

Переход на Rust

Рефакторинг
исходного кода
с других языков

Лили Мара
Джоэл Холмс

MANNING

ДМК
издательство



Лили Мара, Джоэл Холмс

*Переход на Rust.
Рефакторинг исходного кода
с других языков*

Refactoring to Rust

Lily Mara
Joel Holmes



MANNING
Shelter Island

*Переход на Rust.
Рефакторинг исходного кода
с других языков*

Лили Мара
Джоэл Холмс



Москва, 2026

УДК 004.438Rust

ББК 32.973.2

M25

Лили Мара, Джоэл Холмс

M25 Переход на Rust. Рефакторинг исходного кода с других языков / пер. с англ. А. В. Логунова. – М.: ДМК Пресс, 2025. – 344 с.: ил.

ISBN 978-5-93700-228-0

Эта книга научит вас расширять функциональность и повышать производительность приложений за счет поэтапного рефакторинга кодовой базы на язык Rust. Вы узнаете, как использовать Rust для обертывания опасного исходного кода, вызывать стандартные и прикладные библиотеки языка Rust и даже использовать формат байт-кода Wasm для исполнения кода Rust в браузере, а также овладеете навыками создания защищенных приложений с ограниченным потреблением памяти.

Для программистов среднего уровня. Опыт работы с языком Rust не требуется.

УДК 004.438Rust

ББК 32.973.2

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

©DMKPress 2025. Authorized translation of the English edition. © 2025 Manning Publications. This translation is published and sold by permission of Manning Publications, the owner of all rights to publish and sell the same.

ISBN (анг.) 9781617299018
ISBN (рус.) 978-5-93700-228-0

© 2025 by Manning Publications Co.
All rights reserved
© Оформление, издание, перевод,
ДМК Пресс, 2025

*Посвящается всем тем, кто думал,
что у них ничего не получится.
— Лили Мара*

*Посвящается моей бабушке Джози и моей тете Алише,
которые привили мне любовь к чтению и технологиям.
— Джоэл Холмс*

Краткое оглавление

1	■ Зачем перерабатывать исходный код на язык Rust	19
2	■ Обзор языка Rust	36
3	■ Введение в интерфейс с внешними функциями C (C FFI) и незащищенный язык Rust	88
4	■ Продвинутый интерфейс с внешними функциями (FFI)	126
5	■ Структурирование библиотек языка Rust	178
6	■ Интеграция с динамическими языками	206
7	■ Тестирование интеграций с Rust	233
8	■ Асинхронный Python с языком Rust.....	261
9	■ WebAssembly для переработки исходного кода JavaScript	282
10	■ Интерфейс с WebAssembly для переработки исходного кода ...	304

Оглавление

Предисловие.....	10
Признательности.....	11
О книге	13
Кому стоит прочитать эту книгу.....	13
Как эта книга организована: дорожная карта.....	13
Об исходном коде.....	14
Дискуссионный форум liveBook	15
Об авторах	16
От издательства	17
Об иллюстрации на обложке	18
1 Зачем перерабатывать исходный код на язык Rust	19
1.1 Что такое переработка исходного кода?	20
1.2 Что из себя представляет язык Rust?	22
1.3 Почему именно язык Rust?	23
1.4 Следует ли перерабатывать исходный код на язык Rust?	23
1.4.1 Производительность	24
1.4.2 Защищенность памяти	26
1.4.3 Удобство технического сопровождения	27
1.5 Когда не следует перерабатывать исходный код на язык Rust	28
1.6 Как это работает?	29
1.7 Чему вы научитесь в этой книге?	30
1.7.1 Вызов функций Rust непосредственно из своей программы	30
1.7.2 Обмен со службой Rust по сети.....	33
1.8 Для кого предназначена эта книга?.....	34
1.9 Какие инструменты потребуются для начала работы?	34
Краткий итог	35
2 Обзор языка Rust	36
2.1 Владение и заимствование	37
2.2 Управление памятью в других языках	41
2.3 Времена жизни	45
2.3.1 Ссылки и заимствования	49
2.3.2 Контроль за муттируемостью.....	50
2.3.3 Ссылки и времена жизни.....	53
2.4 Строковые типы в языке Rust	56
2.4.1 Муттируемые строковые литералы	58
2.5 Перечисления и обработка ошибок.....	61
2.5.1 Перечисления.....	62
2.5.2 Обработка ошибок с помощью перечислений.....	66
2.5.3 Единичный тип.....	69

2.5.4 Типы ошибки	71
2.5.5 Преобразование ошибок	75
2.5.6 Поднятие паники при появлении ошибок	80
Краткий итог	86
3 Введение в интерфейс с внешними функциями C и незащищенный язык Rust.....	88
3.1 Незащищенный язык Rust	89
3.1.1 Обыкновенные указатели	90
3.2 Интерфейс с внешними функциями С	92
3.2.1 Включение пакета Rust.....	98
3.2.2 Создание динамической библиотеки с использованием языка Rust	100
3.2.3 Решение арифметических выражений на языке Rust	108
3.2.4 Общая черта Display.....	119
Краткий итог	125
4 Продвинутый интерфейс с внешними функциями.....	126
4.1 Скачивание исходного кода веб-сервера NGINX	127
4.2 Создание модуля для веб-сервера NGINX	128
4.3 Связывание языка С с языком Rust	132
4.3.1 Скрипты сборки	134
4.3.2 Инструмент bindgen	139
4.4 Чтение запроса веб-сервера NGINX	145
4.4.1 Аннотации времен жизни.....	152
4.4.2 Аннотации времен жизни в плагине для веб-сервера NGINX....	158
4.5 Использование библиотеки-калькулятора.....	162
4.6 Написание HTTP-ответа	167
Краткий итог	176
5 Структурирование библиотек языка Rust.....	178
5.1 Модули	178
5.1.1 Какая разница?	182
5.1.2 Несколько файлов.....	183
5.2 Пути.....	186
5.2.1 Относительные и абсолютные пути	187
5.2.2 Псевдонимы путей	197
5.3. Восходящая видимость.....	202
Краткий итог	205
6 Интеграция с динамическими языками.....	206
6.1 Обработка данных в Python.....	206
6.2 Планирование перемещения на язык Rust	207
6.3 Разбор данных JSON	208
6.4. Написание модуля расширения Python на языке Rust	214
6.5 Измерение и тестирование производительности в Rust	219
6.6 Оптимизированные сборки	229
Краткий итог	231
7 Тестирование интеграций с Rust.....	233
7.1 Написание тестов на языке Rust.....	234
7.1.1 Документарные тесты	240
7.1.2 Добавление тестов в существующий исходный код	245

7.2 Тестирование исходного кода Rust с использованием языка Python	250
7.2.1 Динамическое латание.....	254
Краткий итог	259
8 Асинхронный Python с языком Rust.....	261
8.1 Генерирование множества Мандельброта на языке Python	262
8.2 Масштабирование	265
8.3 Библиотека asyncio.....	268
8.4 Многопоточность исполнения	271
8.5 Глобальная блокировка	274
8.6 Библиотека PyO3.....	276
Краткий итог	280
9 WebAssembly для переработки исходного кода JavaScript	282
9.1 Что такое WebAssembly?.....	283
9.2 Перенос из JavaScript в Rust	285
9.3 Язык Rust в браузере	286
9.3.1 Запрос данных	286
9.3.2 Компиляция в Wasm	290
9.3.3 Загрузка Wasm в браузер	291
9.4. Создание компонента библиотеки React	292
9.5 Веб-компоненты полностью на языке Rust	296
9.6 Еще раз о переработке JavaScript.....	302
Краткий итог	303
10 Интерфейс с WebAssembly для переработки исходного кода...	304
10.1 Универсальная среда исполнения системного интерфейса WASI	308
10.2 Из браузера на машину	312
10.3 Библиотека Wasm.....	318
10.4 Потребление Wasm	320
10.5 Еще о Wasm	326
10.6 Память Wasm.....	329
10.7 Это только начало	335
Краткий итог	335
Предметный указатель.....	337

Предисловие

На протяжении всей нашей карьеры разработчиков программного обеспечения у нас была возможность участвовать в нескольких проектах по переработке кодовой базы. Их суть нередко состояла в одном и том же: нужно было масштабировать продукты и при этом уложиться в ограниченное время. Такая ситуация приводит к тому, что разработка занимает месяцы, заполненные обсуждениями шаблонов и языков.

Переработка кодовой базы с использованием языков Java и Go со-пряжена со значительными трудностями, включая постоянное перемещение файлов, экспорт пакетов, создание системных оболочек и полное переписывание существующих систем. Пути к успеху редко были четко определены. Цель этой книги состоит в том, чтобы познакомить вас со многими из этих шаблонов, используя язык, предназначенный для разбивки и переписывания существующих систем. Книга «Переход на Rust. Рефакторинг исходного кода с других языков» демонстрирует легкость, с которой указанный язык может легко интегрироваться в вашу экосистему, обеспечивая преимущества масштабирования с первого дня благодаря особенностям языка.

Язык Rust привносит такие преимущества, как типобезопасность и защищенность памяти, а также связанный с этими свойствами прирост в производительности. Из этой книги вы узнаете, как язык Rust может улучшать практически любой проект. Язык Rust, который позиционируется как замена существующим языкам, таким как C и C++, отличается своей надежной цепочкой инструментов и функциональными компонентами защиты памяти. Мы также рассмотрим способы взаимодействия языка Rust с такими языками, как Python, и расскажем об улучшениях производительности при создании библиотек и модулей, работающих на обоих языках. Кроме того, мы откроем для вас неожиданные области применения языка Rust, например в веббраузерах и в качестве универсальной среды исполнения.

В целом цель этой книги – не только продемонстрировать мощь языка и экосистемы языка Rust, но и дать вам навыки уверенной переработки крупных систем.

Признательности

Благодарю своих партнеров за то, что они помогли довести эту книгу до конца, после того как я так долго откладывала ее в долгий ящик.

Спасибо моей маме, которая редактировала некоторые ранние наброски и чья любовь к чтению не давала мне отрываться от книг в течение многих лет.

Благодарю своих родителей, бабушек и дедушек за то, что они всегда поощряли меня в получении технического образования и делали это возможным.

Моему дяде Конраду спасибо за то, что помог развитить любопытство к тому, как все устроено.

Спасибо всем сотрудникам издательства Manning. Эта книга не появилась бы на свет без самоотверженного труда редакторов, технических редакторов, графических редакторов, маркетологов, корректоров и многих других. Благодарю Энди Уолдрона за предоставленную возможность написать мою первую книгу.

Особая благодарность редакторам этой книги, Элеше Хайд и Сьюзан Эттридж. Благодаря вашим советам я смогла воплотить свои фантазии в жизнь.

Компания OneSignal благодарит вас за то, что предоставили мне время и свободу для написания этой книги.

Благодарю учителей и профессоров, которые привили мне любовь к письменному слову, – Кристен Шумахер, Пола Хеберта и покойного Жана Лутца.

Спасибо тебе, Норм Крампе, за то, что удовлетворил мое техническое любопытство, и доктору Пэрис Франц за то, что вдохновила меня на такое важное дело, как написание книги.

Спасибо тебе, Ян Паскуаль, за то, что ты был подопытным кроликом в плане навыков технического общения, которые лежат в основе написания этой книги, и за твою поддержку на протяжении всего процесса.

Написание книги – это титанический труд, и я также благодарю всех, кто был одним из первых рецензентов или клиентов MEAP, оставивших отзывы о книге на онлайновых форумах.

– ЛИЛИ МАРА

Прежде всего хотел бы поблагодарить мою жену и партнера Челси, которая вдохновляет меня на осуществление моих мечтаний о писательстве и обучении.

Также благодарю двух моих сыновей, Эли и Абеля, которые являются неиссякаемым источником вдохновения. Посвящаю эту книгу двум

важным женщинам в моей жизни. Во-первых, это моя бабушка, которая привила мне любовь к книгам и чтению, и, во-вторых, моя тетя Алиша, которая в раннем возрасте повлияла на мою любовь к компьютерам.

Эта книга не была бы написана без огромной поддержки коллектива издательства Manning. Спасибо тебе, Энди Уолдрон, за предоставленную возможность написать книгу на столь интересную тему.

Как автор многих книг издательства Manning, я лично особенно ценю всех тех, кто оставил отзывы в рецензиях на книгу: Алена Кунью, Альфреда Томпсона, Амита Ламбу, Ариэля Отилибили, Крис. Карделла, Кристофера Вильянуэву, Клиффорда Тербера, Дэна Шейха, Даниэля Томаса Лареса, Диего Алонсо, Федерико Кирхейса, Фостера Хейнса, Габора Ласло Хайба, Жиля Якелини, Хаварда Уолл, Джареда Лав, Джеймса Блэкли, Джона Касевич, Джона Риддл, Джонатана Ривз, Жюльена Кастелейн, Кента Р. Спиллнер, Кшиштофа Камичек, Мацея Пшепира, Маркуса Гезелле, Мэттью Сармьенто, Макса Садрие, Михала Рутка, Мохсена Мостафа Йокар, Рамона Снир, Рани Шарим, Ричарда Рэндалл, Сальвадора Наваррете Гарсия, Сэма Ван Овермейр, Самбасиву Андалури, Сына джин Ким, Сейи Огуньеми, Тима Макнамара, Троя Эйслер, Уолта Стоунбернера, Уильяма Э. Уилер и Еркебулана Тулибергенова. И я принателен тем, кто приобрел эту книгу заранее через MEAP и предоставил отзывы и поддержку.

Очень благодарен Элеше Хайд за всю ту помощь, руководство и терпение, которые она проявила. На ее долю выпала трудная задача – с огромным мастерством руководить работой над этой книгой. Я очень ценю ее терпение и поддержку.

Также выражаю благодарность компании Regrow.ag, предоставившей мне свободу и вдохновившей на написание этой книги; моему другу Коди, который был рядом со мной с начальной школы; моим учителям английского языка в старших классах, которые поощряли мою писательскую деятельность и помогали мне обрести голос; и Отто, который всегда был рядом, чтобы выслушать и никогда не осуждать.

– ДЖОЭЛ ХОЛМС

О книге

В известной книге Мартина Фаулера (Martin Fowler) «Переработка исходного кода» (Refactoring) подчеркивается первичная цель переработки: улучшить конструктивное исполнение существующего исходного кода. Знакомые с указанной книгой читатели по достоинству оценили его метод представления различных сегментов исходного кода, за которым следуют улучшенные альтернативы, повышающие удобочитаемость, эффективность или простоту. Несмотря на то что во втором издании стратегии претерпели изменения, стержневая идея остается неизменной: функциональный исходный код всегда можно улучшить.

Книга «Переход на Rust. Рефакторинг исходного кода с других языков» описывает стратегии транзита с одного языка программирования на другой с сохранением внешнего поведения исходного кода. Как это достигается? Как вы увидите, Rust предназначен для постепенной замены других языков путем интеграции и декомпозиции существующего исходного кода – во многом напоминающих процесс ржавления железа – и замены его исходным кодом Rust. Изначально проект был нацелен на замену языка C++, но со временем расширился, включив в него языки JavaScript и Python.

Кому стоит прочитать эту книгу

Эта книга предназначена для разработчиков, которые специализируются на других языках, таких как C, C++, Python и JavaScript, но хотят изучить язык Rust. Хотя наша книга не дает подробного представления о данном языке, в ней приведены практические примеры и варианты использования для замены вашего исходного кода на язык Rust. Формального понимания языка Rust не требуется, хотя оно и полезно.

Как эта книга организована: дорожная карта

В соответствии с подходом Фаулера мы будем представлять задачи на одном языке и демонстрировать способы переработки этих сложностей на языке Rust. Цель состоит в том, чтобы поддерживать положенную в основу функциональность приложения, используя скорость и защищенность языка Rust с целью улучшения системы в целом.

Изложение начинается с ознакомления с языком Rust, обсуждения его механики и сравнения с такими языками, как C, C++ и Python. Эта информация представлена в контексте переработки исходного кода, делая упор на методах систематического улучшения системы, не позволяя им превращаться в неуправляемый исходный код. Кроме того, будут рассмотрены продвинутые функциональные компоненты языка Rust, такие как время жизни переменных и владение, которые имеют решающее значение для овладения языком.

Первостепенное внимание будет уделено языку C, который для многих является основополагающим. В главе 3 будет рассмотрена способность языка Rust создавать как защищенный, так и незащищенный исходный код, обследована тема обертывания опасного исходного кода на языке Rust и использование инструментов отладки. Эта основа подготовит вас к главе 4, где вы будете интегрировать Rust в существующую кодовую базу на языке C, управлять памятью и добавлять новую функциональность в веб-сервер NGINX.

После первоначальной интеграции в другую систему в главе 5 язык Rust будет рассмотрен как библиотечный инструмент. Создание пакетов, совместимых с другими проектами, является эффективным способом переработки приложений, при условии что эти библиотеки предлагают усиленную функциональность. Кроме того, будут обследованы метрики измерения и тестирования производительности, чтобы обосновать переход с более старых языков на язык Rust. В главах 6, 7 и 8 будут продемонстрированы способы применения этих пакетов для переработки исходного кода Python путем исполнения Python в экосистеме Rust либо путем встраивания экосистемы Rust в Python.

В последних двух главах вы познакомитесь с продвинутыми приложениями на языке Rust. Глава 9 посвящена компиляции Rust для работы в веб-браузерах с использованием нового двоичного формата под названием Wasm. В главе 10 эта технология будет использована для создания универсальной среды исполнения, предоставляющей гибкий (но сложный) метод переработки или взаимодействия с существующим исходным кодом.

Главы не обязательно читать по порядку, и если вы уже знакомы с языком Rust, то, вероятно, сможете пропустить первые две главы, при условии что не захотите освежить материал в памяти. Если вам не терпится перейти к определенному языку, то главы 3 и 4 посвящены интеграции с языками C и C++, главы 6–8 – интеграции с языком Python, а глава 9 – интеграции с языком JavaScript.

Главу 10 также можно прочитать отдельно, и в ней предлагается другой способ переработки, заключающийся не в изменении самого исходного кода, а в изменении среды, в которой исполняется приложение.

Переработка исходного кода – это скорее искусство, чем наука. И в нашей книге, и в книге Мартина Фаулера предлагаются шаблоны, которым предлагается следовать; ответственность за эффективное применение этих методов полностью лежит на вас.

Об исходном коде

Описанный в этой книге исходный код в основном посвящен языку Rust, но в контексте других языков. В начале рассматриваются основы языка Rust, а затем в оставшейся части книги описывается интеграция с языками C, Python и JavaScript. Эти языки не преподаются, но предполагается, что читатель их знает, если он занимается переработкой исходного кода на этом языке.

Какие-либо ограничения на используемое оборудование или программное обеспечение отсутствуют. Содержимое книги не относится к конкретной операционной системе и не нуждается в какой-либо спе-

циальной настройке, кроме установки языка Rust. В главах упоминаются дополнительные библиотеки и инструменты, но текст посвящен данной настройке, и читателю не требуется делать это заранее.

В дополнение к этому Rust – это развивающийся язык, и, следовательно, синтаксис и библиотеки могут со временем меняться. Для того чтобы максимально это учесть, в приводимых примерах мы позаботились о выборе стабильных библиотек.

Данная книга содержит много примеров исходного кода, как в про- нумерованных листингах, так и в виде обычного текста. В обоих слу- чаях исходный код отформатирован **вот таким шрифтом фиксированной ширины**, чтобы отделять его от обычного текста.

Во многих случаях изначальный исходный код был переформати- рован; мы добавили разрывы строк и переработали отступы, чтобы уложиться в доступное пространство на странице книги. В некоторых случаях даже этого было недостаточно, и листинги содержали марке- ры продолжения строки (**→**). Кроме того, нередко комментарии из листингов удалялись, когда исходный код описывался в тексте. Мно- гие листинги сопровождаются аннотациями к исходному коду, в кото- рых подчеркиваются важные идеи.

