------------

Каждая подкоманда также выведет свою собственную справку — например, при выполнении команды:

```
cargo run --help create
```

или команды:

```
cargo run --create --help
```

будет выведено следующее:

```
Create a new todo
```

```
Usage: api-client <URL> create <BODY>
```

Arguments:

<BODY>	The todo body
--------	---------------

Options:

-h, --help	Print help
------------	------------

Отлично! Теперь можно перейти к реализации команд.

10.4. Реализация команд

Предоставляемый clap безопасный в отношении типов API существенно упрощает обработку каждой команды и ее аргументов. Мы можем отобразить каждый вариант в перечислении Commands и соответствующим образом обработать команду. Для обработки каждой команды имеется шаблон парсинга аргументов CLI и базового URL (листинг 10.4).

Листинг 10.4. Шаблон разбора аргументов CLI из файла src/main.rs

```
#[tokio::main]

async fn main() -> Result<(), Box

```

Здесь:

- в поз. (1) — вызов в main() функции Cli::parse() для парсинга аргументов CLI с их помещением в нашу структуру Cli;
- в поз. (2) — парсинг базового URL с разбиением его на части и добавлением их в новый построитель hyper::Uri;
- в поз. (3) — извлечение базовой схемы URL (например, http или https);
- в поз. (4) — извлечение базового URL-адреса (например, localhost или 127.0.0.1).

В этом коде базовый URL разбивается на части, хотя, что примечательно, путь к базовому URL мы решили проигнорировать. Вы можете разрешить указывать префиксную базу и добавлять каждый URL запроса к префиксу, но для нашего примера этот путь будет проигнорирован.

Теперь давайте рассмотрим код для обработки каждой команды (листинг 10.5).

Листинг 10.5. Обработка команд CLI из файла src/main.rs

```
match cli.command {
    Commands::List => {
        request(
            uri_builder.path_and_query("/v1/todos").build()?,
            Method::GET,
            None,
        )
    }
}
```

```
.await
}

Commands::Delete { id } => {
    request(
        uri_builder
            .path_and_query(format!("v1/todos/{}/", id))
            .build()?,
        Method::DELETE,
        None,
    )
    .await
}

Commands::Read { id } => {
    request(
        uri_builder
            .path_and_query(format!("v1/todos/{}/", id))
            .build()?,
        Method::GET,
        None,
    )
    .await
}

Commands::Create { body } => {
    request(
        uri_builder.path_and_query("/v1/todos").build()?,
        Method::POST,
        Some(json!({ "body": body }).to_string()),
    )
    .await
}

Commands::Update {
    id,
    body,
    completed,
} => {
    request(
        uri_builder
            .path_and_query(format!("v1/todos/{}/", id))
            .build()?,
        Method::PUT,
        Some(json!({ "body": body, "completed": completed }).to_string()),
    )
    .await
}
```

Для каждой команды здесь вызывается функция `request()` (которая еще нами не определена), куда передается URI запроса, метод HTTP и опциональное тело запроса.

са в формате JSON. Для построения URI используется `uri_builder`, определенный в коде листинга 10.4.

Поскольку Rust относится к обработке каждого варианта из перечисления со всей строгостью, можно с уверенностью утверждать, что нам удалось разобраться с каждым примером команды и ее параметрами (при условии, что все они правильно определены в перечислении `Commands`). Теперь можно продолжить работу и приступить к реализации HTTP-запросов.

10.5. Реализация запросов

Мы хорошо продумали определение команд и аргументов, поэтому теперь выполнение реальных запросов к API не составит особого труда. У нас имеются все необходимые для этого части (URI, HTTP-метод и опциональное тело запроса), поэтому всё, что остается сделать, — выполнить сам запрос. Для этого, как показано в листинге 10.6, можно воспользоваться одной-единственной функцией.

Листинг 10.6. Выполнение CLI-запроса из файла `src/main.rs`

```
async fn request(
    url: hyper::Uri,
    method: Method,
    body: Option<String>,
) -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
    let client = Client::new();

    let mut res = client
        .request(
            Request::builder()
                .uri(url)
                .method(method)
                .header("Content-Type", "application/json")           (1)
                .body(body.map(|s| Body::from(s)))
                .unwrap_or_else(|| Body::empty())?                   (2)
        )
        .await?;

    let mut buf = Vec::new();                                (3)
    while let Some(next) = res.data().await {               (4)
        let chunk = next?;
        buf.extend_from_slice(&chunk);                      (5)
    }
    let s = String::from_utf8(buf)?;                        (6)

    eprintln!("Status: {}", Paint::greenres.status());      (7)
    if res.headers().contains_key(CONTENT_TYPE) {
        let content_type = res.headers()[CONTENT_TYPE].to_str()?;
    }
}
```

```

eprintln!("Content-Type: {}", Paint::green(content_type));      (8)
if content_type.starts_with("application/json") {
    println!("{}", &s.to_colored_json_auto()?);                  (9)
} else {
    println!("{}", &s);                                         (10)
}
} else {
    println!("{}", &s);                                         (11)
}

Ok(())
}

```

Здесь:

- в поз. (1) — предположение о том, что тело запроса, если оно присутствует, всегда имеет формат JSON;
- в поз. (2) — включение тела запроса, если оно предоставлено. В противном случае отправка пустого запроса;
- в поз. (3) — использование `Vec` в качестве буфера для обработки входящего фрагментированного ответа;
- в поз. (4) — пофрагментное считывание ответа с добавлением каждого фрагмента в наш буфер;
- в поз. (5) — добавление в буфер каждого нового поступающего фрагмента;
- в поз. (6) — создание после прочтения всего ответа из буфера строки UTF-8 и ее потребление без необходимости создания копии;
- в поз. (7) — вывод состояния ответа в стандартный поток ошибок и использование крейта `yansi` для печати в формате ANSI с цветовым оформлением;
- в поз. (8) — отправка в стандартный вывод заголовка `Content-Type`, если он существует в ответе;
- в поз. (9) — использование крейта `coloured_json` для наглядного отображения информации в формате JSON на стандартном выводе, если типом контента является JSON;
- в поз. (10) — если тип содержимого не относится к JSON, отправка его на стандартный вывод в виде простой строки;
- в поз. (11) — если в ответе отсутствует заголовок `Content-Type`, отправка ответа на стандартный вывод в виде простой строки;
- в поз. (12) — если выполнение кода дошло до этого места, значит, запрос выполнен успешно, и поэтому здесь возвращается единица.

Заметьте, что наш запрос всегда приводит к выводу тела ответа на стандартный вывод, но состояние запроса и заголовок типа содержимого выводятся на стандартную ошибку. Разделение тела ответа и метаданных позволяет передавать вывод нашей команды в другой инструмент.

10.6. Надлежащая обработка ошибок

В листинге 10.6 возвращаемым типом является `Result` и активно задействуется оператор `?`. Кроме того, мы полагаемся на объекты типажей, используя:

```
Box<dyn std::error::Error + Send + Sync>
```

в качестве возвращаемого типа ошибки. Это удобный, но всё же несколько ленивый способ обработки ошибок. В нашем конкретном случае имеет смысл ничего не усложнять (т. е. придерживаться принципа KISS³), но если мы попадем в ситуацию, при которой потребуется более сложная логика обработки ошибок, или захотим настроить регистрацию ошибок или обработку сообщений об ошибках, то нам, скорее всего, придется создать собственный тип ошибки и воспользоваться для преобразования ошибок типажом `From`.

Кроме того, наша функция `main()`, показанная в листинге 10.4, возвращает тот же тип:

```
Result<(), Box<dyn std::error::Error + Send + Sync>>
```

Поэтому для корректного отображения ошибок оператором `?` можно будет пользоваться во всей программе.

10.7. Тестирование нашего CLI

Давайте наконец-то протестируем наш CLI-инструмент, запустив его с нашим API-сервисом из предыдущей главы. Для демонстрации работы каждой команды в следующих примерах будет открываться терминал, разбитый на две части, где в левой части выведен API-сервис, а в правой — запущен только что созданный CLI-инструмент. Сначала, как показано на рис. 10.1, мы создадим новую todo-запись, воспользовавшись для этого командой:

```
cargo run --http://localhost:3000 create "finish writing chapter 10"
```

Отлично! Обратите внимание на красиво отформатированный вывод с цветовым оформлением (это можно увидеть в электронной версии книги). Давайте попробуем протестировать остальные четыре команды, начиная с:

```
cargo run --http://localhost:3000 list
```

результат выполнения которой показан на рис. 10.2.