Исполняемые фрагменты исходного кода можно получить из liveBook-версии (онлайновой версии) этой книги по адресу <https://livebook.manning.com/book/refactoring-to-rust>. Полный исходный код примеров из книги доступен для скачивания с веб-сайта Manning по адресу <https://www.manning.com/books/refactoring-to-rust> и из репозитория книги на GitHub по адресу <https://github.com/lily-mara/refactoring-to-rust>.

Дискуссионный форум liveBook

Покупка данной книги включает в себя бесплатный доступ к онлай- новой платформе чтения книг издательства Manning под названи- ем liveBook. Используя эксклюзивные возможности обсуждения на liveBook, можно прикреплять комментарии к книге глобально либо к определенным разделам или абзацам. Там можно легко делать замет- ки для себя, задавать технические вопросы и отвечать на них, а также получать помощь от автора и других пользователей. В целях полу- чения доступа к форуму перейдите по ссылке <https://livebook.manning.com/book/duckdb-in-action/discussion>. Подробнее о форумах изда- тельства Manning и правилах поведения также можете узнать по адре- су <https://livebook.manning.com/discussion>.

Издательство Manning видит свою обязанность перед читателями в том, чтобы предоставлять место, где может происходить содер- жательный диалог между отдельными читателями и между читателями и автором. Это обязательство не требует от автора какого-то кон- кретного объема участия, чей вклад в форум остается добровольным (и неоплачиваемым). Мы предлагаем вам попробовать задать автору несколько сложных вопросов, чтобы поддержать его интерес! Форум и архивы предыдущих обсуждений будут доступны на веб-сайте изда- теля на протяжении всего времени, пока книга находится в печати.

Об авторах



ЛИЛИ МАРА (LILY MARA) – разработчик программного обеспечения из Сан-Франциско, штат Калифорния. Много выступает с докладами о разработке программного обеспечения на языке Rust как внутри страны, так и за рубежом. Пишет на Rust с 2015 года и профессионально использует его для разработки высокопроизводительных масштабируемых систем. В настоящее время занимается написанием программного обеспечения в Discord.



ДЖОЭЛ ХОЛМС (JOEL HOLMES) – разработчик программного обеспечения, специализирующийся на разработке облачно-ориентированных приложений. Работал в нескольких стартапах, занимаясь конструированием и разработкой новых продуктов и служб, чтобы помогать компаниям-заказчикам развиваться и расти дальше. Попутно способствовал формированию инструментов и процессов, которые помогали в разработке и повышении качества. Живет в Питтсбурге со своей семьей и в настоящее время работает над созданием облачных приложений в Regrow.ag.

От издательства

Отзывы и пожелания

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

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

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

Список опечаток

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

Нарушение авторских прав

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

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

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

Об иллюстрации на обложке

На обложке книги «Переход на Rust. Рефакторинг исходного кода с других языков» изображена фигура под названием «Пьемонтуаз д'Асти», или «Женщина из города Асти в Пьемонте», взятая из сборника Жака Грассе де Сен-Совера, опубликованного в 1788 году. Эта иллюстрация тонко прорисована и раскрашена вручную.

В те времена по одежде можно было легко определять места проживания людей, их профессии или положение в обществе. В издательстве Manning превозносятся изобретательность и инициатива вычислительного бизнеса с помощью книжных обложек, в основу которых кладется богатое разнообразие региональной культуры столетней давности, ожившее благодаря фотографиям из коллекций, подобных этой.

1

Зачем перерабатывать исходный код на язык Rust

Эта глава охватывает следующие ниже темы:

- зачем может потребоваться переработка приложения;
- почему язык Rust хорошо подходит для переработки;
- когда стоит и не стоит начинать проект по переработке;
- общий обзор методов, которые используются для переработки исходного кода на язык Rust.

Если вы когда-либо слышали о языке программирования Rust, то, возможно, слышали и о компаниях-разработчиках программного обеспечения, которые переписывают свой исходный код с более медленного и интерпретируемого языка на язык Rust. Некоторые из этих компаний то и дело публикуют посты, в которых восхваляются преимущества системы Rust по сравнению с их предыдущими системами, и в них рассказывается очень интересная история: другие языки работают медленно, а Rust просто летает, и поэтому они рекомендуют переписывать исходный код на язык Rust, и системы станут быстрыми.

Хотя, возможно, покажется заманчивым думать, что все мы легко сможем переписать свой исходный код, когда появится что-то более совершенное, мы осознаем, что на самом деле программное обеспечение не существует в пузыре из бесконечных ресурсов. Повышение производительности и выплата технической задолженности должны балансируться разработкой функциональных компонентов, запро- сами пользователями и миллионом других факторов, связанных с работой современного программного обеспечения. При повторной реализации функциональности на новом языке также необходимо обеспечивать, чтобы пользователи получали единообразную и надежную службу. Как же тогда разработчик сможет улучшить свою кодовую базу, сохраняя при этом ожидаемые высокие темпы разработки и надежность? Ответ заключается не в переписывании в стиле «большого взрыва», а в поступательной переработке исходного кода.

1.1 Что такое переработка исходного кода?

Рефакторинг (переработка/переделка исходного кода) – это процесс его реструктуризации таким образом, чтобы он исполнялся лучше, стал проще в плане технического сопровождения или соответствовал какому-либо другому определению «лучшего». Между переработкой и переписыванием существует различие, хотя и нечеткое. Разница между ними сводится к масштабу операции.

Переписывание – это повторная реализация с нуля целого приложения или значительной его части. Можно переписать систему, чтобы использовать преимущества нового языка программирования или модели хранения данных, или просто потому, что текущую систему сложно сопровождать в техническом плане и кажется, что проще будет ее выбросить и начать все сначала, чем улучшать.

Переработка – это переписывание в гораздо меньших масштабах. Вместо того чтобы стремиться к полной замене существующей системы, желательно найти те части системы, которые нуждаются в наибольшей помощи, и заменить как можно меньше исходного кода, чтобы улучшить систему. Преимущества переработки перед переписыванием многочисленны:

- Поскольку текущая система является «новой», она может продолжать работать и обслуживать клиентов, пока идет переработка. Можно разворачивать ряд очень небольших изменений в коде и смотреть, какое изменение вызвало проблему. Если переписывать и разворачивать всю новую систему одним махом, то как узнать, какая часть системы вызывает ошибки при их возникновении?
- Существующий исходный код, вероятно, уже имеет многолетний опыт разработки и мониторинга. Не следует недооценивать опыт других разработчиков в эксплуатации и отладке существующего кода. Если в новой системе возникает проблема, с которой у вас нет опыта работы, то как вы собираетесь ее устранять?

- В идеале существующий исходный код должен быть связан с автоматизированным тестированием. Тесты можно реиспользовать, чтобы удостоверяться в том, что переработанный исходный код соответствует тем же требованиям, что и существующий. Если в существующем исходном коде нет автоматизированных тестов, то переработка – отличный стимул, чтобы начать их писать!

На рис. 1.1 показано, как с течением времени развертывание будет отличаться при переписывании и переработке.

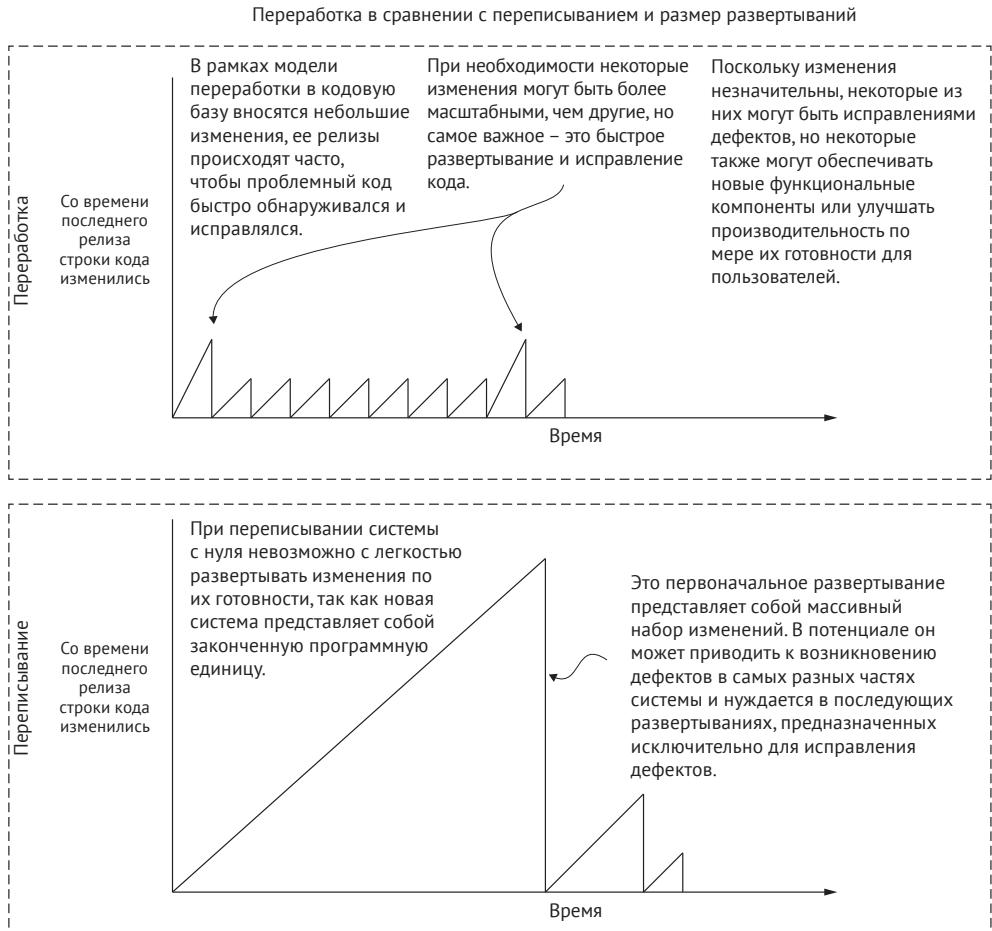


Рисунок 1.1 Как переработка и переписывание исходного кода влияют на размер развертываний

При переписывании системы изменения часто приходится объединять и развертывать одновременно. За счет этого снижается скорость и увеличивается риск возникновения ошибок в развертываниях. Чем дальше функциональные компоненты сидят в ветке или в застоявшейся промежуточной среде, тем сложнее будет отладить этот исходный код

при его развертывании. Если все программное обеспечение подвержено риску возникновения ошибок, то увеличение частоты внесения изменений и уменьшение количества изменяемых при развертывании строк кода поможет находить и устранять ошибки в кратчайшие сроки.

При переработке исходного кода требуется вносить небольшие независимые изменения, которые можно будет развертывать как можно скорее. К этим изменениям добавляются метрики и мониторинг, чтобы обеспечивать неизменность результатов после их развертывания. Этот процесс позволяет быстро и единообразно развертывать небольшие изменения, которые устраниют дефекты, добавляют функциональные компоненты или улучшают производительность системы.

При этом при переработке исходного кода, который уже успешно выполняет свою работу, следует учитывать ряд факторов:

- обеспечение единообразного поведения старого и нового исходного кода:
 - использование существующего автоматизированного тестирования;
 - написание новых тестов, которые работают с новыми структурами данных, введенными в результате переработки;
- развертывание нового исходного кода:
 - определение уровня обособленности между средами развертывания старого и нового исходного кода;
 - принятие решения о том, как сравнивать производительность обеих систем во время их работы;
 - управление развертыванием новой системы таким образом, чтобы только небольшой процент клиентов имел доступ к новым путям доступа к исходному коду.

В этой книге будут рассмотрены методы и подходы, которые используются для переработки медленного или сложного для понимания исходного кода на язык Rust. Мы расскажем о том, как находить наиболее важные части исходного кода, которые нуждаются в переработке, как адаптировать существующий исходный код к Rust, как тестировать только что переработанный исходный код и многое другое.

1.2 Что из себя представляет язык Rust?

Язык программирования Rust отличается быстрым временем исполнения, высокой надежностью и защищенностью памяти. Согласно rust-lang.org, Rust – это «язык, позволяющий разрабатывать надежное и эффективное программное обеспечение». Что это значит?

- *Расшифрене способностей* – Rust стремится наделять разработчиков способностями, которых у них не было бы при других обстоятельствах.
- *Радушне* – сообщество Rust чрезвычайно доброжелательно относится ко всем, независимо от их опыта. Разработчики на языке

Rust имеют все уровни квалификации; для некоторых Rust является первым языком программирования, а другие уже знают несколько. Некоторые из них занимаются низкоуровневым программированием, тогда как другие являются разработчиками приложений на таких языках, как Python, Ruby и JavaScript.

- **Надежность** – программное обеспечение Rust ориентировано на отказоустойчивость и четкую детализацию обработки ошибок, чтобы ничего не просачивалось наружу.
- **Эффективность** – благодаря прямой компиляции в машинный код и отсутствию сборщика мусора во время исполнения код на языке Rust изначально работает намного быстрее, чем код, написанный на интерпретируемых языках, таких как Python, Ruby и JavaScript. В дополнение к этому язык Rust предоставляет разработчикам инструменты управления деталями более низкого уровня, такими как резервирования памяти, когда это необходимо, что может приводить к кардинальному ускорению быстродействия, поддерживая при этом простоту понимания приложения.

1.3 Почему именно язык *Rust*?

Язык Rust сочетает в себе защищенность памяти, производительность и фантастическую систему типов; эти средства языка работают сообща, обеспечивая правильное исполнение приложений. Строгая система типов гарантирует, что обмен данными осуществляется в соответствии с правильным контрактом, а неожиданные данные не будут приводить к неожиданным результатам. Системы времен жизни и владения позволяют использовать память коллективно и напрямую через границы межъязыкового интерфейса с внешними функциями (Foreign Function Interface, аббр. FFI), не задавая вопросов о том, кто несет ответственность за высвобождение памяти. Надежные гарантии потокобезопасности позволяют добавлять параллелизм, который ранее был бы невозможен или сопряжен с большим риском. Совместив эти функциональные компоненты, по изначальному замыслу разработанные для того, чтобы помочь разработчикам писать более качественные программы на языке Rust, вы увидите, что они идеально подходят для поступательной переработки практически с любого языка на язык Rust.

1.4 Следует ли перерабатывать исходный код на язык *Rust*?

Потребность в переработке некоторых частей приложения на язык Rust возникает по самым разным причинам, но в этой книге будут рассмотрены две первичные цели – производительность и защищенность памяти.

1.4.1 Производительность

Представьте себе, что вы работаете над приложением, написанным на таком языке, как Python, Node.js или Ruby. Вы уже какое-то время добавляете новые функциональные компоненты в свое приложение, и у вас большая кодовая база. Однако по мере роста числа пользователей вы начали замечать, что платите слишком много за масштабирование своей службы требуемыми вычислительными ресурсами. Ваше приложение замедляется из-за какой-то части обработки запросов, но вы пока не совсем уверены, какой именно.

Эта книга проведет вас по таким методам, как измерение и тестирование производительности и профилирование, которые помогут определять те части кода, которые больше всего выиграют от переработки, ориентированной на производительность. После отыскания этих мест будут рассмотрены методы реализации той же функциональности на языке Rust, а также настройка производительности, которая может сделать исходный код максимально быстрым.

Давайте взглянем на небольшой пример. Представьте себе, что приведенный в следующем ниже листинге исходный код разбора строкового литерала в формате CSV находится в вашем веб-приложении.

Листинг 1.1 Функция на языке Python: сумма значений из столбца в строковом литерале в формате CSV

```
def sum_csv_column(data, column):
    sum = 0

    for line in data.split("\n"):
        if len(line) == 0:
            continue

        value_str = line.split(",")[column]
        sum += int(value_str)

    return sum
```

Приведенная выше функция на языке Python довольно тривиальна; она возвращает сумму всех значений из заданного столбца в строковом литерале в формате CSV. Написание той же функции на языке Rust выглядит очень похоже.

Листинг 1.2 Та же функция суммирования значений столбца CSV, написанная на Rust

```
fn sum_csv_column(data: &str, column: usize) -> i64 {
    let mut sum = 0;

    for line in data.lines() {
        if line.len() == 0 {
            continue;
```

Ключевое слово `mut` указывает на то, что переменная является мутируемой и ее значение со временем может меняться.

Функции в языке Rust всегда имеют явно обозначенные типы параметров и возвращаемых значений.

```

    }

    let value_str = line
        .split(",")
        .nth(column)
        .unwrap();
    sum += value_str.parse::<i64>().unwrap();

}

sum
}

```

Метод unwrap в конце этих строк
указывает на то, что используемые функции могут отказывать, и будет поднята паника, если это произойдет.

Этот синтаксис (::<i64>) в языке Rust называется оператором «турборыба» (turbofish); он используется, когда компилятору нужна подсказка о том, какой тип должна возвращать функция. Поскольку в зависимости от контекста функция разбора parse может возвращать разные типы, он необходим для устранения неоднозначности (более подробную информацию смотрите в главе 3).

Версия функции на языке Rust на первый взгляд может показаться немного более пугающей, но она очень похожа на версию на языке Python:

- обе функции принимают две переменные: строковый литерал в формате CSV и номер суммируемого столбца. В версии на Rust типы обозначены в явной форме, но в версии на Python всего лишь ожидается, что переменные также будут иметь эти типы, даже если они не обозначены;
- обе функции возвращают числа; опять же, Rust обозначает их в начале объявления функции в явной форме, тогда как Python этого не делает;
- обе функции выдают ошибки, если передаваемые им данные не соответствуют ожиданиям. В случае возникновения ошибки версия на Python генерирует исключения, а версия на Rust поднимает панику (подробнее об обработке ошибок читайте в главе 2);
- для достижения поставленных целей в обеих функциях используется одинаковый алгоритм синтаксико-структурного разбора данных CSV.

Несмотря на схожий внешний вид этих двух функций, они обладают совершенно разными характеристиками производительности. Версия на Python будет резервировать память под список строковых литералов, содержащий каждую строку во входном литерале CSV, помещать эти литералы в список и резервировать память под новый список строковых литералов для каждой строки значений, перечисленных через запятую. Благодаря надежным гарантиям, которые компилятор Rust предоставляет в отношении резервирования и вы свобождения памяти, версия на Rust безопасно использует одну и ту же опорную память под строковые литералы для всей функции, ни разу не перераспределяя ее. Вдобавок строковый метод Rust `.split` создает не список, а объект `Iterator`. Следовательно, вся последовательность подстрок перемещается по одной за раз, вместо того чтобы резервировать память под все сразу, как это делается в версии на Python. Это различие более подробно обсуждается в главе 3. Если размер входных данных составляет много миллионов строк или содержит много полей, то такое поведение будет сильно влиять на производительность.

Оба этих примера были использованы с одинаковым входным файлом, содержащим 1 миллион строк и 100 столбцов. В табл. 1.1 приведены данные об их соответствующем времени и максимальной используемости памяти.

Таблица 1.1 Различия в производительности между функциями агрегации CSV на языках Python и Rust

Версия	Время исполнения	Максимальная используемость памяти
Python	2.9 s	800 MiB
Rust	146 ms	350 MiB

Версия на языке Rust обеспечивает ускорение примерно в 20 раз и потребляет менее половины объема памяти. Это значительный прирост производительности без существенного увеличения сложности исходного кода. Этот пример был выбран специально; в вашем случае Rust может работать лучше или хуже.

1.4.2 Защищенность памяти

Как вариант вы, возможно, работаете над проектом на языке C или C++ и вам понадобится язык Rust из-за его преимуществ в плане защищенности и безопасности, которые он предоставляет по сравнению с этими языками. Во времена компиляции Rust может удостоверяться в том, что приложение защищено от дефектов использования памяти, таких как гонки данных, болтающиеся указатели и многое другое. За счет поступательной переработки критически важных частей кодовой базы на языке Rust можно быстрее поставлять программное обеспечение пользователям и тратить меньше времени на то, чтобы заботиться об инвариантах памяти в исходном коде. Пусть компилятор заботится об этом за вас!

Многие распространенные ошибки в коде на языках C и C++ просто невозможны выразить в обычном коде на языке Rust. Если попытаться написать код, содержащий эти ошибки, то компилятор Rust не примет программу, потому что он управляет одной из самых сложных частей программирования на C и C++ – владением памятью.

ПРИМЕЧАНИЕ. Опытные разработчики на языке C++, возможно, задумываются о разработке с использованием фреймворков, таких как популярный фреймворк Boost C++. Подобные библиотечные экосистемы существуют в языке Rust не так, как в языке C++, поскольку большинство пакетов (в терминологии языка Rust они называются упаковками – crates) взаимодействуют через интерфейс с использованием стандартных библиотечных типов и совместимы друг с другом.

Опытные программисты на языках C и C++, вероятно, будут знакомы с концепцией владения памятью, но всем этим разработчикам рано или поздно придется с этим столкнуться. Данная тема будет

рассмотрена подробнее в последующих главах, но ее суть состоит в том, что один дескриптор всегда управляет временем, когда резервируется и высвобождается часть памяти, и в таких случаях принято говорить, что указанный дескриптор «владеет» этой памятью. В типичной программе на С или С++ программист несет на себе полную ответственность за поддержание состояния владения памятью и обязан все время о нем помнить. Данные языки предоставляют очень мало инструментов для аннотирования того, какие значения принадлежат каким дескрипторам. С другой стороны, компилятор Rust требует, чтобы программы строго придерживались его модели владения памятью.

Владение памятью – одно из самых больших преимуществ разработки на языке Rust. Rust берет ошибки, которые традиционно возникали во время исполнения с непредсказуемыми или опасными последствиями, и превращает их в ошибки времени компиляции, которые могут быть устраниены еще до исполнения кода.

1.4.3 Удобство технического сопровождения

Когда объем проектов, написанных на языках программирования с динамической типизацией, начинает достигать десятков тысяч строк, вы можете столкнуться с такими вопросами, как «Что это за объект?» и «Какие свойства доступны?». Rust стремится решать такие вопросы в отношении строгих систем статических типов. Статическая типизация означает, что тип каждого отдельного значения в программе на языке Rust известен во время компиляции. В наши дни статическая типизация приобретает все большее распространение. Такие проекты, как TypeScript, Муру и Sorbet, добавляют проверку типов соответственно в JavaScript, Python и Ruby. В этих языках программирования никогда не было поддержки проверки типов, и количество усилий, затраченных на разработку указанных систем, показывает, *насколько* полезно знать тип значения заранее.

Применяемая в языке Rust система типов характеризуется большой мощью, но в большинстве случаев она остается в стороне. Типы входных и выходных значений у функций должны аннотироваться в явной форме, но типы переменных внутри функций обычно могут определяться компилятором статически без каких-либо дополнительных аннотаций. То, что типы не обозначены явно, не означает, что они неизвестны. Если в объявлении функции обозначено, что она принимает на входе только булево значение, то невозможно передать ей строковый литерал. Целый ряд интегрированных сред разработки (IDE) и плагинов редактирования способны показывать эти неявно определенные типы, чтобы помочь в разработке, но вам как разработчику не нужно писать их самостоятельно. Некоторые разработчики, возможно, занервничают по поводу статической типизации, поскольку в последний раз они сталкивались с ней, когда язык Java требовал использования следующего ниже синтаксиса в стиле Kafka.

Листинг 1.3 Инициализация словаря чисел, состоящего из списков чисел, на Java 1.6

```
HashMap<Integer, ArrayList<Integer>> map
= new HashMap<Integer, ArrayList<Integer>>();

ArrayList<Integer> list = new ArrayList<Integer>();
list.add(4);
list.add(10);

map.put(1, list);
```

Конкретизация типа каждой отдельной локальной переменной в каждой функции реально утомляет, в особенности когда язык нуждается в том, чтобы это выполнялось более одного раза. Та же операция на языке Rust занимает всего две строки, при этом явные типы не требуются.

Листинг 1.4 Инициализация словаря чисел, состоящего из списков чисел, на Rust

```
let mut map = HashMap::new();
map.insert(1, vec![4, 10]);
```

Как компилятор узнаёт, значения какого типа попадают в `map`? Он просматривает вызов функции `insert` и видит, что ей передается целое число в качестве ключа и список целых чисел в качестве значения. Тот же исходный код можно написать на языке Rust с явными аннотациями типов, но в большинстве случаев это совершенно не обязательно. Некоторые из этих случаев будут обследованы в главе 2.

Листинг 1.5 Инициализация словаря чисел, состоящего из списков чисел, на Rust с явно обозначенными типами

```
let mut map: HashMap<i32, Vec<i32>> = HashMap::new();
map.insert(1, vec![4, 10]);
```

Указанная строгая система типов обеспечивает, что позже при пересмотре исходного кода вы будете больше тратить время на добавление новых функциональных компонентов или повышение производительности и меньше беспокоиться о том, что означает пятый нетипизированный параметр функции `perform_action`.

1.5 Когда не следует перерабатывать исходный код на язык Rust

Если вы рассматриваете проект «на неосвоеной территории», то его не нужно перерабатывать на язык Rust; свое первоначальное решение можно написать на языке Rust! В этой книге в первую очередь предполагается, что у вас существует программный проект,

который вы хотите улучшить. Если вы только начинаете, то вам может пригодиться книга по программированию на языке Rust общего назначения. Кроме того, если ваш проект работает в среде, которую вы не очень хорошо контролируете, например в коллективной PHP-службе размещения сторонних веб-сайтов или на жестко контролируемых корпоративных серверах, где у вас нет возможности устанавливать новое программное обеспечение, то вы можете столкнуться с проблемами при использовании некоторых методов, описанных в этой книге.

При развертывании любого программного проекта всегда необходим план. Как вы собираетесь представлять проект пользователям? Обсуждаемый в этой книге тип переработки предполагает, что развертывание нового кода обходится довольно дешево и может проводиться часто. Если вам необходимо поставлять клиентам физические носители для новых версий либо в вашей организации принята очень жесткая релизная структура, то эта книга, возможно, не будет соответствовать вашим потребностям.

При написании нового программного обеспечения всегда следует планировать на долгие годы то, как оно будет сопровождаться в техническом плане. Если вы единственный разработчик в своей крупной компании, кто заинтересован в разработке на языке Rust, то, возможно, вы ставите перед собой планку стать «специалистом по Rust» на тот случай, если в будущем у этой системы неизбежно возникнут проблемы. Хотите ли вы быть единственным, кто отвечает за техническое сопровождение этой системы?

1.6 Как это работает?

Поступательная переработка зрелой производственной системы – задача не из легких, но ее можно разбить на несколько ключевых этапов:

1 Планирование.

- Что я надеюсь улучшить, переработав исходный код на язык Rust?
 - Если существующий код написан на языке C или C++, то следует подумать о том, как язык Rust улучшит защищенность памяти приложения.
 - Если существующий код написан на интерпретируемом языке со сбором мусора, таком как Python, то вы будете озабочены в основном улучшением производительности приложения.
- Какие части исходного кода следует перерабатывать?
- Как существующий код должен взаимодействовать с новым кодом?

2 Реализация.

- Реплицирование функциональности существующего кода в новом коде на языке Rust.
- Интеграция исходного кода на языке Rust в существующую кодовую базу.

- 3 Верификация.
 - Использование принятых в языке Rust средств тестирования для тестирования новой функциональности.
 - Использование существующих тестов для сравнения результатов двух путей исполнения кода.
- 4 Развёртывание.
 - В зависимости от принятых ранее решений необходимо иметь разные пути исполнения исходного кода на языке Rust при обслуживании клиентов.
 - Как эффективно развертывать переработанный исходный код, не затрагивая при этом конечных пользователей?

Эти этапы и некоторые из их более тонких составляющих подробно перечислены на рис. 1.2.

Как видно по рис. 1.2, большую часть этого процесса занимает планирование. Выполнение такого рода работ по переработке сопряжено со сложностями, и для нее требуется осознание последствий замены исходного кода до того, как исходный код будет заменен. Кроме того, необходимо тщательно продумывать производительность и удобство технического сопровождения, которые идут рука об руку с развертыванием новых шаблонов исходного кода. После планирования самый большой раздел занимает развертывание, где осуществляется контроль за тем, какие пользователи получат доступ к новым функциональностям вместо старых.

1.7 Чему вы научитесь в этой книге?

В этой книге рассматривается поступательная переработка исходного кода в абстрактном смысле, а затем изложение переходит к вопросу о том, как язык Rust конкретно помогает применять подход поступательной переработки и как его можно внедрять в свои приложения. Существует два главных метода интеграции исходного кода на языке Rust в существующие приложения, и каждый из них имеет несколько вариаций.

1.7.1 Вызов функций Rust непосредственно из своей программы

В этой модели создается библиотека Rust, которая работает как библиотека, написанная на вашем существующем языке программирования. В этом разделе различные методы рассматриваются на высоком уровне, которые потом будут подробно рассмотрены в последующих главах. Данная модель проиллюстрирована на рис. 1.3.

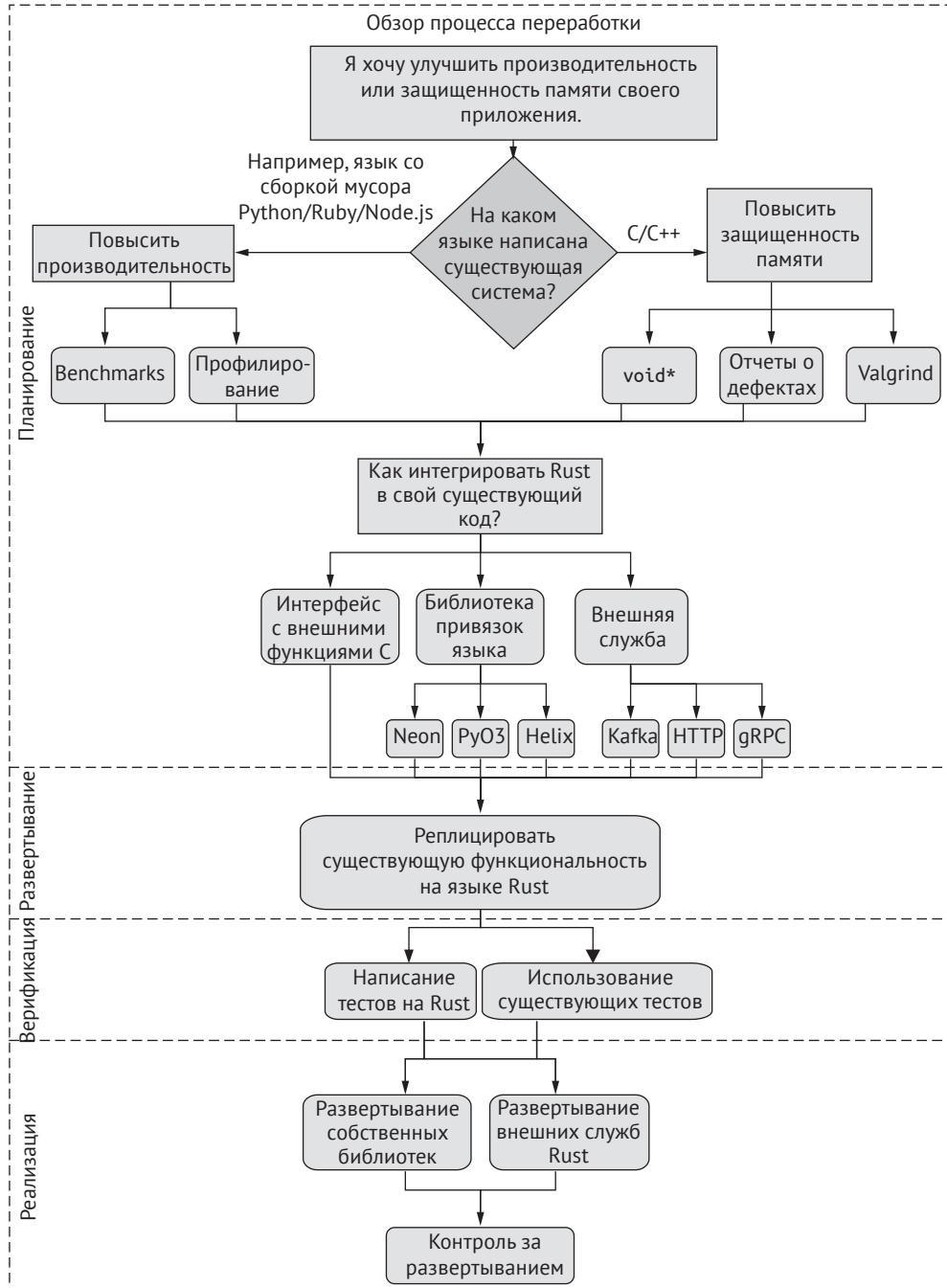


Рисунок 1.2 Обзор обсуждаемого в этой книге процесса переработки

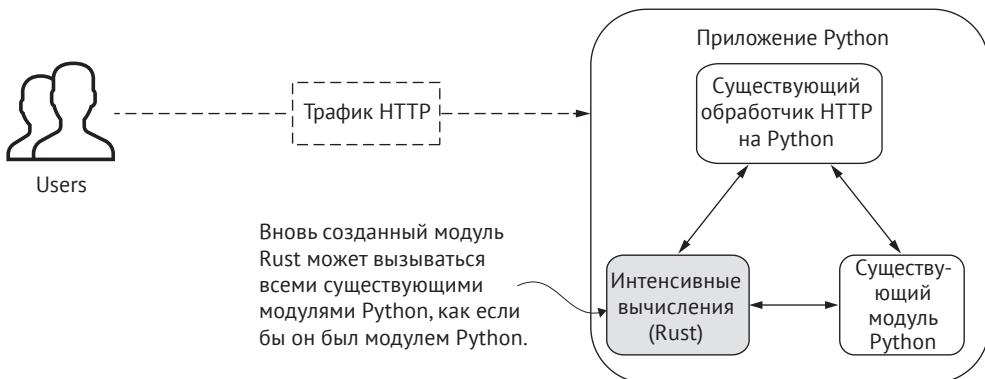


Рисунок 1.3 При вызове Rust непосредственно из своего существующего приложения исходный код Rust выглядит как обычный модуль

Например, если вы перерабатываете Python’овский проект, то ваша библиотека Rust будет выставлять наружу функции и классы, которые действуют подобно функциям и классам Python. Этот метод будет иметь минимально возможные издержки на обмен данными между существующим кодом и новым кодом Rust, поскольку они оба исполняются как часть одного и того же процесса операционной системы и могут делиться памятью напрямую друг с другом.

У этого подхода есть несколько ответвлений.

- Использование с интерфейсом с внешними функциями (FFI).
 - Эта тема подробно обсуждается в главе 3, но его суть состоит в том, что Rust будет позволять писать функцию, которая выглядит как функция C, с учетом того, что многие другие языки знают, как вызывать функции C.
 - Этот подход является наиболее универсальным, поскольку большинство часто используемых языков программирования понимают интерфейс с внешними функциями C (C FFI).
 - Такой подход наиболее подвержен дефектам использования памяти, поскольку программист будет непосредственно отвечать за то, чтобы память резервировалась, высвобождалась и передавалась туда и обратно правильно, а владение всегда было ясным.
 - Если ваши проекты написаны на языке C или C++, то вы будете использовать именно этот метод интеграции.
- Использование библиотек Rust для непосредственной привязки к интерпретатору другого языка.
 - Используя этот метод, можно написать библиотеку языка Rust, которая выглядит точно так же, как, например, библиотека языка Python, Ruby или Node.js.
 - Этот метод, нередко реализуемый проще, чем подход на основе интерфейса с внешними функциями C (C FFI), ломается, если привязки языка Rust недоступны к языку, который вы хотите использовать.

- Компиляция Rust в WebAssembly (Wasm) и использование интерфейса с внешними функциями Wasm (Wasm FFI).
 - Wasm – это формат байт-кода для движков JavaScript, аналогичный байт-коду Java. Многие языки (включая Rust) могут компилироваться в Wasm вместо собственного машинного кода.
 - Этот подход полезен при использовании языка Rust с браузерными движками JavaScript или Node.js.

1.7.2 Обмен со службой Rust по сети

Этот метод основан на использовании сетевого протокола для обмена данными со свежесозданной службой Rust. Его концепция проиллюстрирована на рис. 1.4.

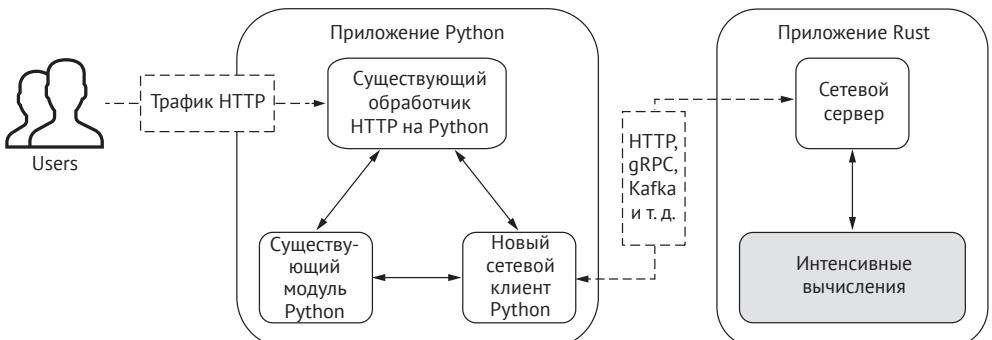


Рисунок 1.4 Когда исходный код на языке Rust находится во внешней службе, возникают дополнительные издержки из-за перехода по сети

Указанный подход имеет ряд преимуществ и недостатков по сравнению с ранее рассмотренной моделью.

■ Преимущества:

- поскольку этот метод не имеет прямого доступа к памяти, отсутствует риск повредить память при взаимодействии между двумя языками;
- такой подход позволяет масштабировать свою систему, написанную на языке Rust, независимо от своего существующего приложения;
- все больше разработчиков имеют опыт сетевого обмена между приложениями, поэтому это такой концептуальный прорыв, как идея сосуществования нескольких языков программирования в одном приложении.

■ Недостатки:

- как упоминалось в предыдущем разделе, из-за дополнительного времени, необходимого для передачи данных по сети, часть производительности будет потеряна;

- добавление дополнительной службы с собственной независимой логикой журналирования, мониторинга и развертывания сопряжено с дополнительными операционными издержками.

1.8 Для кого предназначена эта книга?

Эта книга написана для программистов, которые уже имеют многолетний опыт работы с приложениями на языке, отличном от Rust, и ищут способы повысить производительность, защищенность или удобство технического сопровождения своих приложений.

Данная книга также будет полезна программистам на языке Rust, которые хотят применить свои знания, чтобы повысить производительность или защищенность памяти у существующих приложений, написанных на других языках. Исходного кода, написанного не на языке Rust, гораздо больше, чем кода, написанного на нем.

Разумеется, примеры исходного кода будут в основном написаны на языке Rust, но поскольку в этой книге рассматривается переход с других языков на язык Rust, требуется что-то для сравнения. В главе 3 приведено много примеров исходного кода на языках С и С++, а во многих остальных главах приведены примеры исходного кода на языке Python, чтобы подчеркнуть различия между ним и языком Rust и показать, как работают методы интеграции. Вам не нужно быть экспертом в этих языках; опыта работы с другими процедурными языками семейства языка С должно быть достаточно.