Далее, как показано на рис. 10.3, мы обновим todo-запись. И сделаем это, изменив тело и отметив todo-задание выполненным, для чего воспользуемся командой:

```
cargo run - http://localhost:3000 update 1 "finish writing chapter 10" --completed
```

А теперь давайте, как показано на рис. 10.4, считаем нашу обновленную todo-запись, для чего воспользуемся командой:

```
cargo run --http://localhost:3000 read 1
```

³ KISS (акроним от «Keep it simple, stupid», «Делай проще, тупица») — принцип проектирования, утверждающий, что большинство систем работают лучше всего, если они остаются простыми, а не усложняются.

```

● ● ● ℹ️ brenden@fruit-computer:~/dev/code-like-a-pro-in-rust-book/c10/api-client

X cargo (api-service)

2023-08-11T17:54:28.704741Z INFO sqlx::query: INSERT INTO _sqlx_migr
actions ( _, rows affected: 1, rows returned: 0, elapsed: 199.958µs
INSERT INTO
    _sqlx_migrations (
        version,
        description,
        success,
        checksum,
        execution_time
    )
VALUES
    (?1, ?2, TRUE, ?3, -1)

2023-08-11T17:54:28.706076Z INFO sqlx::query: UPDATE _sqlx_migration
s SET execution_time = ?; rows affected: 1, rows returned: 0, elapsed:
350.208µs
UPDATE
    _sqlx_migrations
SET
    execution_time = ?1
WHERE
    version = ?2

2023-08-11T17:54:54.279076Z DEBUG request{method=POST uri=/v1/todos v
ersion=HTTP/1.1}: tower_http::trace::on_request: started processing r
equest
2023-08-11T17:54:279829Z DEBUG request{method=POST uri=/v1/todos v
ersion=HTTP/1.1}: tower_http::trace::on_response: finished processing
request latency=0 ms status=200
2023-08-11T17:54:54.280535Z INFO sqlx::query: insert into todos (body
)_; rows affected: 1, rows returned: 1, elapsed: 649.334µs
insert into
    todos (body)
values
    (?) returning *

```

Рис. 10.1. Создание todo-записи с помощью нашего CLI-инструмента

```

● ● ● ℹ️ brenden@fruit-computer:~/dev/code-like-a-pro-in-rust-book/c10/api-client

X cargo (api-service)

success,
checksum,
execution_time
)
VALUES
    (?1, ?2, TRUE, ?3, -1)

2023-08-11T17:54:28.706076Z INFO sqlx::query: UPDATE _sqlx_migration
s SET execution_time = ?; rows affected: 1, rows returned: 0, elapsed:
350.208µs
UPDATE
    _sqlx_migrations
SET
    execution_time = ?1
WHERE
    version = ?2

2023-08-11T17:54:54.279076Z DEBUG request{method=POST uri=/v1/todos v
ersion=HTTP/1.1}: tower_http::trace::on_request: started processing r
equest
2023-08-11T17:54:279829Z DEBUG request{method=POST uri=/v1/todos v
ersion=HTTP/1.1}: tower_http::trace::on_response: finished processing
request latency=0 ms status=200
2023-08-11T17:54:54.280535Z INFO sqlx::query: insert into todos (body
)_; rows affected: 1, rows returned: 1, elapsed: 649.334µs
insert into
    todos (body)
values
    (?) returning *

2023-08-11T17:55:23.219826Z DEBUG request{method=GET uri=/v1/todos ve
rsion=HTTP/1.1}: tower_http::trace::on_request: started processing re
quest
2023-08-11T17:55:23.220242Z INFO sqlx::query: select * from todos; r
ows affected: 1, rows returned: 1, elapsed: 137.709µs
2023-08-11T17:55:23.220415Z DEBUG request{method=GET uri=/v1/todos ve
rsion=HTTP/1.1}: tower_http::trace::on_response: finished processing
request latency=0 ms status=200

```

Рис. 10.2. Вывод списка todo-записей с помощью нашего CLI-инструмента

```

● ● ● ℗ 22 brenden@fruit-computer:~/dev/code-like-a-pro-in-rust-book/c10/api-client
X cargo (api-service)
2023-08-11T17:54:54.279076Z DEBUG request{method=POST uri=/v1/todos v
ersion=HTTP/1.1}: tower_http::trace::on_request: started processing r
equest
2023-08-11T17:54:54.279829Z DEBUG request{method=POST uri=/v1/todos v
ersion=HTTP/1.1}: tower_http::trace::on_response: finished processing
request latency=0 ms status=200
2023-08-11T17:54:54.280535Z INFO sqlx::query: insert into todos (body
y) -; rows affected: 1, rows returned: 1, elapsed: 649.334µs

insert into
  todos (body)
values
  (?) returning *

2023-08-11T17:55:23.219826Z DEBUG request{method=GET uri=/v1/todos ve
rsion=HTTP/1.1}: tower_http::trace::on_request: started processing re
quest
2023-08-11T17:55:23.220424Z INFO sqlx::query: select * from todos; r
ows affected: 1, rows returned: 1, elapsed: 137.709µs
2023-08-11T17:55:23.220415Z DEBUG request{method=GET uri=/v1/todos ve
rsion=HTTP/1.1}: tower_http::trace::on_response: finished processing
request latency=0 ms status=200
2023-08-11T17:56:09.493295Z DEBUG request{method=PUT uri=/v1/todos/1 v
ersion=HTTP/1.1}: tower_http::trace::on_request: started processing
request
2023-08-11T17:56:09.493980Z DEBUG request{method=PUT uri=/v1/todos/1 v
ersion=HTTP/1.1}: tower_http::trace::on_response: finished processin
g request latency=0 ms status=200
2023-08-11T17:56:09.495188Z INFO sqlx::query: update todos set body
=; rows affected: 1, rows returned: 1, elapsed: 649.500µs

update
  todos
set
  body = ?,
  completed = ?,
  updated_at = datetime('now')
where
  id = ? returning *

X ..10/api-client (-zsh)
→ api-client git:(main) cargo run -- http://localhost:3000 update 1
1 "finish writing chapter 10" --completed
  Finished dev [unoptimized + debuginfo] target(s) in 0.03s
    Running `target/debug/api-client 'http://localhost:3000' update 1` "finish writing chapter 10" --completed"
Status: 200 OK
Content-Type: application/json
{
  "body": "finish writing chapter 10",
  "completed": true,
  "created_at": "2023-08-11T17:54:54",
  "id": 1,
  "updated_at": "2023-08-11T17:56:09"
}
→ api-client git:(main) █

```

Рис. 10.3. Обновление todo-записи с помощью нашего CLI-инструмента

```

● ● ● ℗ 22 brenden@fruit-computer:~/dev/code-like-a-pro-in-rust-book/c10/api-client
X cargo (api-service)
que
2023-08-11T17:55:23.220424Z INFO sqlx::query: select * from todos; r
ows affected: 1, rows returned: 1, elapsed: 137.709µs
2023-08-11T17:55:23.220415Z DEBUG request{method=GET uri=/v1/todos ve
rsion=HTTP/1.1}: tower_http::trace::on_response: finished processing
request latency=0 ms status=200
2023-08-11T17:56:09.493295Z DEBUG request{method=PUT uri=/v1/todos/1 v
ersion=HTTP/1.1}: tower_http::trace::on_request: started processing
request
2023-08-11T17:56:09.493980Z DEBUG request{method=PUT uri=/v1/todos/1 v
ersion=HTTP/1.1}: tower_http::trace::on_response: finished processin
g request latency=0 ms status=200
2023-08-11T17:56:09.495188Z INFO sqlx::query: update todos set body
=; rows affected: 1, rows returned: 1, elapsed: 649.500µs

update
  todos
set
  body = ?,
  completed = ?,
  updated_at = datetime('now')
where
  id = ? returning *

2023-08-11T17:56:39.360289Z DEBUG request{method=GET uri=/v1/todos/1 v
ersion=HTTP/1.1}: tower_http::trace::on_request: started processing
request
2023-08-11T17:56:39.360831Z DEBUG request{method=GET uri=/v1/todos/1 v
ersion=HTTP/1.1}: tower_http::trace::on_response: finished processin
g request latency=0 ms status=200
2023-08-11T17:56:39.361218Z INFO sqlx::query: select * from todos --;
rows affected: 1, rows returned: 1, elapsed: 143.625µs

select
  *
from
  todos
where
  id = ?

X ..10/api-client (-zsh)
→ api-client git:(main) cargo run -- http://localhost:3000 read 1
1 "Finished dev [unoptimized + debuginfo] target(s) in 0.03s
  Running `target/debug/api-client 'http://localhost:3000' read 1`"
Status: 200 OK
Content-Type: application/json
{
  "body": "finish writing chapter 10",
  "completed": true,
  "created_at": "2023-08-11T17:54:54",
  "id": 1,
  "updated_at": "2023-08-11T17:56:09"
}
→ api-client git:(main) █