В главе 3 рассматриваются многие вопросы, касающиеся защищенности памяти, которые могут быть непонятны разработчикам, в основном работающим с языками, поддерживающими сборку мусора во время исполнения. Эти темы не требуются для переработки с указанных языков со сборкой мусора; они в основном предназначены для читателей, знакомых с языками С и С++.

1.9 Какие инструменты потребуются для начала работы?

Все программные средства, необходимые для начала работы, находятся в свободном доступе. Вам понадобится:

- *свежий компилятор Rust* – инструкции по установке языка Rust находятся по адресу <https://www.rust-lang.org/tools/install>;
- *текстовый редактор*, подходящий для программирования;
- *компьютер или виртуальная машина, работающие под управлением операционной системы GNU/Linux* – большинство приведенных в этой книге примеров программирования на языке Rust будут работать в любой операционной системе, но некоторые примеры написаны с учетом операционной системы GNU/Linux:
 - если вы используете Microsoft Windows, то подсистема Windows для Linux (WSL) предоставляет удобный способ вы-

- полнения программ Linux, которые интегрируются с обычной средой Windows;
- все примеры в книге протестированы на Ubuntu 20.04, работающей под управлением WSL;
 - *пакеты разработки Libclang* – опять же, они не являются обязательным условием для выполнения упражнений по программированию на Rust, но во многих главах используется Libclang (косвенно), чтобы генерировать код для обмена данными между кодом на Rust и на C/C++;
 - *Python 3*, *virtualenv* и *pip* – они необходимы для исполнения Rust-ориентированных Python'овских модулей расширения, описанных в последующих главах.

Краткий итог

- Переработка исходного кода применяется поступательно для замены по одной небольшой части исходного кода за один раз. Более частое внесение небольших изменений помогает повышать производительность без затрат времени и усилий на масштабное переписывание.
- Язык Rust обладает строгой системой статических типов, которая обеспечивает четкое определение входных и выходных значений и обработку крайних случаев.
- Язык Rust обеспечивает простой параллелизм, благодаря чему можно использовать и без того быстрый исходный код на языке Rust и задействовать все доступные ресурсы процессора с целью достижения максимальной производительности.
- Язык Rust легко интегрируется с другими языками и позволяет сосредоточиваться на повышении эффективности, не беспокоясь о том, чтобы изобретать велосипед заново.
- Переработка исходного кода на язык Rust повышает производительность, защищенность памяти и удобство технического сопровождения, что в долгосрочной перспективе поможет быстрее и с меньшими затратами масштабировать программные системы.



Обзор языка Rust

Эта глава охватывает следующие ниже темы:

- разработка систем, которые надлежаще используют применяемую в языке Rust систему владения;
- визуализация применяемой в языке Rust системы времен жизни с целью облегчения отладки;
- управление резервированием памяти под строковые литералы с целью повышения производительности;
- перечисления и базовая обработка ошибок.

Прежде чем можно будет интегрировать библиотеку языка Rust в существующее приложение, написанное на другом языке, сначала нужно понять основы программирования на языке Rust. В этой главе вы познакомитесь с простым приложением по управлению цифровыми произведениями искусства для художественного музея и узнаете о том, как работает система владения. Многие считают, что концепции владения и заимствования являются одними из самых сложных для начинающих разработчиков на языке Rust. Мы начинаем именно с них, а не с чего-то более простого, потому что как раз в этих областях язык Rust больше всего отличается от других языков программирования, и они лежат в основе всех программ на языке Rust. Если не уделить время изложению этих важных идей сейчас, то дальнейшая работа с книгой значительно усложнится. Мы собираемся привести пример, который увязывает компоненты владения и заимствования

в программах на языке Rust с владением и использованием цифровых произведений искусства. Этот процесс должен упростить рассуждения о владении, и мы представим инструменты визуализации изменений во владении с течением времени.

2.1 Владение и заимствование

Одно из самых больших различий между языком Rust и другими языками программирования заключается в навязывании исполнения нескольких очень важных правил, касающихся способов доступа к данным и зависимостей между различными формами доступа к данным. Эти правила не слишком сложны, но они отличаются от многих других языков, в которых исполнение таких правил не навязывается. Правила владения таковы:

- каждому значению в языке Rust соответствует переменная, которая называется его владельцем;
- одновременно может быть только один владелец;
- если владелец выходит за пределы диапазона доступности, то значение удаляется.

При первом просмотре исходного кода на языке Rust, возможно, соблюдение этих правил будет не совсем очевидно. Процедурный исходный код на языке Rust может выглядеть очень похожим на код, написанный на других языках, и вы, возможно, сможете прослеживать его логику без каких-либо проблем. Однако, вероятно, вы обнаружите, что при попытке отредактировать существующий исходный код на языке Rust или написать свой собственный у вас возникают трудности с получением такого, который выглядит вполне приемлемым для компиляции. Эта трудность заключается в том, что компилятор Rust навязывает исполнение этих правил, которые вы еще не вполне усвоили.

Мы рассмотрим простой пример задачи, чтобы продемонстрировать, как правила владения и заимствования могут влиять на программу на языке Rust. Представьте себе, что к вам обратился художественный музей; они хотят, чтобы вы разработали систему на языке Rust, которая позволит им управлять своим каталогом произведений искусства в цифровом плане. Система должна позволять посетителям приобретать билеты, дающие им право просматривать работы.

Мы начнем с создания нового проекта на языке Rust, используя менеджер пакетов Rust под названием Cargo. Для того чтобы начать новый проект с помощью Cargo, используется команда `cargo new`, за которой следует название проекта, который требуется создать:

```
$ cargo new art-museum
```

Приведенный выше исходный код создает новый каталог с именем `art-museum`; в нем есть все файлы, необходимые для того, чтобы начать писать исходный код на языке Rust. Давайте сейчас сосредоточимся на главном сгенерированном файле исходного кода, `art-museum/src/`

`main.rs`. Откройте этот файл в своем любимом текстовом редакторе, и вы сможете приступить к работе.

Когда вы откроете этот файл в первый раз, то, наверное, будете удивлены, обнаружив, что он не пустой и на самом деле уже содержит, пожалуй, самый известный из всех примеров программирования – программу «Hello world!».

Листинг 2.1 Программа «Hello world!» на языке Rust

```
fn main() {
    println!("Hello world!");
}
```

Символ ! после `println` указывает на то, что это не функция, а макрокоманда.

В большинстве программ на языке Rust в качестве точки входа используется функция `main`. Все определения функций Rust содержат ключевое слово `fn`, за которым следует имя определяемой функции.

С помощью еще одной команды Cargo – `cargo run` – эту программу можно исполнить, чтобы убедиться в том, что она выводит именно то, что от нее ожидается. Команда `run` сообщает менеджеру пакетов Cargo, что нужно скомпилировать приложение Rust и исполнить результатирующий исполняемый файл. Команда `cargo run` будет одной из наиболее часто используемых команд:

```
$ cargo run
Hello world!
```

Давайте заменим исходный код в программе «Hello world!» на исходный код программы художественного музея. Начнем с определения типа, который представляет произведения искусства в музее.

Листинг 2.2 Структура, представляющая произведение искусства

```
struct Artwork {
    name: String,
}

fn main() {
    let art1 = Artwork {
        name: "Boy with Apple".to_string()
    };
}
```

Структуры `struct` – это наборы полей, представляющих отдельные логические значения. Структуры в языке Rust похожи на классы в объектно-ориентированных языках программирования, но они не поддерживают наследование, как классы. Они больше похожи на структуры в таких языках, как C++ или Go, так как позволяют разработчикам совмещать данные с функциональностью.

При инициализации новой переменной на языке Rust используется инструкция `let`. Компилятор способен определять тип создаваемой переменной на основе значения в правой части знака равенства.

Может показаться странным, что цепочка символов "Boy with Apple" недостаточно хороша, для того чтобы быть строковым литералом самим по себе, и нуждается в вызове дополнительной функции, чтобы считаться значением типа `String`; эта ситуация будет рассмотрена подробнее в разделе 2.3. На данный момент следует запомнить, что вызов метода `to_string()` необходим для превращения цепочки символов в значение типа `String`. Первая операция, которую вы, возможно, захотите смоделировать, – это просмотр произведения искусства.

Листинг 2.3 Исходный код, позволяющий любоваться произведением искусства

```
struct Artwork {  
    name: String,  
}  
  
fn admire_art(art: Artwork) {  
    println!("Wow, {} really makes you think.", art.name);  
}  
  
fn main() {  
    let art1 = Artwork { name: "La Trahison des images".to_string() };  
    admire_art(art1);  
}
```

Вместо фигурных скобок в строковом литерале, передаваемом в макрокоманду `println!`, подставляются значения, указанные после начального строкового аргумента. Этот процесс аналогичен подстановкам в стиле форматных строковых литералов, которые такие языки, как C и Go, предоставляют в функции `printf`, а такие языки, как Python, предоставляют в методе `.format` на строковых литералах.

Теперь есть функция под названием `admire_art`, которая принимает одно значение типа `Artwork` в качестве единственного аргумента и печатает сообщение о том, насколько фантастично это произведение искусства. Данная программа должна напечатать следующее:

```
$ cargo run  
Wow, La Trahison des images really makes you think.
```

Пока что эта система выглядит довольно привлекательной: у вас есть произведение искусства, и вы, наверное, испытываете тихое восхищение. И то, и другое – ключевые элементы любого художественного музея. Поскольку вы управляете не самым маленьким художественным музеем в мире, давайте добавим еще одно произведение искусства!

Листинг 2.4 Программа, в рамках которой можно полюбоваться двумя произведениями искусства

```
struct Artwork {  
    name: String,  
}
```

```

fn admire_art(art: Artwork) {
    println!("Wow, {} really makes you think.", art.name);
}

fn main() {
    let art1 = Artwork { name: "Las dos Fridas".to_string() };
    let art2 = Artwork { name: "The Persistence of Memory".to_string() };
};

    admire_art(art1);
    admire_art(art2);
}

```

Результаты этой программы не должны вызывать удивления у всех, кто следит за ее логикой:

```

$ cargo run
Wow, Las dos Fridas really makes you think.
Wow, The Persistence of Memory really makes you think.

```

Конечно, восхищаться двумя произведениями искусства – это хорошо, но представьте себе, что у этого музея несколько посетителей, которые хотят смотреть на одно и то же произведение искусства. В листинге 2.5 показано, как этот исходный код может выглядеть.

Листинг 2.5 Программа пытается дважды восхититься одним и тем же произведением искусства

```

struct Artwork {
    name: String,
}

fn admire_art(art: Artwork) {
    println!("Wow, {} really makes you think.", art.name);
}

fn main() {
    let art1 = Artwork { name: "The Ordeal of Owain".to_string() };
    admire_art(art1);
    admire_art(art1);
}

```

Если попытаться выполнить эту, казалось бы, разумную программу, то будет получена ошибка компилятора – ошибка, которая, вероятно, покажется совершенно необычной для тех, кто раньше не работал с языком Rust. Давайте взглянем на нее:

```

$ cargo run
error[E0382]: use of moved value: `art1`
--> src/main.rs:11:16
|   |
8 |     let art1 = Artwork {};
|     ---- move occurs because `art1` has type `Artwork`, which
|           does not implement the `Copy` trait

```

```

9 |
10 |     admire_art(art1);
|             ---- value moved here
11 |     admire_art(art1);
|             ^^^^ value used here after move
|
error: aborting due to previous error; 1 warning emitted

```

Что здесь происходит? Что означает использование перемещенного значения (`use of moved value`)? Что такая общая черта `Copy` (``Copy` trait`)? Что вообще компилятор Rust пытается сообщить?

Компилятор Rust пытается сообщить о том, что вы нарушили правило владения и, следовательно, программа неработоспособна. Но прежде чем можно будет обсудить причины, по которым этот код не работает на языке Rust, нужно провести краткий обзор того, как осуществляется управление памятью в других языках программирования.

2.2 Управление памятью в других языках

Компьютерные программы хранят данные, которые они используют или генерируют во время исполнения, как правило, в памяти компьютера. Память обычно делится на две части: стек и кучу.

Стек используется для хранения локальных переменных, созданных внутри работающей в данный момент функции, и функций, которые привели к вызову текущей функции. Максимальный размер стека ограничен и обычно составляет 8 Мб. Он всегда растет, как стопка бумаг, то есть когда добавляются или удаляются какие-либо значения, они добавляются или удаляются сверху, вследствие чего в стопке нет пустот.

С другой стороны, куча ограничена только объемом памяти компьютера, на котором работает программа, и она может составлять гигабайты или терабайты. Следовательно, куча используется для хранения гораздо более крупных данных или данных, точный размер которых неизвестен до запуска программы. Такие объекты, как массивы и строковые литералы, обычно хранятся в куче. Память, связанная с кучей, также называется *динамической памятью*, поскольку размер значений в куче неизвестен до тех пор, пока программа не будет запущена.

Представьте себе, что требуется поприветствовать посетителя, когда он входит в художественный музей, сказав ему «Welcome {name}». Для этого сначала нужно запросить, чтобы компьютер зарезервировал достаточно места в памяти под хранение имени посетителя, которое сохраняется в переменной `name`. Этот процесс называется *резервированием памяти*. В той области памяти не может храниться ничего, кроме значения имени этого пользователя. Значение в памяти можно заменить или поменять, присвоив новое значение переменной `name`, но переменная `name` всегда будет ссылаться на ту же область в памяти.

При этом необходимо периодически очищать память программы, иначе она в конечном итоге заполнится неиспользуемыми значениями.

ми переменной `name`. После того как приветственное сообщение было успешно напечатано, переменная `name` больше не используется, и нужно сообщить компьютеру о том, что память, которая была связана с этой переменной, можно реиспользовать для других целей, потому что мы ее больше не используем. В языке Rust этот процесс очистки называется *отбрасыванием* значения, но более общим термином является *высвобождение* памяти. В прошлом было два распространенных способа, которыми разработчики на разных языках программирования могли резервировать и высвобождать память:

- разработчик может написать код, который запрашивает требуемый объем памяти в явной форме и отмечает точку, в которой память больше не используется и может быть очищена. Этот процесс называется *ручным управлением памятью*, так как он требует от разработчика ручных усилий по обеспечению резервирования и высвобождения памяти, когда это необходимо. Многие языки с ручным управлением памятью автоматически высвобождают значения из стековой памяти программы, когда функция, зарезервированная под них память, возвращается, и стековый кадр завершает работу. Большая проблема, которая связана с этими языками, – это управление кучей;
- язык может содержать дополнительный код, который работает на фоне всех программ, чтобы периодически проверять, не осталось ли переменных, ссылающихся на зарезервированные блоки памяти, и высвобождать их. Этот процесс называется *сборкой мусора*, или *автоматизированным управлением памятью*, поскольку от разработчика не требуется никаких действий вручную, чтобы высвободить память. Эти языки, как правило, также имеют гораздо более простые методы резервирования памяти, что не позволяет разработчику запрашивать слишком много или слишком мало памяти для хранения значения заданного типа.

Если вы заинтересованы в написании высокопроизводительных программ, то вам, как правило, придется использовать языки, которые предоставляют разработчику инструменты ручного управления памятью. Такие языки, как C и C++, требуют, чтобы программист определял нужный объем памяти и запрашивал у компьютера резервирование именно этого объема памяти. Запрос слишком большого объема может приводить к замедлению резервирования памяти или чрезмерной используемости памяти. Запрос слишком малого объема и неправильное использование памяти за пределами зарезервированного блока могут вызывать серьезные проблемы. Эти проблемы могут приводить к отказу программ, выставлению наружу областей памяти, которые должны быть секретными (например, пароли, ключи шифрования и т. д.), или позволять злоумышленникам внедрять код врабатывающую программу и захватывать ее. Попытка написать большую программу на языке, который требует от разработчика ручного управления памятью, требует от него больших умственных усилий – или, по меньшей мере, большого объема документирования.

Одной из наиболее распространенных проблем, возникающих при ручном управлении памятью, является идея «использовать после вы-свобождения», которая возникает при попытке использовать область памяти после того, как она была высвобождена. Она, возможно, была перепрофилирована под хранение чего-то другого, вероятно, была обнулена, или все еще может содержать данные, что, по вашему мнению, так и есть. Компилятор может делать с высвобожденной памятью все, что ему заблагорассудится.

Представьте себе, что вы хотите написать простую программу, используя воображаемый язык программирования, который назовем «К». Язык программирования К очень похож на язык программирования Python, за исключением того, что К требует от разработчика явного высвобождения динамической памяти путем вызова функции `free` на значениях. Вы должны вызывать `free` на каждом значении, зарезервированном в динамической памяти, и вызывать ее нужно ровно один раз. Если попытаться использовать высвобожденное значение, то программа завершит работу аварийным отказом. Давайте попробуем написать приветственную программу, используя язык К.

Листинг 2.6 Программа приветствия, написанная на языке К

```
def welcome(name):
    print('Welcome ' + name)

name = input('Please enter your name: ')
welcome(name)
free(name)
```

Приведенный выше исходный код запрашивает у пользователя имя, отправляет ему персонализированное приветственное сообщение, а затем высвобождает память, используемую под хранение его имени. Эта программа прекрасна, думаете вы про себя, но в большинстве случаев вызова функции `welcome` разве не нужно высвобождать строковый литерал в следующей строке? Давайте перенесем вызов `free` внутрь функции `welcome`, чтобы не нужно было помнить о ее вызове.

Листинг 2.7 Программа приветствия с высвобождением внутри функции `welcome`

```
def welcome(name):
    print('Welcome ' + name)
    free(name)

name = input('Please enter your name: ')
welcome(name)
```

Перенос вызова `free` внутрь функции `welcome` избавляет от необходимости помнить о вызове `free` при каждом вызове функции `welcome`. В этом небольшом примере совершенно очевидно, что программа по-прежнему работает, но в ней было создано тонкое недокументи-

рованное поведение функции `welcome`. Любой строковый литерал, переданный в функцию `welcome`, теперь становится непригодным для использования после ее вызова. Если имеется 10 000 строк кода, то теперь придется проверять каждый вызов функции `welcome`, чтобы убеждаться в том, что переданные в нее строковые литералы никогда не будут реиспользованы, иначе возникает риск привести программу к аварийному отказу.

Если бы логика функции `welcome` была обновлена поддержанием журнала посетителей, которые заходили в музей с определенного входа, то пришлось бы изменить функцию `welcome`, чтобы снова не высвобождать переданные ей строковые литералы. Этот процесс снова требует изучения кодовой базы, просмотра всех вызовов `welcome` и выявления того, что именно следует делать сразу после этого: высвобождать имя либо заносить его в журнал. Программист должен принимать все эти решения до выполнения программы, но язык К не предоставляет никаких инструментов проверки правильности работы программы, кроме как путем ее выполнения.

Здесь уже можно начать видеть преимущества применяемой в языке Rust системы владения. В языке Rust информация о том, когда была зарезервирована память, когда допустимо ее использовать и когда она высвобождается, кодируется на уровне типов. Знание этой информации защищает от ошибок «использования после высвобождения» и многих других типов ошибок, связанных с повреждением памяти. Их просто невозможно выразить на языке Rust. Компилятор будет останавливать выполнение программ, если они будут нарушать правила языка Rust.

Также хорошо видно, что программы на языке Rust обладают лучшим из обоих миров: и преимуществами сбора мусора, и ручного управления памятью. Имеется скорость ручного управления памятью, потому что в программе на языке Rust не выполняется дополнительный процесс фонового сканирования памяти, и можно быть спокойным, зная, что компилятор защитит от ошибок использования памяти, которые могут приводить к аварийному отказу программы или к чему-то худшему.

Вспомните исходный код из листинга 2.5. Ниже он приводится снова.

Листинг 2.8 Повтор исходного кода из листинга 2.5

```
struct Artwork {
    name: String,
}

fn admire_art(art: Artwork) {
    println!("Wow, {} really makes you think.", art.name);
}

fn main() {
    let art1 = Artwork { name: "The Ordeal of Owain".to_string() };
    admire_art(art1);
    admire_art(art1);
}
```

Определив функцию `admire_art`, вы сообщили Rust о том, что при вызове функции источник вызова должен предоставлять функции владеемое значение типа `Artwork` и что функция будет брать это значение в свое владение. Напомним, что во всех программах Rust у каждого значения может быть только один владелец. Поскольку переменная `art1` владеет значением типа `Artwork`, на которое она ссылается при вызове функции `admire_art` с переменной `art1` в качестве параметра, Rust удаляет у переменной `art1` владение этим значением и перемещает владение художественным произведением переменной `art` внутри функции `admire_art`. Этот шаг очень важен: после первоначального вызова функции `admire_art` переменная `art1` больше недействительна, так как она больше ни на что не ссылается и, следовательно, не может быть использована. При вызове функции `admire_art` с любым значение типа `Artwork` память, связанная с этим художественным произведением, высвобождается в момент завершения работы функции.

Понимание принципов владения и перемещения имеет решающее значение при написании исходного кода на языке Rust, но не менее важным является понимание времен жизни.

2.3 Времена жизни

Применяемая в языке Rust концепция времен жизни лежит в основе понимания процесса управления памятью. У всех значений во всех языках программирования есть время жизни, хотя в большинстве из них это обозначено не так явно, как в языке Rust. Время жизни значения описывает период времени, в течение которого это значение является действительным. Если это локальная переменная в функции, то ее временем жизни может быть время, в течение которого вызывается функция. Если это глобальная переменная, то она может существовать в течение всего времени исполнения программы. Значение является действительным в течение времени после резервирования памяти и до отбрасывания. Попытка использовать значение в любое время, выходящее за пределы этого диапазона, недопустима. В таких языках, как C и C++, использование значения, время жизни которого истекло, может приводить к отказам или ошибкам, связанным с повреждением памяти. В языке Rust это приводит к тому, что программа не будет компилироваться.

В целях облегчения понимания давайте введем новый тип визуализации, который назовем «графиком времен жизни». Такие графики будут часто появляться в этой главе и периодически на протяжении всей книги. Прежде чем попытаться наглядно представить ошибку из листинга 2.5, давайте сначала рассмотрим более простой пример, приведенный ранее в этой главе. На рис. 2.1 показан график времен жизни для листинга 2.2; исходный код приведен для удобства.

Точка, в которой у компьютера запрашивается резервирование памяти под хранение значения переменной `art1`.

В этом поле представлены все переменные, которые создаются и используются при вызове функции `main`.

Время жизни переменной `art1`. Подобного рода линию, которая показывает, когда значение создано, когда оно является действительным и когда оно отбрасывается, необходимо нарисовать для каждой переменной в программе Rust.

Точка непосредственно перед завершением работы функции `main`, когда содержащая переменную `art1` память отбрасывается.

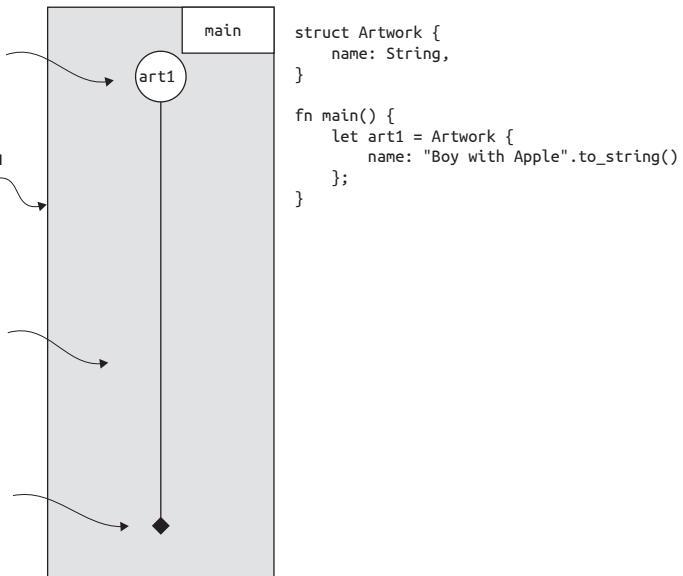


Рисунок 2.1 График времен жизни для листинга 2.2

Обратите внимание, что переменная `art1` содержит одну строку, которая показывает, когда переменная была создана, когда ее можно использовать и когда она уничтожается. В языке Rust значения отбрасываются, когда они выходят за пределы диапазона доступности. Локальные переменные внутри функции отбрасываются непосредственно перед завершением работы функции. Когда возникают трудности в решении проблем с применяемой в языке Rust системой управления памятью, принято опираться на эти графики, которые помогают понимать, что происходит.

Теперь давайте посмотрим, как выглядит время жизни для листинга 2.3.

На рис. 2.2 представлена концепция «перемещения» значения или передачи владения значением другой переменной. Как вы знаете из обсуждения листинга 2.3, при вызове функции `admire_art` с параметром `art1` оно «перемещается» из функции `main` в функцию `admire_art`. После этого оно больше недоступно из функции `main`. Исчезновение времени жизни для переменной `art1` из функции `main` сразу при запуске функции `admire_art` намекает на то, что ее значение было перемещено.

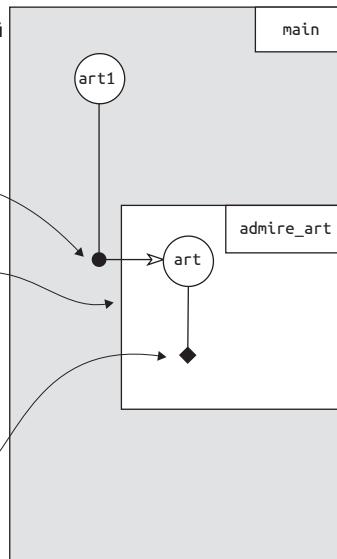
Если визуализировать исходный код, приведенный в листинге 2.4, то можно увидеть, как это выглядит при существовании двух переменных со своими собственными независимыми временами жизни.

На рис. 2.3 хорошо видно, что каждая из двух переменных типа `Artwork` создается в функции `main`, а затем перемещается в разные места вызова внутри функции `admire_art`. Каждая переменная имеет свое собственное независимое время жизни, и у каждой есть соответствующие начало, середина и конец.

«Перемещение» означает, что владение значением передается от одной переменной к другой. В этом случае владение перемещается из переменной `art1` в функции `main` в переменную `art` в функции `admire_art`.

В этом поле представлены все переменные, созданные и используемые при вызове функции `admire_art`; обратите внимание, что она появляется внутри блока для функции `main`.

Обратите внимание, что в функции `main` больше нет символа отбрасывания. Поскольку владение переменной `art1` было передано функции `admire_art`, функция `main` больше несет ответственности за ее отбрасывание.



```

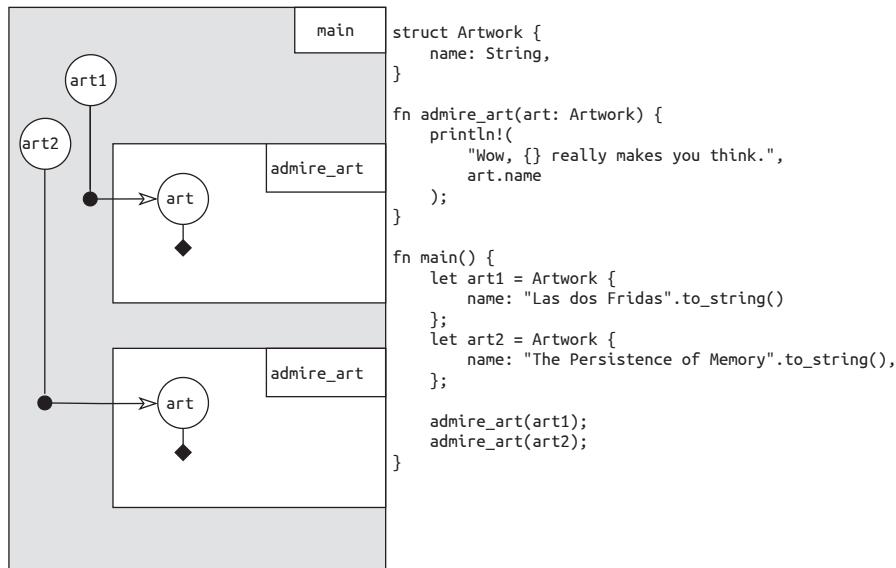
struct Artwork {
    name: String,
}

fn admire_art(art: Artwork) {
    println!(
        "Wow, {} really makes you think.",
        art.name,
    );
}

fn main() {
    let art1 = Artwork {
        name: "La Trahison des images"
            .to_string(),
    };
    admire_art(art1);
}

```

Рисунок 2.2 График времен жизни для листинга 2.3



```

struct Artwork {
    name: String,
}

fn admire_art(art: Artwork) {
    println!(
        "Wow, {} really makes you think.",
        art.name,
    );
}

fn main() {
    let art1 = Artwork {
        name: "Las dos Fridas".to_string()
    };
    let art2 = Artwork {
        name: "The Persistence of Memory".to_string(),
    };
    admire_art(art1);
    admire_art(art2);
}

```

Рисунок 2.3 График времен жизни для листинга 2.4

При попытке построить время жизни для листинга 2.5 начинают возникать проблемы. Давайте посмотрим, получится ли пролить свет на то, что происходит, взглянув на визуализацию на рис. 2.4.

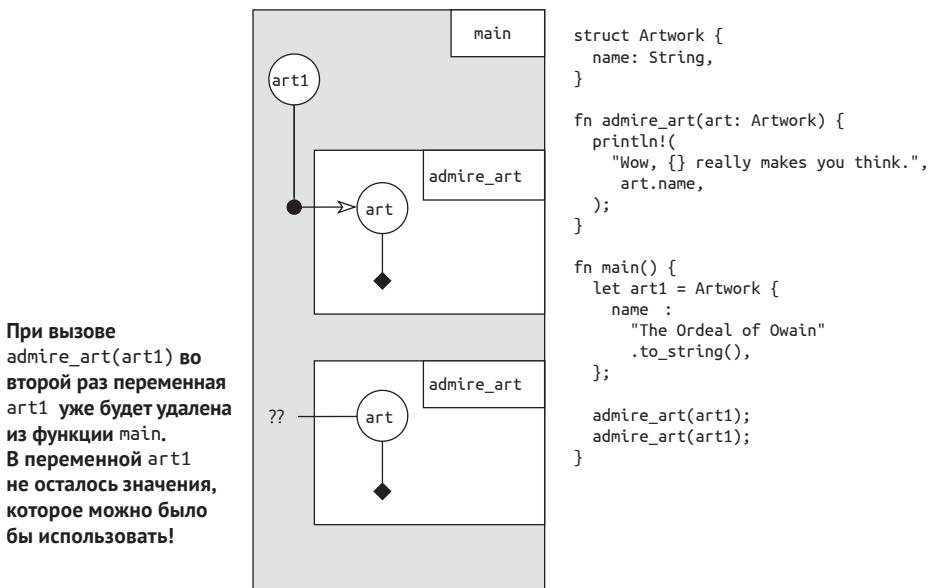


Рисунок 2.4 График времен жизни в листинге 2.5

Давайте разберем происходящее по полочкам. Обратите внимание, что значение переменной `art1` перемещено в функцию `admire_art` и больше недоступно из функции `main`. При попытке вызвать `admire_art` во второй раз значение исчезает; в переменной ничего нет, о чём и пытается сказать сообщение об ошибке, предоставленное компилятором Rust. Вспомните, что в заголовке этого сообщения об ошибке было указано «использование перемещенного значения». В исходном коде значение переменной `art1` было перемещено из функции `main`, но была предпринята попытка его использовать из функции `main`. Другими словами, делается попытка использовать значение, после того как оно было перемещено, что делает ее недопустимой.

В этот момент вы, возможно, задаетесь вопросом: «Ну и что? Почему значения должны исчезать, когда я передаю их функции? Все это отслеживание похоже на пустую трата времени!» Может показаться, что Rust влагает на программиста дополнительную нагрузку, просто усложняя его жизнь, но правда состоит в том, что программисты, использующие языки с ручным управлением памятью, такие как C или C++, должны постоянно следовать подобным правилам. Единственное отличие заключается в том, что там компилятор не навязывает исполнение правил; программист должен помнить о необходимости следовать им!

Давайте вкратце обсудим вопрос о том, как писать функции, которые не принимают во владение используемые ими значения.

2.3.1 Ссылки и заимствования

Если только вы не пишете программу, которая использует каждый фрагмент данных только один раз, вы обнаружите, что передача значений путем их перемещения чрезвычайно сковывает. В какой-то момент захочется использовать одно и то же значение из нескольких мест или использовать значение без передачи владения им. В языке Rust значения можно заимствовать, а не владеть ими. Заимствование значения в языке Rust всегда приводит к появлению *ссылки* на то, что заимствуется. Ссылки можно рассматривать как значения, которые подсказывают Rust, как найти другие значения. Если представить память своего компьютера в виде огромного массива значений, то ссылки в этом массиве подобны индексам, которые позволяют находить в нем значения.

Заимствование значения в языке Rust очень похоже на заимствование физического объекта в реальной жизни. Поскольку мы не владеем используемым значением, его нельзя уничтожить, когда работа с ним будет закончена. Его можно использовать временно, но мы всегда обязаны вернуть его владельцу до того, как владелец будет уничтожен. Заимствование сопровождается своими правилами. Как и в случае с владением, эти правила задают способ перемещения данных в программе Rust, и со временем они станут для вас второй натурой. Давайте взглянем на них:

- каждое значение в любой момент времени может иметь ровно одну мутирующую ссылку либо любое количество немутируемых ссылок;
- ссылки всегда должны быть действительными.

Первое правило может показаться немного странным разработчикам, работающим на языках, в которых нет концепции контролируемой мутации. Эта концепция будет рассмотрена подробнее в разделе 2.2.2, но сначала давайте посмотрим, как работают ссылки в более общем плане, применив их к программе о произведениях искусства из листинга 2.5. Вспомните, что в том листинге делалась попытка передать переменную в одну и ту же функцию несколько раз, но возникли трудности, поскольку передача переменной переместила ее из функции `main`. Если изменить сигнатуру функции `admire_art` из этого примера таким образом, чтобы она ссылалась не на владеемое произведение искусства, а на любое произведение искусства, то она будет работать, как положено.

Листинг 2.9 Программа, дважды любующаяся одним и тем же произведением искусства

Обратите внимание на использование амперсанда (&) в этой строке. Когда этот символ появляется в объявлении типа, например, в виде `&Artwork`, это означает, что указанный тип является ссылкой на тип, следующий за амперсандом. Следовательно, функция `admire_art` будет работать только со ссылкой на художественное произведение, но не на владеемое художественное произведение.

```
struct Artwork {  
    name: String,  
}  
  
fn admire_art(art: &Artwork) {  
    println!("Wow, {} really makes you think.", art.name);  
}
```

```
fn main() {
    let art1 = Artwork { name: "The Ordeal of Owain".to_string() };

    admire_art(&art1); ←
    admire_art(&art1);
}
```

Апмерсанд в выражении называется «оператором заимствования», при этом вычисление выражения &x дает ссылку на все, что содержится в выражении x.

Листинг 2.9 очень похож на листинг 2.5. Единственное отличие заключается в изменении типа, который принимает функция `admire_art`. Вместо того чтобы нуждаться в передаче владеемого значения типа `Artwork`, функция `admire_art` теперь принимает ссылку на значение типа `Artwork`. Если на это посмотреть с точки зрения музея, то такое поведение обретает смысл. Ведь мы не хотим создавать и уничтожать произведения искусства только ради того, чтобы ими можно было любоваться один раз; мы хотим иметь возможность делиться любованием произведениями искусства со многими людьми в разное время. Это также имеет смысл с точки зрения памяти: перегрузка памяти за счет постоянных операций создания и уничтожения значений неэффективна. Гораздо лучше использовать память повторно, когда это возможно. Если сравнить график времен жизни для листинга 2.9, то сразу станет очевидно, что в этом больше смысла. Давайте посмотрим на график времен жизни для приведенного выше примера, чтобы увидеть, как представлять подобного рода немутируемые заимствования.

На рис. 2.5 видно, что переменная `art1` больше не перемещается ни в один из вызовов функции `admire_art`. Передается ссылка, но переменная `art1` остается во владении функции `main`. Память, связанная с переменной `art1`, не высвобождается до конца функции `main`, и поскольку ссылки на нее отбрасываются по завершении вызовов их функций, это совершенно нормально.

Для того чтобы понять разницу между мутуируемыми и немутуируемыми ссылками в языке Rust, давайте посмотрим, как Rust по-разному подходит к работе с мутуируемой и немутуируемой переменными.

2.3.2 Контроль за мутуируемостью

Все переменные в языке Rust помечены дополнительной информацией, которая помогает разработчику (и компилятору Rust) понимать, как программа будет вести себя во время исполнения. Эта информация определяет мутуируемость или немутуируемость переменной, то есть можно ли ее изменять или нельзя.

Все переменные в языке Rust являются немутуируемыми, если только они не помечены как мутуируемые в явной форме при объявлении. В следующем ниже листинге показано, как выглядит объявление и использование немутуируемой и мутуируемой переменной.

Переменная `art` обведена пунктирным кругом, чтобы указать на то, что это не владеемое значение, а ссылка. Однако ссылки сами по себе являются значениями, поэтому у них по-прежнему есть времена жизни, и они отбрасываются, как обычное значение.

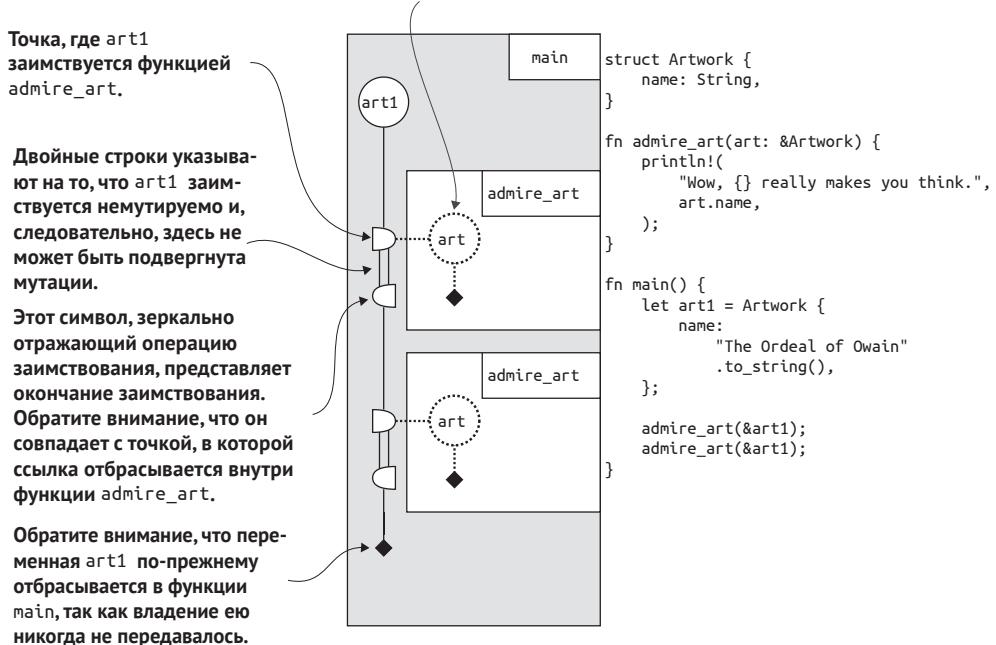


Рисунок 2.5 График времен жизни для листинга 2.9

Листинг 2.10 Использование немутуируемой и мутуируемой переменных в языке Rust

```

fn main() {
    let x = 0;
    let mut y = 0;

    println!("x={}, y={}", x, y);

    y += 10;
    println!("x={}, y={}", x, y);
}
  
```

Обявление переменной `x` не содержит аннотации, а значит, она является немутуируемой и не может быть изменена.