```

Рис. 10.4. Чтение todo-записи с помощью нашего CLI-инструмента

```

brenden@fruit-computer:~/dev/code-like-a-pro-in-rust-book/c10/api-client
X cargo (api-service)
  todos
set
  body = ?,
  completed = ?,
  updated_at = datetime('now')
where
  id = ? returning *

2023-08-11T17:56:39.360289Z DEBUG request{method=GET uri=/v1/todos/1
version=HTTP/1.1}: tower_http::trace::on_request: started processing
request
2023-08-11T17:56:39.360831Z DEBUG request{method=GET uri=/v1/todos/1
version=HTTP/1.1}: tower_http::trace::on_response: finished processin
g request latency=0 ms status=200
2023-08-11T17:56:39.361218Z INFO sqlx::query: select * from todos --
rows affected: 1, rows returned: 1, elapsed: 143.625µs

select
  *
from
  todos
where
  id = ?

2023-08-11T17:57:13.324809Z DEBUG request{method=GET uri=/v1/todos/1
version=HTTP/1.1}: tower_http::trace::on_request: started processing
request
2023-08-11T17:57:13.325130Z DEBUG request{method=GET uri=/v1/todos/1
version=HTTP/1.1}: tower_http::trace::on_response: finished processin
g request latency=0 ms status=200
2023-08-11T17:57:13.325577Z INFO sqlx::query: select * from todos --
rows affected: 1, rows returned: 1, elapsed: 47.416µs

select
  *
from
  todos
where
  id = ?

```

Рис. 10.5. Передача вывода нашего CLI-инструмента в jq

```

brenden@fruit-computer:~/dev/code-like-a-pro-in-rust-book/c10/api-client
X cargo (api-service)
2023-08-11T17:56:39.361218Z INFO sqlx::query: select * from todos --
rows affected: 1, rows returned: 1, elapsed: 143.625µs

select
  *
from
  todos
where
  id = ?

2023-08-11T17:57:13.324809Z DEBUG request{method=GET uri=/v1/todos/1
version=HTTP/1.1}: tower_http::trace::on_request: started processing
request
2023-08-11T17:57:13.325130Z DEBUG request{method=DELETE uri=/v1/todos/1
version=HTTP/1.1}: tower_http::trace::on_request: started processi
ng request
2023-08-11T17:57:13.325577Z INFO sqlx::query: delete from todos wher
e --; rows affected: 1, rows returned: 0, elapsed: 679.042µs

delete from
  todos
where
  id = ?

2023-08-11T17:57:35.262722Z DEBUG request{method=DELETE uri=/v1/todos
/1 version=HTTP/1.1}: tower_http::trace::on_response: finished proces
sing request latency=1 ms status=200

```

Рис. 10.6. Удаление todo-записи с помощью нашего CLI-инструмента

Мы можем также протестировать конвейеризацию вывода нашего CLI-инструмента в другую команду — например, `jq`, воспользовавшись командой:

```
cargo run --http://localhost:3000 read 1 | jq '.body'
```

В результате, как показано на рис. 10.5, из нашего вывода в формате JSON будет выполнена выборка поля `body`.

Обратите внимание на то, как Cargo удобно отправляет свой вывод в стандартную ошибку (вместо стандартного вывода), позволяя нам всё же воспользоваться конвейерами с командой `cargo run --....` И, наконец, давайте удалим нашу todo-запись с помощью команды:

```
cargo run --http://localhost:3000 delete 1
```

действие которой показано на рис. 10.6.

На этом демонстрация CLI-инструмента для нашего API-сервиса завершается. Этим кодом можно воспользоваться в качестве шаблона или отправной точки для любого из ваших будущих проектов, хотя в качестве упражнения для читателей я бы рекомендовал заменить крейт `hyper` на `reqwest`.

Резюме

- Создание надежных CLI-инструментов в Rust дается на удивление легко и просто. Использование API с безопасными типами (например, `clap`) для анализа аргументов CLI существенно упрощает создание отличных и весьма быстродействующих инструментов.
- Благодаря имеющейся в Rust богатой экосистеме крейтов, нам не приходится прилагать слишком много усилий для добавления в наш CLI-инструмент таких обогащенных функций, как создание четко отформатированного и легко читаемого информационного вывода. Большая часть всего этого нам достается совершенно бесплатно за счет использования крейтов `yansi` и `coloured_json`.
- HTTP-библиотека `hyper` представляет собой низкоуровневую реализацию HTTP в Rust, позволяющую реализовать как клиентский, так и серверный API, но всё же на практике может потребоваться использование для HTTP API-интерфейса более высокого уровня: `axum` — для HTTP-серверов и `reqwest` — для HTTP-клиентов.
- Если вам не захочется самостоятельно заниматься обработкой ошибок, можно воспользоваться объектами типажей с `Box<dyn std::error::Error + Send + Sync>` в качестве типа ошибки. Это будет работать при условии, что всеми встречающимися вам типами ошибок реализуется `std::error::Error`. Эта функциональная возможность предоставляется также еще несколькими крейтами, среди которых `thiserror`⁴ и `Anyhow`⁵.

⁴ См. <https://crates.io/crates>thiserror>.

⁵ См. <https://crates.io/crates/anyhow>.

Часть V

Оптимизация

Может настать момент, требующий повышения производительности вашего программного продукта сверх того, чего можно достичь просто за счет качественного проектирования с использованием подходящих структур данных и применением совершенных алгоритмов. Современные операционные системы, процессоры и компиляторы превосходно справляются за вас с большей частью этой работы, но временами всё же приходится вникать во все тонкости этого процесса еще глубже.

Сама по себе тема оптимизации кода заслуживает целой книги, но в качестве неплохой отправной точки я позволил себе выделить в единственной главе этой части самую суть целого ряда ее ключевых моментов. Присущие Rust безопасность, конкурентная и параллельная работа, асинхронность и SIMD-возможности¹ делают его чрезвычайно привлекательным языком программирования, и всё это представлено нам в виде чистого программного средства с открытым исходным кодом.

¹ SIMD (Single Instruction, Multiple Data — одиночный поток команд, множественный поток данных) — принцип компьютерных вычислений, позволяющий обеспечить параллелизм на уровне данных.

11

Оптимизация кода

В этой главе:

- понятие абстракций Rust с нулевой стоимостью;
- эффективное использование векторов;
- программирование в Rust с применением SIMD-возможностей;
- распараллеливание с помощью Rayon;
- использование Rust для ускорения других языков.

В этой, заключительной главе книги мы рассмотрим доступные в Rust стратегии оптимизации, имея в виду, что абстракции Rust с нулевой стоимостью позволяют уверенно создавать код на этом языке, особо не задумываясь о производительности, поскольку основную часть генерации машинного кода Rust делегирует проекту LLVM, имеющему вполне состоявшуюся, надежную, широко распространенную и хорошо протестированную возможность оптимизации кода. Код, написанный на Rust, будет и без затрат времени на его ручную настойку быстрым и хорошо оптимизированным.

Однако возникают определенные ситуации, когда вам может потребоваться более глубокое погружение в оптимизацию кода, и мы обсудим их здесь вместе с инструментами, которые для этого понадобятся. Мы также посмотрим, как можно использовать возможности Rust для ускорения кода, написанного на других языках программирования, что представляет собой интересный способ внедрения Rust в их кодовые базы без необходимости выполнять полное переписывание кода.

11.1. Абстракции с нулевой стоимостью

Уникальной особенностью Rust является использование *абстракций с нулевой стоимостью*. Не особо углубляясь в теорию, можно лишь отметить, что Rust-абстракции позволяют создавать высокоуровневый код, на основе которого производится оптимизированный машинный код, свободный при выполнении програм-

мы от каких-либо накладных расходов. Все заботы по оптимальному переходу от высокоуровневого кода Rust к низкоуровневому машинному коду, не имеющему накладных расходов, берет на себя компилятор Rust. Вы можете абсолютно безопасно пользоваться Rust-абстракциями, не беспокоясь о том, что ими будет нанесен какой-либо ущерб производительности.

Платой за возможность реализовать в Rust абстракции нулевой стоимости является отсутствие в этом языке ряда присущих высокоуровневым языкам функций, которые, возможно, вы ожидали в нем найти, или же как минимум их присутствие в нем в непривычной форме. Некоторые из таких функций включают виртуальные методы, рефлексию, перегрузку функций и optionalные аргументы функций. Rust предоставляет альтернативы таким привычным функциям или способы эмуляции их поведения, но они не встроены в сам язык. Если вам захочется пойти на подобные издержки, это придется сделать самостоятельно (что, конечно, значительно упростит рассуждения) — например, используя для динамической диспетчеризации объекты-типажи, как это делается с виртуальными методами.