Ключевое слово `mut` перед именем переменной сообщает компилятору о том, что переменная `y` является мутуируемой и может быть изменена.

Поскольку требуется подвергнуть значение в у-мутации, оно должно быть объявлено как мутуируемое. Что произойдет, если заменить `y` в этой строке на `x`?

На первый взгляд может показаться странным, что Rust требует, чтобы заранее конкретизировалось, будет ли значение изменено позже или нет, но вы будете удивлены тем, как часто в большинстве исходного кода на языке Rust можно обходиться без мутаций. В дополнение к этому компилятор Rust знает о мутациях, в силу чего он может статически верифицировать исходный код, который в противном случае было бы сложно исправлять на других языках. Эта тема будет рассмотрена подробнее в главе 8, при обсуждении конкурентного исходного кода на языке Rust.

На данный момент следует запомнить, что это небольшое изменение в способе объявления переменных существует в обмен на большую выгоду от вашей способности рассуждать об исполняемом коде.

Как видно из листинга 2.10, помечать переменную как мутируемую очень просто. Придание переменной мутируемости позволяет присваивать ей значение повторно. В таком небольшом примере, возможно, будет неочевидной причина, по которой полезно иметь такой контроль над мутируемостью, но при его объединении со ссылками преимущества станут совершенно очевидными. Давайте вернемся к исходному коду художественного музея и посмотрим, как использовать в нем концепцию мутируемости.

Текущая версия функции `admire_art` поддерживает немутируемую ссылку, но что, если возникнет потребность, чтобы у каждого произведения искусства был счетчик просмотров, который увеличивается всякий раз, когда им кто-то любуется? В этом случае в функцию нужно будет внести правки, чтобы она принимала мутируемые ссылки.

Листинг 2.11 Приращение счетчика просмотров художественного произведения с использованием мутируемых ссылок

```
struct Artwork {
    view_count: i32,
    name: String,
}

fn admire_art(art: &mut Artwork) { ←
    println!("{} people have seen {} today!", ←
        art.view_count, art.name);
    art.view_count += 1; ←
}

fn main() {
    let mut art1 = Artwork { ←
        view_count: 0, name: "".to_string() }; ←
    ←
    admire_art(&mut art1); ←
    admire_art(&mut art1); ←
}
```

Изменение типа с `&Artwork` на `&mut Artwork` указывает на то, что художественное произведение может изменяться в рамках этой функции.

Для этой строки требуется мутируемая ссылка. Поскольку значение `view_count` здесь подвергнуто мутации, требуется мутируемая ссылка на владельца счетчика `view_count`, которым является содержащая его структура `Artwork`.

Несмотря на то что `art1` не подвергнута мутации внутри функции `main`, на нее создаются мутируемые ссылки, что требует аннотирования объявления ключевым словом `mut`. `&mut` создает мутируемую ссылку на `x`.

В листинге 2.11 показано, что поставленная цель увеличивать счетчик и читать его при каждом просмотре художественного произведения была достигнута. «Но подождите! – возможно, вы скажете. – Я думал, что в любой момент времени может быть только одна мутируемая ссылка на значение! Разве эта программа не нарушает данное правило?» Если немного поразмыслить над тем, что происходит в программе, то можно увидеть, что две мутируемые ссылки никогда не указывают на одно и то же значение. Этот момент проиллюстрирован на рис. 2.6.

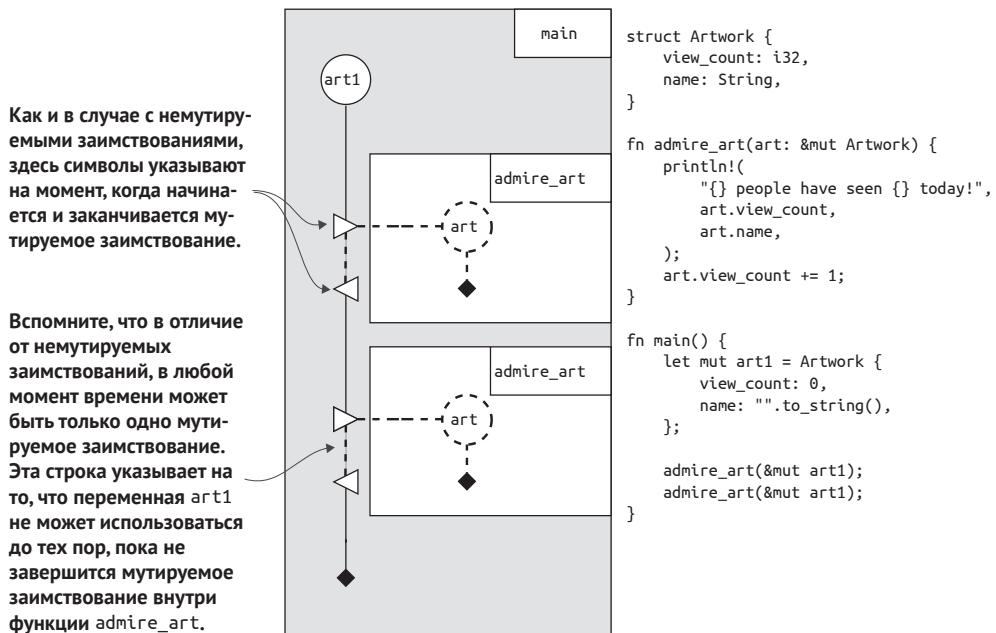


Рисунок 2.6 График времен жизни для листинга 2.11

Обратите внимание, что создаваемые ссылки имеют точки отбрасывания, после которых они больше не существуют. При вызове функции `admire_art` ей передается ссылка, а когда функция завершается, эта ссылка выходит за пределы диапазона доступности и отбрасывается. За время между двумя вызовами функции нет ни одной ссылки на `art1`. Следовательно, программа является легальным исходным кодом на языке Rust.

Возвращаясь к исходному коду в листинге 2.9, можно увидеть значение явных мутабельных аннотаций. Из объявления типа функции `admire_art` известно, что она не будет изменять передаваемое в нее значение типа `Artwork`. Почему? Потому что она принимает значение `&`, а не `&mut`. Вообще, можно смотреть на объявление функций в документации к библиотекам и знать, а не гадать, какие функции будут изменять передаваемые им значения, а какие будут иметь только немутабельный вид на такие значения. Эта структура имеет значительные перекрывающиеся последствия для целей обеспечения безопасности, производительности и отладки. Данная тема будет рассмотрена подробнее в главе 3 во время обсуждения темы интеграции исходного кода на языке Rust с языками C и C++.

2.3.3 Ссылки и времена жизни

В языке Rust у ссылок, как и у значений, есть времена жизни. Ссылки указывают на значения, но они сами также являются значениями и отбрасываются, когда выходят за пределы диапазона доступности. В до-

полнение к этому языку Rust налагает для ссылок дополнительное правило. Вспомните из первоначального обсуждения темы ссылок, что все ссылки должны быть действительными. Что это значит? Проще говоря, все ссылки должны указывать на действительные значения. Кроме того, вспомните, что времена жизни – это способ, которым компилятор Rust определяет, является значение действительным или нет. Таким образом, ссылки и времена жизни очень тесно связаны друг с другом. Ссылки не только имеют времена жизни, но и должны учитывать времена жизни значений, на которые они указывают. Давайте обследуем конкретный пример.

Листинг 2.12 Программа, пытающаяся использовать значение после его перемещения

```
struct Artwork {
    name: String,
}

fn admire_art(art: Artwork) { ←
    println!("Wow, {} really makes you think.", art.name);
}

fn main() {
    let art1 = Artwork { name: "Man on Fire".to_string() };

    let borrowed_art = &art1; ←
    admirable_art(art1);

    println!("I really enjoy {}", borrowed_art.name);
}
```

Здесь функция `admire_art` была изменена, чтобы принимать не ссылку на значение, а само значение типа `Artwork`.

`borrowed_art` – это ссылка на переменную `art1`.

При попытке выполнить этот исходный код будет получена ошибка компилятора! Давайте попробуем построить график времен жизни и посмотрим, где вкрадлась ошибка.

Как видно по рис. 2.7, программа неработоспособна, так как ссылка `borrowed_art` лишается способности ссылаться после вызова функции `admire_art`. Давайте рассмотрим еще одну распространенную ошибку, связанную с временами жизни ссылок.

Обратите внимание, что переменная `art1` перемещается в функцию `admire_art`, все еще будучи заимствованной. Таким образом, когда функция `admire_art` завершится и переменная `art` будет отброшена, ссылка `borrowed_art` не будет указывать ни на что!

Ключевым признаком того, что здесь есть проблема, является то, что время жизни ссылки `borrowed_art` предположительно превышает время жизни владеемого значения, на которое она предположительно ссылается.

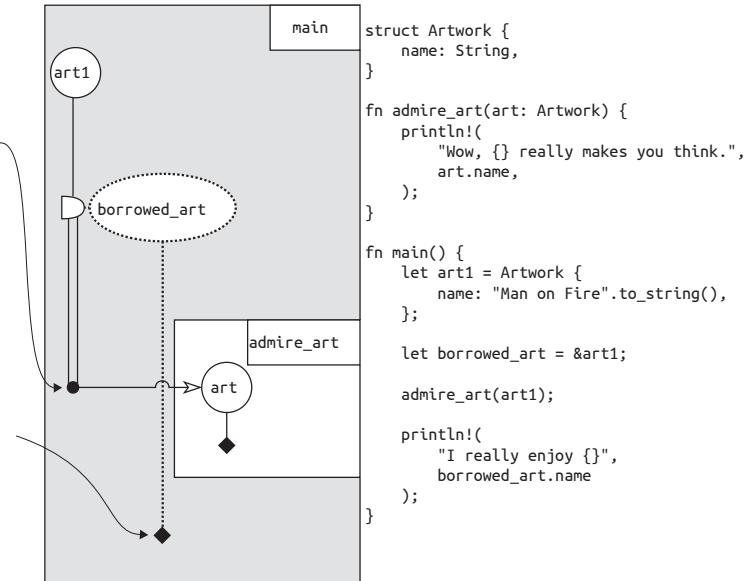


Рисунок 2.7 График времен жизни для листинга 2.12

Листинг 2.13 Функция, пытающаяся вернуть ссылку на отброшенное значение

```

struct Artwork {
    name: String,
}

fn build_art() -> &Artwork {
    let art = Artwork { name: "La Liberté guidant
        le peuple".to_string() };

    &art
}

fn main() {
    let art = build_art();
}
  
```

Использование ключевого слова `return` в языке Rust не является обязательным. Если в конце строки нет точки с запятой, то в качестве возвращаемого значения используется последнее выражение функции.

Функция `build_art` в листинге 2.13 неработоспособна по слегка иной причине. Переменная `art` никогда не перемещается; однако делается попытка вернуть ссылку на нее, даже если она отбрасывается в конце исполнения функции. Давайте посмотрим на график времен жизни для этой программы на рис. 2.8.

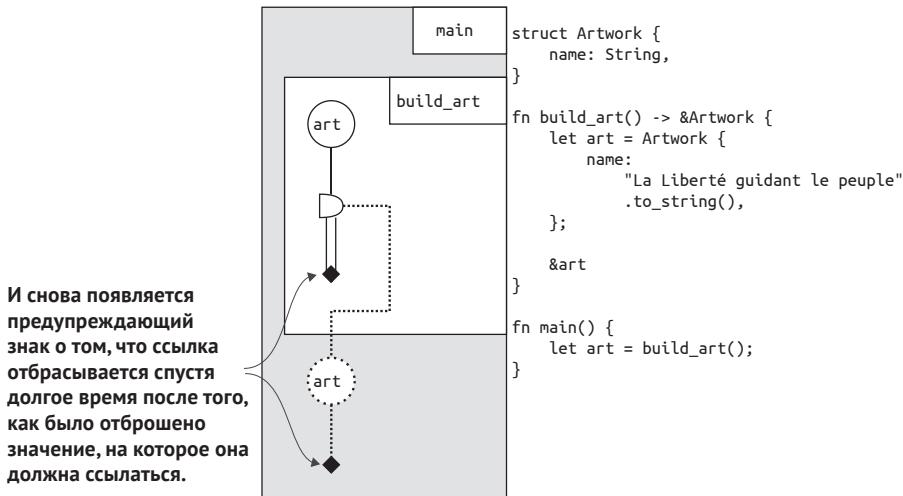


Рисунок 2.8 График времен жизни для листинга 2.13

График времен жизни на рис. 2.8 показывает тот же общий предупреждающий знак, что и график на рис. 2.7. Ссылка выходит за пределы точки отбрасывания значения, на которое она должна ссылаться. На языке Rust можно писать функции, которые возвращают ссылки, но эти функции на входе обычно тоже принимают ссылки. Если функция возвращает ссылку, но не имеет параметров или принимает только владеемые параметры, то это обычно является признаком того, что при ее компиляции будет показана ошибка времени жизни.

2.4 Строковые типы в языке Rust

Почти каждый язык программирования в той или иной степени поддерживает операции на строковых литералах. Их полезность общеизвестна, как же иначе? Во многих языках программирования есть тип `String`, но Rust немного стоит особняком: в нем для представления строковых литералов применяется несколько типов. Наиболее распространенными типами являются `String` и `&str`. Давайте посмотрим, как они оба используются.

Тип `&str`, также именуемый ссылкой на строковый литерал, является более простым из двух типов и состоит только из указателя на начальную позицию в памяти и длины. Из-за своей простоты тип `&str` является более гибким из двух, поскольку ссылка может указывать на любые строковые данные в любом месте памяти. Он может подкрепляться резервируемым в стеке буфером в виде массива, значением типа `String` или даже строковым литералом, скомпилированным и встроенным в сам двоичный файл программы. Если вы работаете с языком C или C++, то, возможно, знаете, что строковые данные в них слегка отличаются от других строковых значений, даже если они имеют одинаковые типы. Строковые литералы в C и C++ предна-

значены только для чтения, поскольку они компилируются в двоичный формат и хранятся в памяти, которую можно только читать. Если попытаться выполнить эту программу на C, то, скорее всего, будет получена ошибка сегментации (ошибка незаконного доступа к памяти во время исполнения).

Листинг 2.14 Программа на языке C, пытающаяся выполнить запись в память, которую можно только читать

```
int main(void) {
    char *str = "hello, world!";
    str[0] = '!';    ← Эта строка вызывает
                     ошибку сегментации.
    return 0;
}
```

Исходный код в листинге 2.14 неработоспособен, поскольку он пытается писать данные в область, которую можно только читать. Компилятор C не знает, что `str` указывает на память, которую можно только читать, поскольку типы C не предоставляют никакой информации о том, могут значения подвергаться мутации или нет. Эквивалентным типом для строковых литералов в языке Rust является `&'static str`. Здесь новый синтаксис, «статическая часть», представляет собой *аннотацию* времени жизни. Указанный маркер для компилятора явно указывает на то, как долго эта ссылка будет действительной. Указанная тема будет более подробно рассмотрена в главе 4, но на данный момент важно запомнить, что `&'static` «и что-то еще» означает, что ссылка будет жить в течение всего времени исполнения программы. Поскольку строковые литералы компилируются и встраиваются в двоичный файл, статические ссылки `&'static str` могут ссылаться на них в любой момент, не беспокоясь о том, были ли они отброшены (поскольку они не могут быть отброшены). В языке Rust также разрешено иметь нестатическую ссылку на строковый литерал. Давайте посмотрим, как она выглядит.

Листинг 2.15 Пример нестатической ссылки

```
struct Artwork {
    name: &'static str,
}

fn admire_art(art: &Artwork) {
    print_admiration(art.name);    ← При передаче &'static str в функцию,
                                    которая принимает &str в качестве
                                    аргумента, ссылка &'static конверти-
                                    руется в &.
}

fn print_admiration(name: &str) {
    println!("Wow, {} really makes you think.", name);
}

fn main() {
```

```

let art1 = Artwork { name: "The Ordeal of Owain" };
    ↪
admire_art(&art1);
}

```

Больше не нужно вызывать метод `to_string()`, потому что ожидаемым типом для `name` будет не `String`, а `&str`.

Важен тот факт, что ссылки на строковые литералы являются немутуируемыми. Поскольку они всего лишь указывают на буферы памяти, не имея понятия о том, как эти буферы сконструированы или какой дополнительной емкостью они могут обладать, их невозможно изменить. Если возникает потребность изменять строковые значения, то нужно обратиться к другому типу строковых литералов в языке Rust, типу `String`.

2.4.1 Мутируемые строковые литералы

Если вы знакомы с такими языками программирования, как Java, JavaScript или Python, то, возможно, впервые слышите о мутуемости в контексте строковых литералов. В этих и многих других подобных им языках все строковые литералы не способны муттировать; их нельзя изменить после их создания. Возможно, вы говорите себе, что часто меняете значения строковых литералов, используя операции `+=` в этих языках для конкатенации одного строкового литерала с другим, однако вы не совсем правы. В языках с немутуируемыми строковыми литералами невозможно редактировать память строкового литерала после его создания; правка строкового литерала возможна только путем создания нового строкового литерала, содержащего новое запрошенное содержимое.

Представьте себе, что нужно создать программу, которая добавляет символ точки `"."` в строковый литерал при каждом выполнении какого-либо действия, которое будет аппроксимироваться циклом `for` из 10 миллионов итераций.

Листинг 2.16 Создание очень крупного строкового литерала по одному символу за раз на языке Python

```

x = ""

for _ in range(0, 10_000_000):
    x += "."

```

```

print(len(x))

```

При каждой итерации цикла `for` в листинге 2.16 он создает новый строковый литерал, содержащий копию всех текущих строковых данных плюс один символ точки. Следовательно, для того чтобы создать строковый литерал из 10 млн точек, программе необходимо выполнить 10 млн резервирований памяти, в результате чего будет скопировано 9 999 999 ненужных строковых литералов. Процесс копирования памяти в более крупную область хранения называется *перераспределением памяти*. Давайте сравним этот процесс с Rust, который предоставляет разработчику возможность подвергать строковые литералы мутации.

В языке Rust значение типа `String`, или владеемый строковый литерал, состоит из резервируемого в куче динамически расширяемого буфера, в котором хранятся символьные данные. Если требуется добавлять дополнительные символы в конец строкового литерала, то их можно добавлять в конец буфера. Если требуется менять местами символы в середине, то их можно перемещать там же в середине. Эти буфера имеют как длину, так и емкость. Длина представляет собой количество действительных элементов в буфере, а емкость – количество элементов, которые буфер может содержать при его заполнении. Единственный раз, когда в языке Rust значения типа `String` нуждаются в дополнительном резервировании памяти и копировании, как в Python, – это когда изменение строкового литерала может приводить к превышению длины буфера над его емкостью. В таких случаях память под буфер будет перераспределяться емкостью, по меньшей мере такой, какая потребуется для хранения новых данных. Стандартная библиотека Rust не гарантирует какой-либо конкретной стратегии увеличения емкости буфера, но возможно, что при вставке одного символа в строковый литерал емкость буфера, например, удвоится, так что будущие вставки символов не потребуют перераспределения памяти.