Абстракции в Rust можно, к примеру, сравнить с абстракциями в C++, *имеющими* — в отличие от Rust — издержки времени выполнения. Так, абстракция базового класса C++ может содержать виртуальные методы, требующие в ходе выполнения программы наличия таблиц поиска (называемых *vtables*). Хотя такие издержки обычно не слишком велики, их размер в определенных случаях, например при вызове виртуальных методов в коротком цикле прохода по многим элементам, может существенно возрастать.

ПРИМЕЧАНИЕ

Имеющиеся в Rust объекты-типажи используют *vtables* для вызовов методов. Такие объекты-типажи представляют собой функциональную возможность, включаемую синтаксисом `dyn Trait`. Например, объект-типаж можно сохранять с помощью `Box<dyn MyTrait>`, где элемент в этой упаковке должен иметь реализацию типажа `MyTrait`, а методы из `MyTrait` могут вызываться с помощью поиска в *vtable*.

Еще одной скрытой абстракцией, широко используемой в некоторых языках для диспетчеризации вызовов функций или выполнения других операций в ходе выполнения программы, является *рефлексия*. Она довольно часто применяется в Java — например, для решения различных задач в ходе выполнения программы, но может привести к генерации значительного количества ошибок, слабо поддающихся отладке. То есть рефлексия предлагает программисту удобные подходы к решению задач за счет снижения надежности программного кода.

Имеющиеся в Rust абстракции с нулевой стоимостью основаны на проведении оптимизации в ходе компиляции. В рамках этого процесса Rust способен по мере необходимости оптимизировать неиспользуемый код или незадействованные значения. Абстракции Rust также могут быть глубоко вложенными, а компилятор (в большинстве случаев) в состоянии выполнять оптимизацию вниз по цепочке абстракций. То есть, когда заходит речь о имеющихся в Rust абстракциях с нулевой стоимостью, на самом деле имеются в виду *абстракции с нулевой стоимостью после выполнения всех оптимизаций*.

Когда требуется получить производственный двоичный файл или провести оценку производительности программного кода, в компиляторе нужно включить режим оптимизации и провести компиляцию с выдачей готового релиза. Сделать это можно с помощью команды `cargo` с включением флага `--release`, поскольку по умолчанию для компиляции включается режим отладки. Если вы забудете включить режим релиза, то можете на выходе получить неожиданные потери производительности, и одну из таких потерь мы рассмотрим в следующем разделе.

11.2. Векторы

Векторы являются базовой абстракцией коллекции в Rust. Как уже ранее упоминалось, потребности в коллекциях элементов должны в большинстве случаев удовлетворяться за счет использования `Vec`. Учитывая весьма частое использование `Vec` в программах на Rust, важно разбираться в подробностях реализации этого типа и в том, как он способен повлиять на производительность создаваемого кода. Кроме того, следует понимать, когда имеет смысл воспользоваться вместо `Vec` чем-то другим.

Прежде всего нам нужно разобраться с тем, как для `Vec` выделяется память. Этот вопрос уже немного рассматривался в главах 4 и 5, но здесь ему мы уделим более пристальное внимание. Память для `Vec` выделяется непрерывными блоками с настраиваемым размером фрагмента на основе общего объема. Выделение происходит в ленивом режиме с откладыванием собственно выделения до тех пор, пока в нем не возникнет насущная потребность, и это выделение, как только что отмечалось, всегда осуществляется непрерывными блоками.

11.2.1. Выделение памяти для вектора

Первое, что следует понять про выделение памяти для `Vec`, — это порядок определения ее объема. По умолчанию объем у пустого `Vec` равен нулю, и, следовательно, память под него не выделяется. Выделение памяти происходит только после добавления данных. При достижении предельного значения объем для `Vec` удваивается (т. е. объем увеличивается экспоненциально).

Проследить за добавлением объема для `Vec` можно путем выполнения небольшого теста:

```
let mut empty_vec = Vec::<i32>::new();
(0..10).for_each(|v| {
    println!(
        "empty_vec has {} elements with capacity {}",
        empty_vec.len(),
        empty_vec.capacity()
    );
    empty_vec.push(v)
});
```

Обратите внимание, что объем измеряется в количестве элементов, а не в количестве байтов. Количество байтов, необходимое для вектора, равно объему, умноженному на размер каждого элемента. При запуске приведенного кода на экран выводится следующая информация:

```
empty_vec has 0 elements with capacity 0
empty_vec has 1 elements with capacity 4 ← объем увеличен с 0 до 4
empty_vec has 2 elements with capacity 4
empty_vec has 3 elements with capacity 4
empty_vec has 4 elements with capacity 4
empty_vec has 5 elements with capacity 8 ← объем увеличен с 4 до 8
empty_vec has 6 elements with capacity 8
empty_vec has 7 elements with capacity 8
empty_vec has 8 elements with capacity 8
empty_vec has 9 elements with capacity 16 ← объем увеличен с 8 до 16
```

Чтобы увидеть сам алгоритм, являющийся частью RawVec (внутренней структуры данных, используемой Vec), можно изучить исходный код стандартной библиотеки Rust, показанный в листинге 11.1.

Листинг 11.1. Vec::grow_amortized() из стандартной библиотеки Rust

```
// This method is usually instantiated many times. So we want it to be as small as possible,
// to improve compile times. But we also want as much of its contents to be statically
// computable as possible, to make the generated code run faster. Therefore, this method
// is carefully written so that all of the code that depends on `T` is within it,
// while as much of the code that doesn't depend on `T` as possible is in functions that
// are non-generic over `T`.
fn grow_amortized(&mut self, len: usize, additional: usize) ->
Result<(), TryReserveError> {
    // This is ensured by the calling contexts.
    debug_assert!(additional > 0);

    if mem::size_of::<T>() == 0 {
        // Since we return a capacity of `usize::MAX` when `elem_size` is
        // 0, getting to here necessarily means the `RawVec` is overfull.
        return Err(CapacityOverflow.into());
    }

    // Nothing we can really do about these checks, sadly.
    let required_cap = len.checked_add(additional).ok_or(CapacityOverflow)?;

    // This guarantees exponential growth. The doubling cannot overflow
    // because `cap <= isize::MAX` and the type of `cap` is `usize`.
    let cap = cmp::max(self.cap * 2, required_cap); (1)
    let cap = cmp::max(Self::MIN_NON_ZERO_CAP, cap); (2)
```

```
let new_layout = Layout::array::array(<T>(cap));  
// `finish_grow` is non-generic over `T`.  
let ptr = finish_grow(new_layout, self.current_memory(), &mut self.alloc)?;  
self.set_ptr_and_cap(ptr, cap);  
Ok(())  
}
```

Здесь:

- в поз. (1) — тут объем (`self.cap`) удваивается;
- в поз. (2) — `Self::MIN_NON_ZERO_CAP` варьируется в зависимости от размера элементов, но может иметь значение 8, 4 или 1.

Что можно извлечь из этой информации? Есть два основных вывода:

- ленивое распределение памяти для `Vec` может быть неэффективным, если добавляется много элементов по несколько за раз;
- для крупных векторов объем будет двойным для того количества элементов, которое имеется в массиве.

Первый вывод говорит о том, что такое (ленивое) распределение может создавать проблемы при частых созданиях новых векторов и помещении в них данных. Пере распределения тогда обходятся слишком дорого, поскольку они могут быть связаны с масштабным перестроением памяти. И хотя перераспределения при небольшом количестве элементов не обходятся так дорого, поскольку используемая машина, вероятно, имеет вполне достаточно памяти для выделения небольших смежных областей, однако по мере роста структуры находить доступные смежные области может быть всё труднее (соответственно потребуется больше перестроений памяти).

Вторую проблему — с крупными векторами — можно смягчить, либо используя другую структуру (например, связанный список), либо сохраняя объем обрезанным с помощью метода `Vec::shrink_to_fit()`. Стоит также отметить, что векторы могут быть крупными в двух разных измерениях: при большом количестве мелких элементов или при небольшом количестве крупных элементов. В последнем случае (при нескольких крупных элементах) снизить нагрузку при работе с памятью может позволить применение связанного списка или хранение элементов в `Box`.

11.2.2. Итераторы векторов

Еще одним важным моментом, учитываемым при рассмотрении вопроса производительности применительно к `Vec`, является последовательный перебор элементов. Есть два способа выполнения циклического перебора элементов `Vec` — с помощью либо `iter()`, либо `into_iter()`. Итератор `iter()` позволяет выполнять итерацию по элементам со ссылками, а `into_iter()` получает `self`. Чтобы разобраться с итераторами `Vec`, рассмотрим код листинга 11.2.

Листинг 11.2. Демонстрация производительности итератора Vec

```
let big_vec = vec![0; 10_000_000];
let now = Instant::now();
for i in big_vec {
    if i < 0 {
        println!("this never prints");
    }
}
println!("First loop took {}s", now.elapsed().as_secs_f32());

let big_vec = vec![0; 10_000_000];
let now = Instant::now();
big_vec.iter().for_each(|i| {
    if *i < 0 {
        println!("this never prints");
    }
});
println!("Second loop took {}s", now.elapsed().as_secs_f32());
```

В этом коде имеется несколько больших векторов, перебор элементов которых мы намерены осуществить с помощью блока кода без операций, чтобы компилятор его не оптимизировал. Протестируем код с запуском команды:

`cargo run --release`

поскольку нам нужно включить все режимы оптимизации, присущие компилятору. После запуска кода на экран будет выдана следующая информация:

```
First loop took 0.007614s
Second loop took 0.00410025s
```

Стоп! Так что же там произошло? Почему цикл `for` работает почти в два раза медленнее?

Ответ кроется в применении синтаксического средства — выражение цикла `for` просто превращается в использование метода `into_iter()`, чтобы был получен итератор и выполнен цикл по итератору до исчерпания элементов¹. Метод `into_iter()` по умолчанию принимает `self` — т. е. он получает исходный вектор и (в ряде случаев) может даже потребовать выделения памяти под совершенно новую структуру.

А вот метод `for_loop()`, предоставляемый в Rust типажом базового итератора, хорошо оптимизирован для этой цели, что дает небольшой прирост производительности. Кроме того, `iter()` принимает `&self` и перебирает ссылки на имеющиеся в векторе элементы, что может быть дополнительно оптимизировано компилятором.

Чтобы проверить сказанное, можно обновить код, использовав `into_iter()` вместо `iter()`.

¹ Полное выражение задокументировано по адресу: <http://mng.bz/W1nd>.

Попробуем добавить третий цикл:

```
let big_vec = vec![0; 10_000_000];
let now = Instant::now();
big_vec.into_iter().for_each(|i| {
    if i < 0 {
        println!("this never prints");
    }
});
println!("Third loop took {}s", now.elapsed().as_secs_f32());
```

и снова запустить код в режиме релиза, чтобы получить следующий результат:

```
First loop took 0.011229166s
Second loop took 0.005076166s
Third loop took 0.008608s
```

Результаты гораздо ближе друг к другу, но, похоже, что быстродействие при непосредственном использовании итераторов вместо выражений цикла `for` немного выше. А что получится, если этот же тест запустить в режиме отладки? Давайте посмотрим:

```
First loop took 0.074964s
Second loop took 0.14158678s
Third loop took 0.07878621s
```

Надо же, результаты кардинально отличаются! Интереснее всего то, что в режиме отладки быстродействие циклов `for` немного выше. Скорее всего, так получается из-за дополнительных издержек, налагаемых включением отладочных символов и отключением оптимизаций компилятора. Из этого следует вывод, что замеры производительности в режиме отладки могут привести к весьма странным результатам.

При использовании векторов включается довольно много других встроенных способов оптимизации, но нас обычно интересуют только те, что связаны с выделением памяти и итераторами. Требуемые объемы выделения памяти могут быть уменьшены за счет предварительного выделения памяти с помощью применения метода `Vec::with_capacity()`, а малопонятных проблем с производительностью можно избежать путем непосредственного использования итераторов, а не выражения цикла `for`.

11.2.3. Быстрое копирование с помощью `Vec` и слайсов

Рассмотрим здесь еще один способ оптимизации, связанный с векторами и копированием памяти. В Rust имеется средство оптимизации, позволяющее выбрать быстрый путь для векторов и слайсов и при определенных обстоятельствах выполнять более быстрое копирование всего, что находится внутри `Vec`. Таким средством оптимизации служит метод `Vec::copy_from_slice()`, ключевые моменты реализации которого показаны в коде листинга 11.3.

Листинг 11.3. Листинг фрагмента метода copy_from_slice() из стандартной библиотеки Rust

```
pub fn copy_from_slice(&mut self, src: &[T])
    where
        T: Copy,
    {
        // ... snip ...
        unsafe {
            ptr::copy_nonoverlapping(src.as_ptr(), self.as_mut_ptr(), self.len());
        }
    }
}
```

В этом фрагменте, взятом из стандартной библиотеки Rust, следует отметить два важных момента: привязку типажа `Copy` и небезопасный вызов `ptr::copy_nonoverlapping`. Иными словами, если при использовании `Vec` нужно выполнить копирование элементов между двумя векторами, при условии, что в этих элементах реализован типаж `Copy`, можно пойти по быстрому пути. Чтобы заметить разницу, можно запустить средство быстрой оценки производительности, код которого показан в листинге 11.4.

Листинг 11.4. Оценка производительности ptr::copy_nonoverlapping()

```
let big_vec_source = vec![0; 10_000_000];
let mut big_vec_target = Vec::<i32>::with_capacity(10_000_000);           (1)
let now = Instant::now();
big_vec_source
    .into_iter()
    .for_each(|i| big_vec_target.push(i));
println!("Naive copy took {}s", now.elapsed().as_secs_f32());

let big_vec_source = vec![0; 10_000_000];
let mut big_vec_target = vec![0; 10_000_000];
let now = Instant::now();
big_vec_target.copy_from_slice(&big_vec_source);
println!("Fast copy took {}s", now.elapsed().as_secs_f32());
```

- Здесь в поз. (1) — инициализация `Vec` с предварительно выделенной памятью нужного объема.

Выполнение этого кода в режиме релиза дает следующий результат:

```
Naive copy took 0.024926165s
Fast copy took 0.003599458s
```

Получается, что использование `Vec::copy_from_slice()` даст нам примерно 8-кратное ускорение непосредственного копирования данных из одного вектора в другой. Такое же средство оптимизации имеется и для типов слайсов (`&mut [T]`), и для массивов (`mut [T]`).

11.3. Применение возможностей SIMD

В практике любого разработчика может настать момент, требующий использования *одиночного потока команд при множестве потоков данных* (Single Instruction, Multiple Data, SIMD). SIMD относится к аппаратной функции многих современных микропроцессоров, позволяющей одновременно выполнять операции с наборами данных с помощью одной инструкции. Наиболее распространенным вариантом использования такой системы является оптимизация кода для конкретного процессора или обеспечение согласованного времени выполнения операций (например, для предотвращения синхронных атак в криптографических приложениях).

Конкретика SIMD зависит от платформы: у разных центральных процессоров имеются разные функции SIMD, но те или иные SIMD-функции имеются почти у всех современных процессоров. К наиболее часто используемым инструкциям SIMD относятся MMX, SSE и AVX на устройствах Intel, а также Neon на устройствах ARM.

Раньше необходимость прибегать к SIMD влекла за собой самостоятельное создание кода на встроенном ассемблере. К счастью, современные компиляторы предоставляют интерфейс, позволяющий задействовать SIMD без непосредственного создания ассемблерного кода. Такие функциональные возможности приводят в некоторой степени к переносимости стандарта общего поведения при использовании различных реализаций SIMD. Преимущество переносимой SIMD-технологии заключается в отсутствии необходимости беспокоиться об особенностях набора инструкций для какой-либо конкретной платформы, а платой за это служит доступ только к функциям, приведенным к общему знаменателю. При желании можно по-прежнему создавать код на встроенном ассемблере, но я намерен здесь сконцентрировать ваше внимание на переносимой SIMD-технологии. Одной из удобных особенностей переносимой SIMD является то, что компилятор способен автоматически генерировать замещающий не-SIMD код в тех случаях, когда соответствующие функции на аппаратном уровне недоступны.

Стандартной библиотекой Rust для SIMD является модуль `std::simd`, который на текущий момент относится к категории нового экспериментального API².

В качестве иллюстрации можно провести сравнительную оценку скорости выполнения некоторых математических операций на 64-элементных массивах с SIMD-технологией и без нее (листинг 11.5).