Давайте посмотрим, как использовать строковый литерал для имитации функциональности, описанной в листинге 2.16.

Листинг 2.17 Создание крупного строкового литерала по одному символу за раз на языке Rust

```
fn main() {
    let mut x = String::new(); ←
    for _ in 0..10_000_000 {
        x.push('.');
    }
    println!("{}", x.len());
}
```

`String::new` создает новый строковый литерал с буфером, емкость которого равна нулю. Эта функция не выполняет никаких резервирований памяти.

Как видно из листинга 2.17, от разработчика скрыта большая часть работы с буфером. Обычно единственное, что можно делать с ним напрямую, – это устанавливать его емкость равной некоторому заранее определенному размеру, чтобы пытаться ограничивать количество резервирований памяти, выполняемых исходным кодом. Если требуется делать как можно меньше резервирований, чтобы обеспечивать максимально быстрое время исполнения программы, то можно воспользоваться функцией `String::with_capacity`, чтобы заранее настраивать емкость явным образом. Благодаря этому программа с 10 млн точек могла бы работать только с одним резервированием памяти! Если вы работаете с крупными строковыми литералами, то эта возможность дает значительный прирост производительности.

В следующем ниже листинге показано, как использовать функцию `with_capacity`.

Листинг 2.18 Перераспределение памяти под строковые литералы с целью повышения производительности

```
fn main() {
    let mut x = String::with_capacity(10_000_000);
    for i in 0..10_000_000 {
        x.push('.');
    }
    println!("{}", x.len());
}
```

Эта строка – единственная, которую нужно было изменить, чтобы перейти к одному резервированию памяти. Исходный код использования строкового литерала остается прежним.

Функция `String::with_capacity` оптимизирует производительность. Возвращаемые ею значения типа `String` можно использовать так же, как и строковые литералы из `String::new`, но в некоторых случаях они могут работать лучше. Используя `push`, можно безопасно увеличивать размер строкового литерала, выходя за пределы емкости; на внутреннем уровне строковый литерал будет перераспределять память под свой буфер.

Возможно, вам будет интересно узнать, как конвертировать два разных строковых типа в языке Rust, поэтому давайте рассмотрим, как это делать. Оба способа конвертации просты для разработчика, но один из них намного дороже для компьютера во время исполнения. Конвертация значения типа `String` в значение типа `&str` обходится очень дешево. Поскольку значения типа `&str` – это просто указатель и длина, достаточно скопировать начальный указатель буфера значений типа `String` и его длину. Это всего лишь два 64-битовых целых числа, которые можно копировать на большинстве компьютеров, что очень недорого. В следующем ниже листинге это показано.

Листинг 2.19 Конвертация значения типа `String` в ссылку на строковый литерал

```
fn print_admiration(name: &str) {
    println!("Wow, {} really makes you think.", name);
}

fn main() {
    let value = String::new();

    print_admiration(value.as_str());
}
```

Двигаясь в обратную сторону, конвертация ссылки типа `&str` в строковый литерал типа `String` обходится компьютеру немного дороже. Поскольку все значения типа `String` имеют свой собственный резервируемый в куче буфер, для создания значения типа `String` из ссылки типа `&str` требуется, чтобы компьютер зарезервировал память под буфер, по меньшей мере достаточно большой, чтобы вместить все данные из `&str`, а затем скопировал все данные из `&str` в только что созданный буфер.

Если это выполнять в плотном цикле, то в результате может снизиться производительность. Положительный момент состоит в том, что в большинстве случаев легко увидеть, где происходит конвертация, и ее ограничить. В этой главе такая конвертация уже выполнялась; это достигается путем вызова функции `.to_string()` для значений типа `&str`.

Листинг 2.20 Конвертация ссылки на строковый литерал в значение типа String

```
fn print_admiration(name: String) {  
    println!("Wow, {} really makes you think.", name);  
}  
  
fn main() {  
    let value = "Artwork";  
  
    print_admiration(value.to_string());  
}
```

В языке Rust принято как идиома использовать похожие методы с префиксами `as_` и `to_`. Метод `as_` обычно означает получение дешевой ссылки на что-либо, а `to_` указывает на резервирование памяти под владеемую структуру данных и копирование туда.

Как и большинство материалов в данной главе, эти разные типы строковых литералов окажутся полезными в долгосрочной перспективе, но могут приводить к путанице в краткосрочной перспективе. Знание ситуаций, когда следует использовать разные типы строковых литералов, приходит с опытом; на данный момент можно обобщить. Если данные хранятся в структуре, которая будет жить в течение длительного времени, то, вероятно, следует использовать тип `String`, а если в функцию передаются данные с доступом только для чтения, то, вероятно, следует использовать тип `&str`. Если нет уверенности в том, какой из них использовать, то тип `String` является более гибким вариантом, и дополнительные затраты, возникающие при создании значений типа `String` из ссылок на строковые литералы, могут быть устраниены позже. Теперь давайте перейдем к последней области, в которой Rust значительно отличается от других языков программирования, – обработке ошибок.

2.5 Перечисления и обработка ошибок

Во многих языках программирования исключения используются для того, чтобы проталкивать ошибки вверх по стеку от места их возникновения до какого-нибудь исходного кода обработки. Язык Rust отличается от этих языков. Ошибки – это обычные значения, обрабатываемые с помощью обычных элементов потока управления, которые не являются специфичными для ошибок. Сначала давайте применим пример, чтобы пошагово пройтись по простому варианту использования перечислений вне контекста ошибок. Вы познакомитесь с обработкой ошибок после того, как у вас будет четкое понимание.

2.5.1 Перечисления

FizzBuzz – это популярная задача на программирование, предназначенная для проверки способности кандидата использовать базовые элементы потока управления, такие как циклы и инструкции `if`. Она заключается в следующем: написать программу, которая считает от 1 до 100. Всякий раз, когда достигается число, кратное 3, печатать слово `"fizz"`. Всякий раз, когда достигается число, кратное 5, печатать слово `"buzz"`. Если число делится на 3 и на 5, то печатать `"fizzbuzz"`. В противном случае печатать само число. Далее будет реализовано решение задачи FizzBuzz с использованием одной внешней функции, чтобы выполнять цикл и печать, а также вспомогательной функции, чтобы выполнять проверку делимости. Вспомогательная функция должна возвращать перечисление, указывающее функции `main`, что делать.

Давайте начнем с написания функции `main`, которая будет выполнять цикл и печать чисел.

Листинг 2.21 Функция, которая выполняет итерации по числам в интервале от 1 до 100

```
fn main() {
    for i in 1..101 {           ←
        println!("{}", i);
    }
}
```

Этот цикл `for` будет выполнять итерации по числам в интервале от 1 до 100. Диапазонный синтаксис `x..y` имеет включающую нижнюю границу и исключающую верхнюю границу.

Далее сначала давайте сделаем первые наброски вспомогательной функции, которая выполняет проверку делимости входного значения.

Листинг 2.22 Программа FizzBuzz со вспомогательной функцией

```
fn main() {
    for i in 1..101 {
        print_fizzbuzz(i);
    }
}

fn print_fizzbuzz(x: i32) {           ←
    println!("{}", fizzbuzz(x));
}

fn fizzbuzz(x: i32) -> String {
    if x % 3 == 0 && x % 5 == 0 {
        String::from("FizzBuzz")
    } else if x % 3 == 0 {
        String::from("Fizz")
    } else if x % 5 == 0 {
        String::from("Buzz")
    } else {
        format!("{}", x)           ←
    }
}
```

Функции `print_fizzbuzz` и `fizzbuzz` отделяют вычисление результата от представления этого результата пользователю. Преимущества такого подхода станут более очевидными по мере продвижения вперед.

`format!` – это макрокоманда, в которой используется тот же синтаксис, что и в макрокоманде `println!`, но вместо печати результата в `STDOUT` она возвращает значение типа `String`.

Хотя этот исходный код и решает задачу FizzBuzz, в нем есть возможности для улучшения. В крупной системе не желательно раздавать строковые литералы повсюду, чтобы сообщать о состоянии. Rust – язык со строгой типизацией, и следует воспользоваться ее преимуществами, чтобы обеспечить правильность обработки значений, возвращаемых из функции `fizzbuzz`. Что, если потребуется использовать ту же проверку на делимость, но показывать результаты по-другому? Например, может возникнуть потребность отправлять результат по какому-либо сетевому потоку компактным способом. В этом случае пришлось бы выполнять разбор строковых литералов "`Fizz`" / "`Buzz`" / "`FizzBuzz`", а также извлекать цифры из строковых литералов. Это можно сделать лучше.

Для организации надлежащего обмена данными между функциями `print_fizzbuzz` и `fizzbuzz` используется перечисление. Перечисления – это типы, которые могут иметь ровно одно из заранее заданных возможных чисел. Поскольку функция `fizzbuzz` имеет четыре возможных возвращаемых значения ("`fizz`", "`buzz`", "`fizzbuzz`" или что-то, указывающее на неделимость), это идеальный вариант использования. Перечисления существуют во многих языках программирования, но они лежат в основе Rust. Позже в этом разделе вы увидите, как перечисления используются в языке Rust для обработки ошибок, а пока будем придерживаться программы FizzBuzz. Давайте напишем перечисление, которое позволит вспомогательной функции сообщать разные результаты ее работы обратно в функцию `print_fizbuzz`. В следующем ниже листинге показано, как выглядит это перечисление.

Листинг 2.23 Перечисление, содержащее результаты работы функции `fizzbuzz`

```
enum FizzBuzzValue {  
    Fizz,  
    Buzz,  
    FizzBuzz,  
    NotDivisible,  
}
```

Каждая запись в списке возможных состояний в перечислении называется *вариантом*. Вы видите, что в перечислении `FizzBuzzValue` представлены все возможные возвращаемые значения. Теперь давайте посмотрим, как его использовать в функции `fizzbuzz`.

Листинг 2.24 Возвращение перечисления из функции

```
enum FizzBuzzValue {  
    Fizz,  
    Buzz,  
    FizzBuzz,  
    NotDivisible,  
}
```

```
fn fizzbuzz(x: i32) -> FizzBuzzValue {
    if x % 3 == 0 && x % 5 == 0 {
        FizzBuzzValue::FizzBuzz
    } else if x % 3 == 0 {
        FizzBuzzValue::Fizz
    } else if x % 5 == 0 {
        FizzBuzzValue::Buzz
    } else {
        FizzBuzzValue::NotDivisible
}
```

Теперь, если нужно использовать возвращаемое из функции `fizzbuzz` значение для печати сообщения, то можно воспользоваться выражением `match`. Данное выражение похоже на инструкции `switch` в языках Java, C, C++ и Go, но у него есть некоторая дополнительная функциональность, которая будет рассмотрена чуть позже.

Листинг 2.25 Использование выражения match с вариантами перечисления

```
enum FizzBuzzValue {
    Fizz,
    Buzz,
    FizzBuzz,
    NotDivisible,
}

fn main() {
    for i in 1..101 {
        print_fizzbuzz(i);
    }
}

fn print_fizzbuzz(x: i32) {
    match fizzbuzz(x) {
        FizzBuzzValue::FizzBuzz => { ←
            println!("FizzBuzz");
        }
        FizzBuzzValue::Fizz => {
            println!("Fizz");
        }
        FizzBuzzValue::Buzz => {
            println!("Buzz");
        }
        FizzBuzzValue::NotDivisible => {
            println!("{}", x);
        }
    }
}

fn fizzbuzz(x: i32) -> FizzBuzzValue {
    if x % 3 == 0 && x % 5 == 0 {
```

Каждая ветвь, или рукав, выражения `match` содержит условие, символ «большая стрелка» (`=>`), а затем выражение, которое будет вычислено, если это условие является истинным.

```

        FizzBuzzValue::FizzBuzz
    } else if x % 3 == 0 {
        FizzBuzzValue::Fizz
    } else if x % 5 == 0 {
        FizzBuzzValue::Buzz
    } else {
        FizzBuzzValue::NotDivisible
    }
}

```

Принятый подход, по-видимому, работает хорошо. Вычисление результатов было эффективно обособлено от представления этих результатов пользователю. Подобное разделение в таком маленьком примере может показаться странным, если учитывать, что, безусловно, потребовалось бы меньше кода, удалив его или даже поместив вызовы макрокоманд `println!` внутрь функции `fizzbuzz`, но в более крупных программах перечисления очень полезны в создании единого стандартизированного способа представления значений, которые во время исполнения могут иметь несколько вариантов.

В этом небольшом примере перечисление `FizzBuzzValue` работает достаточно хорошо, но у него есть недостаток, который может проявиться в более крупных программах. Последний вариант в перечислении, `NotDivisible`, содержит дополнительный фрагмент данных, который должен быть с ним связан, но исходный код его не улавливает, а именно входное число, которое не делится на 3 или 5. Если желательно печатать этот результат в программе где-нибудь еще, то нужно придумать способ сохранять число и информацию `NotDivisible`. Перечисления в языке Rust делают это дополнительное хранилище чрезвычайно прямолинейным образом. Каждый вариант перечисления, помимо данных о том, к какому варианту он относится, может содержать любое количество дополнительных полей данных. Давайте посмотрим пример того, как это может выглядеть.

Листинг 2.26 Перечисление `FizzBuzzValue`, содержащее число, не делящееся на 3 или 5

```

enum FizzBuzzValue {
    Fizz,
    Buzz,
    FizzBuzz,
    NotDivisible(i32), ←
}

```

Этот аргумент типа `i32` указывает на то, что с вариантом `NotDivisible` всегда будет связано значение типа `i32`.

```

fn main() {
    for i in 1..101 {
        print_fizzbuzz(i);
    }
}

fn print_fizzbuzz(x: i32) {

```

```

match fizzbuzz(x) {
    FizzBuzzValue::FizzBuzz => {
        println!("FizzBuzz");
    }
    FizzBuzzValue::Fizz => {
        println!("Fizz");
    }
    FizzBuzzValue::Buzz => {
        println!("Buzz");
    }
    FizzBuzzValue::NotDivisible(num) => { ←
        println!("{} ", num);
    }
}

fn fizzbuzz(x: i32) -> FizzBuzzValue {
    if x % 3 == 0 && x % 5 == 0 {
        FizzBuzzValue::FizzBuzz
    } else if x % 3 == 0 {
        FizzBuzzValue::Fizz
    } else if x % 5 == 0 {
        FizzBuzzValue::Buzz
    } else { ←
        FizzBuzzValue::NotDivisible(x) ←
    }
}

```

Переменной `num` присваивается значение из `i32`, которое хранится в варианте `NotDivisible` перечисления.

Здесь значение числа `x` помещается в вариант `NotDivisible` перечисления.

Заключительная ветвь выражения `match` слегка изменилась. Теперь была добавлена переменная `num`, которая получает свое значение из `i32`, хранящегося в варианте `NotDivisible`. Такое удаление значений из контейнерных типов, таких как варианты перечислений, называется *деструктуризацией*. Известно, что каждый вариант `NotDivisible` будет содержать `i32`, потому что этого требует объявление перечисления. При таком объявлении перечисления невозможно создать `NotDivisible` без указания `i32`. Более того, невозможно получить доступ к `i32` в варианте `NotDivisible` без какой-нибудь проверки, обеспечивающей, чтобы `FizzBuzzValue` содержало значение `NotDivisible`.

Теперь, когда есть понимание того, как использовать перечисления и выражения `match`, давайте посмотрим, как их применять для обработки ошибок.

2.5.2 Обработка ошибок с помощью перечислений

Многие языки программирования представляют ошибки в виде исключений, и у них есть методы, чтобы сообщать об исключительных условиях в программах. Исключения «всплывают» в стеке до тех пор, пока не столкнутся с каким-либо специальным исходным кодом обработки ошибок, таким как блок `try/except`. В языке Rust ошибки представлены так же, как и обычные значения, и в них используются те же элементы потока управления, что и в обычных значениях. В данном разделе будут продемонстри-

рованы способы написания функций, которые могут отказывать во время исполнения, и обработки ошибок, возникающих в этих функциях.

Представьте себе, что к функции `fizzbuzz` предъявлено новое требование. Теперь, в дополнение к ее функциональности, определяющей делимость, функция должна возвращать ошибку, если указанное число является отрицательным. В разрабатываемой программе значения, которые будут передаваться функции `fizzbuzz`, известны, потому что они, конечно же, набраны непосредственно в исходном коде. Однако представьте на мгновение, что они получены от пользователя. Эти ошибки нужно уметь обрабатывать иным образом, чем обычные значения перечисления, возвращаемые функцией, и перечисление `FizzBuzzValue` не должно расширяться, чтобы учитывать возможное состояние ошибки.

Давайте посмотрим, как бы представить это возможное условие отказа в программе. Стандартная библиотека языка Rust имеет в своем составе тип под названием `Result`, который содержит указание на успешное вычисление и выходные данные этого вычисления либо указание на ошибку и более подробную информацию об этой ошибке. В следующем ниже листинге показано объявление этого перечисления.

Листинг 2.27 Определение типа Result

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Как и вариант `Ok`, вариант `Err` может содержать значение любого типа.

Синтаксис `<T, E>` создает две обобщенные переменные, или переменные типа `T` и `E`. `T` относится к переменному типу `T`, созданному в первой строке. Он указывает на то, что вариант `Ok` может содержать значение абсолютно любого типа.

Тип `Result` является одним из наиболее часто используемых типов в исходном коде на языке Rust, потому что любая функция, которая может отказать, возвращает значение, обернутое в тип `Result`. Давайте вернемся к программе, чтобы увидеть, что нужно изменить, если функция `fizzbuzz` может вернуть ошибку.

Листинг 2.28 Функция fizzbuzz, которая может возвращать ошибку

```
enum FizzBuzzValue {
    Fizz,
    Buzz,
    FizzBuzz,
    NotDivisible(i32),
}

fn main() {
    for i in 1..101 {
        match print_fizzbuzz(i) {
            Ok(_) => {}
            Err(e) => {
                eprintln!("Error: {}", e);
            }
        }
    }
}
```

Макрокоманда `eprintln!` работает так же, как и макрокоманда `println!`, но печатает свое сообщение не в `STDOUT`, а в `STDERR`. Она обычно используется для показа сообщений об ошибках, так как они не будут мешать нормальному выводу результатов работы программы, направляемому в `STDOUT`.

```

        return;
    }
}
}

fn print_fizzbuzz(x: i32) -> Result<(), &'static str> {
    match fizzbuzz(x) {
        Ok(result) => {
            match result {
                FizzBuzzValue::FizzBuzz => {
                    println!("FizzBuzz");
                }
                FizzBuzzValue::Fizz => {
                    println!("Fizz");
                }
                FizzBuzzValue::Buzz => {
                    println!("Buzz");
                }
                FizzBuzzValue::NotDivisible(num) => {
                    println!("{} {}", num);
                }
            }
        }
        Ok(())
    }
    Err(e) => {
        Err(e)
    }
}
}

fn fizzbuzz(x: i32) -> Result<FizzBuzzValue, &'static str> {
    if x < 0 {
        Err("Provided number must be positive!")
    } else if x % 3 == 0 && x % 5 == 0 {
        Ok(FizzBuzzValue::FizzBuzz)
    } else if x % 3 == 0 {
        Ok(FizzBuzzValue::Fizz)
    } else if x % 5 == 0 {
        Ok(FizzBuzzValue::Buzz)
    } else {
        Ok(FizzBuzzValue::NotDivisible(x))
    }
}

```

Тип успешности представлен единичным типом, то есть () (см. следующий раздел). Для варианта Err указан тип &'static str. Для такого рода простого обмена сообщениями об ошибках иногда используются типы String и &'static str.

Как и в случае с вариантом NotDivisible, получить доступ к перечислению FizzBuzzValue внутри объекта Result невозможно, если нет выражения match, обеспечивающего, чтобы возвращенное из fizzbuzz значение было успешным или Ok.