Листинг 11.5. Умножение векторов с применением SIMD в сравнении с использованием итераторов

```
#![feature(portable_simd, array_zip)] (1)

fn initialize() -> ([u64; 64], [u64; 64]) {
    let mut a = [0u64; 64];
```

² См. <https://doc.rust-lang.org/std/simd/struct.Simd.html>.

```

let mut b = [0u64; 64];
(0..64).for_each(|n| {
    a[n] = u64::try_from(n).unwrap();
}); b[n] = u64::try_from(n + 1).unwrap();
(a, b)
}
fn main() {
    use std::simd::Simd;
    use std::time::Instant;

    let (mut a, b) = initialize();                                (2)

    let now = Instant::now();                                     (3)
    for _ in 0..100_000 {
        let c = a.zip(b).map(|(l, r)| l * r);
        let d = a.zip(c).map(|(l, r)| l + r);
        let e = c.zip(d).map(|(l, r)| l * r);
        a = e.zip(d).map(|(l, r)| l ^ r);                      (4)
    }
    println!("Without SIMD took {}s", now.elapsed().as_secs_f32());
}

let (a_vec, b_vec) = initialize();                                (5)

let mut a_vec = Simd::from(a_vec);                               (6)
let b_vec = Simd::from(b_vec);

let now = Instant::now();                                     (7)
for _ in 0..100_000 {
    let c_vec = a_vec * b_vec;
    let d_vec = a_vec + c_vec;
    let e_vec = c_vec * d_vec;
    a_vec = e_vec ^ d_vec;                                    (8)
}
println!("With SIMD took {}s", now.elapsed().as_secs_f32());

assert_eq!(&a, a_vec.as_array());                                (9)
}

```

Здесь:

- в поз. (1) — включение экспериментальных функций для этого крейта;
- в поз. (2) — инициализация наших 64-элементных массивов;
- в поз. (3) — выполнение расчетов с использованием обычной математики;
- в поз. (4) — сохранение результата в a;
- в поз. (5) — повторная инициализация с теми же значениями;
- в поз. (6) — преобразование наших массивов в SIMD-вектор;

- в поз. (7) — выполнение тех же вычислений с применением SIMD;
- в поз. (8) — сохранение результата в `a_vec`;
- в поз. (9) — в завершение: проверка, имеют ли `a` и `a_vec` один и тот же результат.

Выполнение этого кода даст следующий результат:

Without SIMD took 0.07886646s

With SIMD took 0.002505291s

Удивительно! У нас с применением SIMD получилось почти что 40-кратное ускорение. Кроме того, код SIMD обеспечивает согласованную синхронизацию, весьма важную для тех приложений, работа которых от нее зависит, — например, для криптографии.

11.4. Распараллеливание с применением Rayon

Решение задач, позволяющих повысить производительность за счет *распараллеливания* (т. е. путем параллельного программирования на нескольких потоках), лучше всего начинать с использования крейта Rayon. Хотя в Rust, конечно же, предоставляется возможность потоковой обработки, которая является частью базовой библиотеки этого языка, но она несколько примитивна, и в большинстве случаев код лучше создавать на основе API более высокого уровня.

Rayon предоставляет два способа параллельной работы с данными — это реализация параллельного итератора и ряд вспомогательных средств для создания на основе потоков легких заданий. Наше внимание будет в основном уделено итератору Rayon, поскольку он является самой полезной частью библиотеки.

Обычно Rayon используется при наличии значительного количества задач, на решение которых уходит относительно много времени или требуются значительные вычислительные затраты. При небольшом количестве задач или их нетребовательности к существенным вычислительным затратам внедрение параллельного выполнения кода, скорее всего, просто снизит производительность. В целом распараллеливание из-за проблем синхронизации и нехватки данных, которые могут проявляться при перемещении данных между потоками, характеризуется убывающей отдачей по мере увеличения количества потоков (рис. 11.1).

Одной из удобных особенностей итераторов Rayon является их чуть ли не полная совместимость с базовым типажом `Iterator`, позволяющая очень просто оценивать производительность кода с применением и без применения распараллеливания. Давайте это продемонстрируем, создав два разных теста: один, который будет с Rayon медленнее, а другой — который с Rayon будет быстрее.

В начало теста поместим код, который будет выполняться быстрее *без Rayon*:

```
let start = Instant::now();
let sum = data
    .iter()
    .map(|n| n.wrapping_mul(*n))
    .reduce(|a: i64, b: i64| a.wrapping_add(b));
```

(1)

```

let finish = Instant::now() - start;
println!(
    "Summing squares without rayon took {}s",
    finish.as_secs_f64()
);

let start = Instant::now();
let sum = data
    .par_iter()
    .map(|n| n.wrapping_mul(*n))
    .reduce(|| 0, |a: i64, b: i64| a.wrapping_add(b));           (2)
let finish = Instant::now() - start;
println!("Summing squares with rayon took {}s", finish.as_secs_f64());

```

Здесь:

- в поз. (1) — `reduce()` используется вместо `sum()`, поскольку `sum()` выполняет сложение без обработки переполнения;
- в поз. (2) — заметьте, что с Rayon, требующим значения идентичности, которое может быть вставлено для создания возможностей распараллеливания, сигнатура `reduce()` имеет некоторые различия.

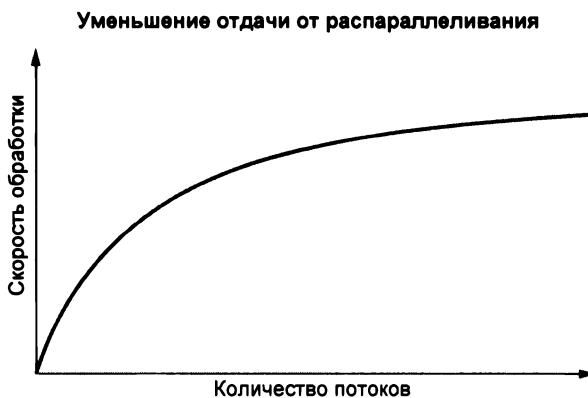


Рис. 11.1. Уменьшение отдачи при увеличении распараллеливания

В этом фрагменте кода был сгенерирован массив, заполненный случайными целочисленными значениями. Затем каждое значение возводилось в квадрат и вычислялась сумма по всему набору. Это классический пример `map/reduce` (отображение/уменьшение). Запуск кода приводит к следующему результату:

```

Summing squares without rayon took 0.000028875s
Summing squares with rayon took 0.000688583s

```

На тест с применением Rayon затрачивается в 23 раза больше времени, чем без него! Получается, что это явно не лучший кандидат для распараллеливания.

Давайте создадим еще один тест, который для сканирования длинной строки на наличие слова будет использовать регулярное выражение. Перед запуском поиска здесь будут сгенерированы произвольным образом несколько очень больших строк:

```
let re = Regex::new(r"catdog").unwrap();           ← Regex — это и Send, и Sync. Он может использоваться
                                                    с Rayon в параллельном фильтре
let start = Instant::now();
let matches: Vec<_> = data.iter().filter(|s| re.is_match(s)).collect();
let finish = Instant::now() - start;
println!("Regex took {}s", finish.as_secs_f64());
```



```
let start = Instant::now();
let matches: Vec<_> =
    data.par_iter().filter(|s| re.is_match(s)).collect();
let finish = Instant::now() - start;
println!("Regex with rayon took {}s", finish.as_secs_f64());
```

Запуск кода приведет к следующему результату:

```
Regex took 0.04357333s
Regex with rayon took 0.006173s
```

В этом случае распараллеливание с Rayon дает нам ускорение в 7 раз. Сканирование строк — одна из тех задач, для которых при достаточно больших наборах данных будет заметен существенный прирост производительности.

Еще одной примечательной особенностью Rayon является предоставляемая им возможность распараллеливания сортировки слайсов. При этом для больших наборов данных может быть обеспечен приличный прирост производительности. Имеющаяся в Rayon функция `join()` также обеспечивает реализацию перехвата работы, выполняющего задачи в параллельном режиме при доступности свободных рабочих потоков, но вместо нее следует по возможности использовать параллельные итераторы. Более подробную информацию о Rayon можно найти в его документации³.

11.5. Использование Rust для ускорения программ на других языках

Последняя тема, рассматриваемая в этой главе, относится к одному из самых впечатляющих применений Rust — вызову кода Rust из других языков для выполнения операций, либо критичных для безопасности, либо требующих больших вычислительных затрат. Такое положение дел свойственно также для языков C и C++ — многие среди выполнения тех или иных языков реализуют критичные для производительности функции на C или C++. Но Rust дает дополнительный бонус в виде его функциональных возможностей обеспечения безопасности. Это, по сути, и есть один из основных мотивирующих факторов внедрения Rust в практику применения

³ См. <https://docs.rs/rayon/latest/rayon/index.html>.

несколькими организациями — например, Mozilla, с ее планами повышения уровня безопасности и производительности браузера Firefox.

Примером возможностей использования Rust в этом качестве может послужить анализ или проверка данных из внешних источников — вроде веб-сервера, получающего ненадежные данные из Интернета. Многие уязвимости безопасности обнаруживаются при подпитке открытых интерфейсов случайными данными и наблюдении за происходящим, и зачастую в программном коде выявляются ошибки (например, чтение данных с выходом за пределы буфера), появление которых в Rust просто невозможно.

Большинство языков программирования и сред выполнения предоставляют некую форму привязок интерфейса внешних функций (FFI), продемонстрированную в главе 4. Но для многих популярных языков доступны привязки и инструменты более высокого уровня, способные существенно — по сравнению с применением FFI — упростить интеграцию Rust, а некоторые к тому же способны помочь с упаковкой собственных двоичных файлов. В табл. 11.1 приведены сводные данные по некоторым полезным инструментам для интеграции Rust с другими популярными языками программирования.

Таблица 11.1. Привязки Rust и инструменты для интеграции Rust с другими языками

Язык	Название	Описание	URL	Звезды GitHub*
Python	PyO3	Привязки Rust для Python, с инструментами для создания собственных пакетов Python с Rust-кодом	https://pyo3.rs	10,090
Python	Milksnake	Расширение setuptools для включения двоичных файлов в пакеты Python, включая двоичный Rust-код	https://github.com/getsentry/milksnake	783
Ruby	Ruru	Библиотека для создания собственных расширений Ruby с применением Rust-кода	https://github.com/d-unseable/ruru	822
Ruby	Rutie	Привязки между Ruby и Rust, позволяющие интегрировать Rust с Ruby или Ruby с Rust	https://github.com/danielpclarke/rutie	812
Elixir и Erlang	Rustler	Библиотека для создания безопасных привязок к Rust для Elixir и Erlang	https://github.com/rusterlium/rustler	3,999
JavaScript и TypeScript на Node.js	Neon	Привязки Rust для создания собственных модулей Node.js с применением Rust-кода	https://neon-bindings.com	7,622
Java	jni-rs	Собственные привязки Rust для Java	https://github.com/jni-rs/jni-rs	1,018
Rust	bindgen	Генерирует привязки Rust FFI из собственного Rust-кода	https://github.com/rust-lang/rust-bindgen	3,843

* Это количество звезд GitHub по состоянию на 30 декабря 2023 года.

11.6. Что делать дальше?

Поздравляю, вы дочитали книгу до конца. Давайте призадумаемся, о чем вам удалось узнать, прочитав книгу, и о более важном вопросе: что делать дальше, чтобы расширить свои познания.

В главах с 1 по 3 основное внимание уделялось инструментам, структуре проекта и базовым навыкам, необходимым для эффективной работы с Rust. В главах 4 и 5 рассматривались структуры данных и модели памяти Rust. Главы 6 и 7 были посвящены имеющимся в Rust функциям тестирования и способам извлечения из их применения наибольшей пользы. Главы 8, 9 и 10 познакомили вас с асинхронным Rust, а последняя глава рассказала о возможностях оптимизации кода Rust.

На следующем этапе освоения материалов книги вам, возможно, захочется перечитать предыдущие главы, особенно если их содержимое показалось вам слишком насыщенным или сложным для понимания. Зачастую бывает полезно дать мозгу немного отдохнуть, а затем, как только выдастся время на усвоение новой информации, вернуться к сложным темам. Для дальнейшего ознакомления с Rust я рекомендую прочитать мою следующую книгу «Rust Design Patterns», а также книгу Тима Макнамары «Rust в действии»⁴ (Tim McNamara, *Rust in Action*) — обе они выпущены издательством Manning Publications.

Rust и его экосистема устремлены в будущее и постоянно совершенствуются. Язык Rust, будучи уже достаточно зрелым, активно развивается и неуклонно движется вперед. Поэтому в завершение последней главы книги я оставлю вам ссылки на несколько информационных ресурсов, позволяющих наметить дальнейшее движение Rust и узнать о его новых функциях, изменениях и предложениях, а также о том, как можно активнее принимать участие в работе Rust-сообщества.

Основной материал по разработке Rust и его инструментария размещен на GitHub в рамках проекта rust-lang⁵. Помимо этого, отличными источниками для более глубокого погружения в Rust могут послужить следующие ресурсы:

- заметки о выпуске для каждой версии Rust — <https://github.com/rust-lang/rust/blob/master/RELEASES.md>;
- запросы на комментарии по языку Rust (RFC), предлагаемые для Rust функции и текущее состояние этих функций — <https://rust-lang.github.io/rfcs>;
- официальный справочник по языку Rust — <https://doc.rust-lang.org/reference>;
- официальный форум пользователей языка Rust, где можно обсудить Rust со своими единомышленниками, — <https://users.rust-lang.org>.

⁴ См. <https://bhv.ru/product/rust-v-dejstvii/>.

⁵ См. <https://github.com/rust-lang>.

Резюме

- Абстракции Rust с нулевой стоимостью позволяют создавать быстрый код, не беспокоясь об издержках, но, чтобы воспользоваться соответствующими оптимизациями, код должен быть скомпилирован в режиме релиза.
- Если требуемый объем памяти известен заранее, то под векторы, основную структуру данных для Rust-последовательностей, осуществляется предварительное выделение памяти нужного объема.
- При копировании данных между структурами быстрый путь для перемещения данных между слайсами обеспечивается применением метода `copy_from_slice()`.
- Экспериментальная переносимая SIMD-функция Rust облегчает создание кода с использованием SIMD-технологии. При этом исключается непосредственная работа с ассемблерным кодом или со встроенными функциями компилятора, а также не устраняется беспокойство о доступности тех или иных инструкций.
- Применение крейта `Rayon`, построенного на основе паттерна итераторов Rust, позволяет без особых трудностей распараллелить программный код. Создавать распараллеленный код в Rust не сложнее использования итераторов.
- Код на языке Rust можно внедрить в программы на других языках, заменив отдельные компоненты на их Rust-эквиваленты, предлагая при этом присущие Rust преимущества в сфере безопасности и производительности без необходимости полного переписывания имеющихся приложений.

Приложение

В этом приложении содержатся инструкции по установке утилит командной строки, необходимых для компиляции и запуска приводимых в книге примеров кода. Инструкции даются для удобства, но строго следовать содержащимся в них процедурам при наличии других необходимых инструментов или предпочтаемых иных способов установки не обязательно.

Установка инструментов для примеров, приводимых в книге

Прежде чем приступать к компиляции и запуску кода представленных в книге примеров, необходимо установить все необходимые зависимости.

Установка инструментов на macOS с помощью Homebrew

```
$ brew install git
```

На системах под управлением macOS нужно установить инструменты командной строки Xcode:

```
$ sudo xcode-select -install
```

Установка инструментов на системах под управлением Linux

На системах, основанных на Debian:

```
$ apt install git build-essential
```

На системах, основанных на Red Hat:

```
$ yum install git make automake gcc gcc-c++
```

СОВЕТ

Возможно, вместо GCC вам захочется установить clang, поскольку в его среде компиляция идет быстрее.

Установка rustup на Linux- или UNIX-системах

Чтобы установить rustup в среде операционных систем на базе Linux или UNIX, включая macOS, нужно запустить на выполнение следующую команду:

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

После установки `rustup` рекомендуется убедиться, что установлены как стабильная, так и ночная версия набора инструментов:

```
$ rustup toolchain install stable nightly  
...
```

Установка инструментов на системах под управлением Windows

При использовании операционной системы семейства Windows нужно загрузить последнюю версию `rustup`¹. Можно также загрузить готовые двоичные файлы Windows для `clang`².

В качестве альтернативы в Windows можно воспользоваться Windows Subsystem для Linux (WSL)³ и следовать приведенным ранее инструкциям по установке в системы под управлением Linux. Для многих такой способ работы с примерами кода может оказаться самым простым.

Управление `rustc` и другими Rust-компонентами с помощью `rustup`

После установки `rustup` нужно установить компилятор Rust и сопутствующие ему инструменты. Как минимум рекомендуется установить как стабильный, так и ночной канал Rust.

Установка `rustc` и других компонентов

Изначально рекомендуется устанавливать как стабильные, так иочные наборы инструментов, но, в общем-то, по возможности лучше работать со стабильными версиями. Чтобы установить оба набора инструментов, следует запустить на выполнение следующие команды:

```
$ rustup default stable      ← установка стабильной версии Rust и объявление ее исходным инструментарием  
...  
$ rustup toolchain install nightly  ← установка ночной версии Rust
```

Кроме этих наборов, в примерах книги используются инструменты `Clippy` и `rustfmt`. Они устанавливаются с помощью `rustup`:

```
$ rustup component add clippy rustfmt
```

¹ См. <https://rustup.rs/>.

² См. <https://releases.llvm.org/download.html>.

³ См. <https://docs.microsoft.com/en-us/windows/wsl/>.

Переключение исходных наборов инструментов с помощью *rustup*

При работе с Rust можно довольно часто сталкиваться с переключением между стабильным и ночным инструментарием. Сделать это с помощью *rustup* совсем несложно:

```
$ rustup default stable           ← переключение на исходное использование стабильной версии инструментов  
...  
$ rustup default nightly          ← переключение на исходное использование ночной версии инструментов
```

Обновление Rust-компонентов

rustup существенно облегчает поддержку компонентов в актуальном состоянии. Чтобы обновить все установленные наборы инструментов и компоненты, надо просто запустить на выполнение следующую команду:

```
$ rustup update
```

В обычных условиях запускать обновление нужно только при выходе крупных новых релизов. Иногда в *nightly*-версии могут возникать какие-то проблемы, требующие обновления, но такое, как правило, случается довольно редко. Слишком часто (например, ежедневно) обновлять работоспособную установку не рекомендуется, поскольку это с высокой долей вероятности может стать причиной возникновения ненужных проблем.

ПРИМЕЧАНИЕ

Обновление всех компонентов Rust приводит к загрузке и обновлению всех наборов инструментов и компонентов, что на системах с ограниченной пропускной способностью может занять немало времени.

Установка HTTPie

HTTPie — инструмент командной строки для выполнения HTTP-запросов, входящий в состав многих популярных диспетчеров пакетов, в числе которых Homebrew, apt, yum, choco, Nixpkgs и другие. Если в вашем диспетчере пакетов HTTPie недоступен, для установки этого инструмента можно обратиться к использованию имеющейся в Python системе управления пакетами pip:

```
# Install httpie  
$ python -m pip install httpie
```


Предметный указатель

C

CRUD-команды 281

F

FFI-интерфейс 56

Fuzz-тестирование 96, 211

R

Rust-компилятор rustc 37, 48

W

WebAssembly (Wasm) 26, 27, 32, 33

A

Абстракции с нулевой стоимостью 297

Асинхронные среды выполнения 221, 224, 230, 247, 248

Асинхронный ввод/вывод 222

Атомарные счетчики 152

Б

Блокировка основного потока 226

Блокирующее бездействие 233

В

Варианты использования Rust 33

Векторы 111, 299

Виртуальные манифести 71

Владение 140

Внутренняя изменчивость 152

Г

Генератор шаблонов Cargo 39

Глобальный распределитель 159

Глубокое копирование 142, 170

Д

Дескриптор среды выполнения 230, 243

Динамическое выделение памяти 74

З

Заимствование 140–142, 151, 152

Закрепленный указатель 237

И

Инструмент

◊ Clippy 54

◊ rustup 42

Интеграционное тестирование 200, 203

Интерфейс

◊ внешней функции 56, 133

◊ командной строки 79

К

Каналы 42, 43

Категоричное форматирование 83

Клонирование 143, 154–156, 170, 210

◊ при записи 155, 170

Конкурентность 219–221, 230, 240, 243, 249
 Кортежи 118, 121
 Крейты 41, 43, 49, 71, 75
 Куча 138

Л

Линты 78, 86
 Лицензия MIT 26, 33

М

Магические завершения 80
 Массивы 108, 111, 117
 Метод нечеткого анализа 212
 Миграции 256, 257, 261, 263
 Модульное тестирование 175, 178, 198, 200, 206
 Монады 146
 Мытекс 190, 193

Н

Нечеткое тестирование 96, 211
 Ночной канал 92

О

Открытые интерфейсы 200
 Охват кода 196, 197
 Ошибка нехватки памяти 147

П

Паника 176, 177
 Параллелизм 217, 219–221, 224, 230–232, 249
 Парсинг 109
 Переименование 192
 Перемещение 140, 142, 194
 Переписывание 194
 Переформатирование 192
 Перечисления 117, 125–129, 136
 Пользовательское восприятие (User Experience, UX) 203
 Примитивные типы 117
 Примитивы 117
 Проект LLVM 297
 Проход 144
 Псевдонимы 117, 128
 Пул базы данных 260

Р

Рабочие пространства 51, 75
 Размерные типы 119
 Разновидности требований 44
 Разработка через тестирование 203
 Распараллеливание 307–309
 Регрессии 191, 198
 Режим
 ◊ отладки 299
 ◊ релиза 299, 303, 304, 312
 Реструктуризация 191
 Рефакторинг 191
 Рефлексия 298

С

Сборка мусора 28
 Сервис GitHub Actions 52, 53, 56
 Синхронный ввод/вывод 222
 Системы непрерывной интеграции и непрерывного развертывания 52
 Слайсы 106, 108–111, 118, 121
 Сниппеты 80
 Состояние гонки 184
 Средство
 ◊ Rosetta 59
 ◊ rustdoc 48, 62–66, 75
 Стек 138, 139
 Строковые типы 104
 Структуры 101–104, 114, 117, 122–125, 127, 133, 134
 Сценарии 71

Т

Тестирование
 ◊ Fuzz-тестирование 96, 211
 ◊ асинхронного кода 247
 ◊ интеграционное тестирование 200, 203
 ◊ модульное тестирование 175, 178, 198, 200, 206
 ◊ нечеткое тестирование 96, 211
 ◊ разработка через тестирование 203
 ◊ свойств 180, 198, 211
 ◊ юнит-тестирование 175
 Технология SIMD 305
 Типаж 106, 112, 113, 115–117, 124, 131–133, 136, 177
 Трассировка 244, 245, 249, 252, 255, 258, 259, 268, 278

У

Умные указатели с подсчетом ссылок 151,
156, 171

Ф

Фаззинг 199, 211
Фича-флаги 46, 47
Флаг особенности 93
Функция защищенной памяти 94
Фьючерсы 224, 227–230, 235, 236, 238, 241

Х

Хеш-функции 115

Ц

Целочисленные типы 118

Ш

Шаблон SemVer 44, 45

Э

Экстрактор 269

Ю

Юнит-структуры 123
Юнит-тестирование 175

Бренден Мэтьюз
Rust.
Профессиональное программирование

Перевод с английского Н. Вильчинского

ТОО "АЛИСТ"
010000, Республика Казахстан,
г. Астана, пр. Сарыарка, д. 17, ВП 30

Подписано в печать 04.03.25.
Формат 70×100^{1/16}. Печать офсетная. Усл. печ. л. 25,8.
Тираж 1200 экз. Заказ № 12580.

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, РФ, М.О., г. Чехов, ул. Полиграфистов, д. 1

Rust

Профессиональное программирование

Язык Rust успел прославиться своей высокой производительностью, надежностью и безопасностью. Но не так просто овладеть им в совершенстве, чтобы все эти достоинства раскрылись в полной мере. Эта книга поможет вам быстро стартовать в изучении сложных тем, уверенно приступить к работе с низкоуровневыми системами, веб-приложениями, заниматься асинхронным программированием, оптимизацией, писать конкурентный код. С этой книгой вы сможете работать более продуктивно. Она насыщена примерами и, опираясь на уже имеющиеся у вас знания, познакомит вас со специфичными для Rust паттернами проектирования, приемами асинхронного программирования, а также поможет интегрировать Rust с другими языками. Также в ней рассказано о замечательных инструментах для тестирования, анализа кода, управления жизненным циклом приложения на Rust. Все самое нужное — под одной обложкой!

В этой книге

- Структуры данных Rust
- Управление памятью
- Создание эффективных API
- Инструментарий Rust, средства для тестирования и многое другое

Книга ориентирована на читателей, имеющих базовые представления о Rust.

Бренден Мэтьюз — программист, ИТ-предприниматель и активный контрибьютор свободного ПО, работающий с Rust почти с самого появления этого языка.

“Каждый Rust-разработчик что-то найдет для себя в этой книге. Просто кладезь советов.”

— Тим Макнамара,
основатель компании
Accelerant.dev, автор книги
«Rust в действии»

“Ta самая книга, которая поможет стать Rust-профессионалом.”

— Хайме Лопес,
Институт Гуттманна,
Барселона

“Практичная, удобная и понятная книга.”

— Сатедж Кумар Сахи,
компания «Боинг»

“Для всех амбициозных растгофилов.”

— Симон Чоке,
компания German
Edge Cloud



ISBN 978-601-08-4833-7