Все пути исполнения кода в этой функции больше не возвращают просто перечисление FizzBuzzValue; теперь они должны заключать значения перечисления FizzBuzzValue в Ok, чтобы указывать на то, что вычисление было успешным.

В приведенном выше исходном коде появилось несколько новых важных изменений. Первое и наиболее очевидное – это введение значений типа `Result` в типы значений, возвращаемых из функций `print_fizzbuzz` и `fizzbuzz`. Обе функции теперь возвращают значения типа `Result` с одинаковым типом ошибки (`&'static str`), но у них разные типы для `Ok`. Функция `fizzbuzz` возвращает то же перечисление

`FizzBuzzValue`, что и раньше, но что из себя представляет () в типе значения, возвращаемого из функции `print_fizzbuzz?` Это единичный тип, и он будет рассмотрен прямо сейчас.

2.5.3 Единичный тип

Единичный тип – это тип, единственным возможным значением которого является он сам, и он не может содержать никакой информации. Он представляет собой концепцию «ничего». Он аналогичен `null` в других языках программирования, но с очень важным отличием. В большинстве языков программирования, содержащих значение `null`, оно является действительным значением для любого ссылочного типа. Например, следующий ниже исходный код на языке Java компилируется и исполняется, печатая `null` на консоли.

Листинг 2.29 null в Java

```
public class Main {
    public static void main(String[] args) {
        String x = null;
        System.out.println(x);
    }
}
```

Приведенный выше исходный код работает, потому что Java и многие другие языки позволяют присваивать значение `null` всем ссылочным типам. Такая возможность влечет за собой большое количество ошибок во время исполнения, когда разработчики забывают проверить, содержит ссылка значение `null` или нет. Давайте попробуем написать тот же исходный код на языке Rust.

Листинг 2.30 Единичный тип в языке Rust

```
fn main() {
    let x: String = ();
    println!("{}", x);
}
```

Если попытаться исполнить этот исходный код, то будет обнаружено, что он не компилируется. Компилятор Rust выдает сообщение об ошибке, объясняющее, что фактический тип () не соответствует ожидаемому типу `String`:

```
$ cargo run
error[E0308]: mismatched types
--> src/main.rs:2:19
  |
2 |     let x: String = ();
  |                 ^^^ expected struct `String`, found `()`
  |                 |
  |                 expected due to this

error: aborting due to previous error
```

Он не компилируется, потому что единичный тип – это самостоятельный тип, полностью независимый от всех других типов. Лучшим аналогом единичного типа, чем `null`, является `void`. Возможно, вы заметили, что метод `main` в исходном коде на языке Java из листинга 2.29 возвращает тип `void`. Указанный тип представляет «ничего» на уровне типов Java. В отличие от единичного типа в языке Rust, значение типа `void` в языке Java невозможно сохранить. При написании исходного кода на языке Rust вы также, возможно, заметили, что если функции не возвращают значение, то типы возвращаемых из них значений не аннотируются, и это не потому, что они не возвращают значение, а потому, что все неаннотированные функции возвращают единичный тип. Три функции в следующем ниже листинге эквивалентны.

Листинг 2.31 Три функции, которые возвращают единичный тип

```
fn foo() {           ←
    println!("Hello!");
}

fn bar() -> () {   ←
    println!("Hello!");
}

fn baz() -> () {   ←
    println!("Hello!");
    ()               ←
}
```

Функции, которые не возвращают значений, обычно пишутся как неаннотированные функции. Обратите внимание, что эта функция по-прежнему возвращает единичный тип, но неявно.

Эта функция вводит явную аннотацию для единичного типа в качестве типа возвращаемого из функции значения.

В дополнение к аннотации типа возвращаемого значения эта функция включает в себя явно заданное возвращение значения единичного типа.

Все три функции печатают «Hello» и завершают работу, возвращая значение единичного типа. Единственное различие заключается в том, что последние две являются более явными. Функция `bar` похожа на то, как могла бы быть написана функция `void` на другом языке – явная аннотация типа возвращаемого значения, но неявное возвращение самого значения.

Давайте вернемся к функции `print_fizzbuzz` из листинга 2.28. Ее объявление имеет следующую ниже сигнатуру.

```
fn print_fizzbuzz(x: i32) -> Result<(), &'static str>
```

Возвращаемое значение типа `Result` имеет единичный тип в позиции `Ok`, в силу чего при построении варианта `Ok` он всегда будет содержать значение, не дающее никакой дополнительной информации. Если задуматься о том, что функция делает, то это имеет смысл. Если функция завершается успешно, то какое значение она, возможно, должна будет предоставить источнику вызова, кроме указания на то, что она завершилась успешно? Поскольку случай успешного выполнения функции не содержит никакой значимой дополнительной информации, при успешном выполнении функции возвращается единичный тип. Значения единичного типа, как правило, сами по себе бесполезны; просто в данном случае его нужно использовать, потому что тип `Result` нуждается в том, чтобы был указан тип для вариантов `Ok` и `Err`,

а () является наиболее разумным типом для варианта Ok функции, которой не нужно отправлять обратно какие-либо другие значения. До того, как был добавлен результат, тип значения, возвращаемого из функции `print_fizzbuzz`, на самом деле был (); он просто был не явно заданным, как сейчас, а неявным.

Давайте вернемся к исходному коду программы FizzBuzz и завершим рассмотрение обработки ошибок введением конкретно-прикладного типа ошибки.

2.5.4 Типы ошибки

Будучи разработчиками, мы знаем типы ошибок, с которыми исходный код обычно сталкивается во время своей работы; это могут быть ошибки ввода-вывода, сетевые ошибки, отказы предварительных условий, отсутствие данных и т.д. В большинстве программ на языке Rust создаются конкретно-прикладные типы, перечисляющие ошибки, которые могут быть возвращены, чтобы каждая из них могла быть обработана по-своему. После обнаружения сетевой ошибки желательно повторить запрос, тогда как ошибка наподобие отсутствия файла, вероятно, должна быть зарегистрирована, и программа должна продолжить работу, если это возможно, или прерваться, если нет. Поскольку желательно представлять различные варианты ошибок в одном типе, будет создано перечисление. Ввиду того, что в программе FizzBuzz есть только одна возможная ошибка, которая возвращается, когда функция `fizzbuzz` получает отрицательное число, давайте посмотрим, как это может выглядеть.

Листинг 2.32 Тип ошибки для программы FizzBuzz

```
enum Errort {
    GotNegative,
}
```

По традиции используется имя `Errort`, но на самом деле ошибку можно назвать как угодно; следует учитывать, что это всего лишь обычный тип. В программе, которая выполняет больше операций, на типе ошибки может быть определено много разных вариантов или могут быть определены варианты, которые обертивают типы ошибок из других библиотек. Теперь, когда есть тип `Errort`, давайте добавим его в исходный код.

Листинг 2.33 Программа FizzBuzz с конкретно-прикладным типом ошибки

```
enum FizzBuzzValue {
    Fizz,
    Buzz,
    FizzBuzz,
    NotDivisible(i32),
}

enum Errort {
    GotNegative,
}
```

```

fn main() {
    for i in 1..101 {
        match print_fizzbuzz(i) {
            Ok(_) => {}
            Err(e) => {
                match e {
                    Error::GotNegative => {
                        eprintln!("Error: Fizz Buzz only
                                  supports positive numbers!");
                        return;
                    }
                }
            }
        }
    }
}

fn print_fizzbuzz(x: i32) -> Result<(), Error> {
    match fizzbuzz(x) {
        Ok(result) => {
            match result {
                FizzBuzzValue::FizzBuzz => {
                    println!("FizzBuzz");
                }
                FizzBuzzValue::Fizz => {
                    println!("Fizz");
                }
                FizzBuzzValue::Buzz => {
                    println!("Buzz");
                }
                FizzBuzzValue::NotDivisible(num) => {
                    println!("{} {}", num);
                }
            }
            Ok(())
        }
        Err(e) => {
            Err(e)
        }
    }
}

fn fizzbuzz(x: i32) -> Result<FizzBuzzValue, Error> {
    if x < 0 {
        Err(Error::GotNegative)
    } else if x % 3 == 0 && x % 5 == 0 {
        Ok(FizzBuzzValue::FizzBuzz)
    } else if x % 3 == 0 {
        Ok(FizzBuzzValue::Fizz)
    } else if x % 5 == 0 {
        Ok(FizzBuzzValue::Buzz)
    } else {
        Ok(FizzBuzzValue::NotDivisible(x))
    }
}

```

```

    } else {
        Ok(FizzBuzzValue::NotDivisible(x))
    }
}

```

Хорошо видно, что включение конкретно-прикладного типа ошибки не сильно отличается от того, как исходный код выглядел раньше. Изменились некоторые типы возвращаемых значений, и пришлось обновить то, что делалось с ошибкой в функции `print_fizzbuzz`, так как ее больше нельзя печатать напрямую.

Теперь давайте посмотрим, как упростить обработку ошибок в функции `print_fizzbuzz`. Прямо сейчас она возвращает непосредственно источнику вызова любую ошибку, которую видит. Она не выполняет никакой проверки ошибки, кроме как сообщает «Ошибка или нет?». Этот способ обработки ошибок очень распространен в функциях Rust. Если какая-то функция возвращает ошибку, достаточно перенаправить ее источнику вызова этой функции, что аналогично тому, как исключения всплывают в стеке вверх до тех пор, пока не попадут в исходный код обработки ошибок. Разница в том, что этот выбор делается программистом сознательно, а не является чем-то таким, о чём можно забыть.

Поскольку этот шаблон очень распространен, его поддержку на уровне языка можно найти в синтаксисе. В этом синтаксисе используется оператор вопросительного знака (`?`). Оператор `?` чаще всего используется для объектов типа `Result`, и вот как он работает при инспектировании объекта `Result`:

- Если он содержит вариант `Ok`, то вычисление выражения возвращает значение внутри `Ok`.
- Если он содержит вариант `Err`, то он немедленно возвращает `Err` из функции.

Давайте посмотрим на реальный исходный код на языке Rust. Представьте себе, что требуется вызывать функцию `fizzbuzz` и напечатать сообщение в случае успешного выполнения или переслать ошибку в случае отказа. Две приведенные в следующем ниже листинге функции на языке Rust решают проблему аналогичным образом, но в одной из них используется оператор вопросительного знака. Вспомните, что функция `fizzbuzz` возвращает тип `Result<FizzBuzzValue, Err>`.

Листинг 2.34 Пример использования оператора ?

```

fn foo(i: i32) -> Result<FizzBuzzValue, Err> {
    let result = match fizzbuzz(i) { ←
        Ok(x) => {
            x
        }
        Err(e) => {
            return Err(e);
        }
    };
}

```

Поскольку `match` – это не инструкция, а выражение, в языке Rust его можно использовать в позиции выражения, например присваивая переменной результат вычисления выражения `match`.

```

    println!("{} is a valid number for fizzbuzz", i);
}

fn bar(i: i32) -> Result<FizzBuzzValue, Error> {
    let result = fizzbuzz(i)?;
    println!("{} is a valid number for fizzbuzz", i); ←
    Ok(result);
}

```

Обратите внимание на использование оператора ?. Эта строка покинет функцию досрочно, если вызов fizzbuzz вернет Err.

Достигнуть эту строку можно только в том случае, если вызов fizzbuzz возвращает Ok.

Здесь можно заметить, что в первой функции результат выражения `match` присваивается переменной `result`. Ввиду того, что ветвь `Err` выражения `match` возвращается из функции во время ее исполнения, то если выполняется ветвь `Ok`, вычисление всего выражения `match` будет возвращать тип `FizzBuzzValue`, который находится внутри параметра `Ok`. Таким образом, типом переменной `result` в этой функции является не `Result<FizzBuzzValue, Error>`, а `FizzBuzzValue`.

Функциональность второй функции идентична, так как оператор `?`, по сути, является сжатой формой выражения `match` и досрочного возврата, который можно было увидеть в первой функции. Давайте применим эту обработку ошибок с оператором `?` к существующему исходному коду программы FizzBuzz.

Листинг 2.35 Программа FizzBuzz с добавлением оператора ?

```

enum FizzBuzzValue {
    Fizz,
    Buzz,
    FizzBuzz,
    NotDivisible(i32),
}

enum Error {
    GotNegative,
}

fn main() {
    for i in 1..101 {
        match print_fizzbuzz(i) {
            Ok(_) => {}
            Err(e) => {
                match e {
                    Error::GotNegative => {
                        eprintln!("Error: Fizz Buzz only
                                supports positive numbers!");
                        return;
                    }
                }
            }
        }
    }
}

```

```

        }
    }
}

fn print_fizzbuzz(x: i32) -> Result<(), Error> {
    match fizzbuzz(x)? {
        FizzBuzzValue::FizzBuzz => {
            println!("FizzBuzz");
        }
        FizzBuzzValue::Fizz => {
            println!("Fizz");
        }
        FizzBuzzValue::Buzz => {
            println!("Buzz");
        }
        FizzBuzzValue::NotDivisible(num) => {
            println!("{}", num);
        }
    }
    Ok(())
}

fn fizzbuzz(x: i32) -> Result<FizzBuzzValue, Error> {
    if x < 0 {
        Err(Err::GotNegative)
    } else if x % 3 == 0 && x % 5 == 0 {
        Ok(FizzBuzzValue::FizzBuzz)
    } else if x % 3 == 0 {
        Ok(FizzBuzzValue::Fizz)
    } else if x % 5 == 0 {
        Ok(FizzBuzzValue::Buzz)
    } else {
        Ok(FizzBuzzValue::NotDivisible(x))
    }
}

```

Добавлен оператор ?, который будет возвращать досрочно из функции print_fizzbuzz, если вычисление функции fizzbuzz(i) дает Err.

Многие библиотеки Rust разработаны с использованием четко сформулированных типов ошибки, которые используются для определения первопричины отказов. Однако иногда требуется проделать дополнительную работу, чтобы охватывать слишком общие ошибки более конкретными контекстами. Давайте кратко рассмотрим вопрос о том, как преобразовывать ошибки.

2.5.5 Преобразование ошибок

В языке Rust функции, которые могут отказывать, возвращают значения типа `Result`. Следовательно, при инспектировании возвращаемого из функции значения можно четко отделять случай ошибки от случая успешного выполнения. Обычно тип в варианте с ошибкой выражает причину ошибки, давая возможность опреде-

лять причину отказа функции, но в некоторых случаях этого сделать невозможно.

Представьте себе, что требуется написать функцию для выполнения простых валидаций в инструменте создания пользователя. Нужно написать функцию `validate_username`, которая на входе принимает пользовательское имя типа `&str` и на выходе возвращает результат, указывающий на то, была ли валидация успешной или безуспешной, а также природу отказа, если таковой присутствовал. Для выполнения валидации предусмотрены две библиотечные функции: `validate_lowercase` констатирует, что все символы пользовательского имени находятся в нижнем регистре, и `validate_unique` подтверждает, что это пользовательское имя еще не существует в системе. Ни одну из этих функций валидации вы не пишете сами и не можете изменить их сигнатуры. Их сигнатуры выглядят следующим образом:

```
fn validate_lowercase(username: &str) -> Result<(), ()>
```

```
fn validate_unique(username: &str) -> Result<(), ()>
```

Собственная функция `validate_username` должна иметь приведенную ниже сигнатуру и использовать приведенный ниже тип ошибки:

```
enum UsernameError {
    NotLowercase,
    NotUnique,
}
```

```
fn validate_username(username: &str) -> Result<(), UsernameError>
```

Если сделать первоначальный набросок этой задачи, то можно получить что-то вроде этого:

```
fn validate_username(username: &str) -> Result<(), UsernameError>
{
    validate_lowercase(username)?;
    validate_unique(username)?;

    Ok(())
}
```

Если функции `validate_lowercase` и `validate_unique` написаны с учетом типа `UsernameError`, то приведенный выше исходный код в точности соответствует тому, как нужно было бы написать функцию валидации. Однако обе эти функции возвращают один и тот же тип ошибки – единичный тип. Нужен какой-то механизм конвертации этого единичного значения в значения `UsernameError`, которые соответствуют отдельным функциям валидации. Если функция `validate_lowercase` отказывает, то необходимо вернуть `UsernameError::NotLowercase`; и схожим образом для функции `validate_unique`, в каковом случае необходимо возвращать значение `NotUnique`. Этого можно добиться с помощью стандартного выражения `match`, но было бы неплохо, если бы

не нужно было писать много ненужного исходного кода, который ничего не делает в случае `Ok`.

Один из инструментов, к которому можно обратиться за помощью, – это функция на объекте типа `Result` под названием `map_err`. Если вы знакомы с функцией `map` в функциональном программировании, то, возможно, сможете догадаться о назначении функции `map_err`. Функция `map_err` принимает на входе еще одну функцию, которую будем называть `F`, и вызывает `F`, когда результат содержит вариант `Err`. Функция `F` принимает на входе тип в варианте `Err` изначального объекта `Result` и возвращает на выходе новое значение, которое обертыивается в вариант `Err` нового объекта `Result`. На словах это может показаться немного отпугивающим, но на самом деле реализация довольно элементарна:

```
fn map_err<T, E1, E2>(
    r: Result<T, E1>,
    transform: fn(E1) -> E2,
) -> Result<T, E2> {
    match r {
        Ok(x) => Ok(x),
        Err(e) => Err(transform(e)),
    }
}
```

Вот и все – вот и вся функция! Эта реализация немного упрощена, так как еще не был рассмотрен вопрос о том, как писать функции-экземпляры. На практике приведенная выше автономная функция работает точно так же, как `Result::map_err` в стандартной библиотеке. Давайте вернемся к примеру валидации пользовательского имени.

Имеется `Result<(), ()>`, и требуется получить `Result<(), UsernameErr>`. Для того чтобы его получить, можно воспользоваться функцией `map_err` и передать ей функцию с приведенной ниже сигнатурой:

```
fn(err: ()) -> UsernameErr
```

Значение `err` – это значение в варианте `Err` изначального объекта `Result`. Возвращаемая из этой функции ошибка `UsernameErr` будет помещена в вариант `Err` результата, возвращаемого функцией `map_err`. Если объект `Result` содержит значение `Ok`, то переданная в `map_err` функция никогда не будет вызвана. Давайте посмотрим, как применить `map_err` к функции валидации пользовательского имени:

```
fn validate_username(username: &str) -> Result<(), UsernameErr>
{
    validate_lowercase(username).map_err(lowercase_err)?;
    validate_unique(username).map_err(unique_err)?;

    Ok(())
}
```

```
fn lowercase_err(x: ()) -> UsernameError {
    UsernameError::NotLowercase
}

fn unique_err(x: ()) -> UsernameError {
    UsernameError::NotUnique
}
```

Приведенный выше исходный код успешно соотнесет варианты ошибки `UsernameError` с функциями, с которыми они должны быть связаны. Возможно, вам интересно, использует ли этот метод меньше исходного кода, чем при применении некоторых инструкций `match`. На самом деле использование `map_err` с именованными функциями и явными типами параметров / возвращаемых значений не сильно сокращает объем исходного кода. Однако использование замыкания позволяет выражать то же самое меньшим объемом исходного кода.

Замыкания, в других языках программирования иногда именуемые *лямбдами*, представляют собой анонимные функции, которые пишутся внутристрочно. Они очень полезны при использовании функций, которые в качестве параметров принимают другие функции, таких как `map_err`. Замыкания в языке Rust могут содержать одно выражение или блок с несколькими выражениями. Ниже будут рассмотрены замыкания, содержащие одно выражение. Для того чтобы получить замыкание, которое принимает два параметра и возвращает сумму этих двух параметров, пишется следующее:

 $|x, y| \ x + y$

Параметры указываются между символами вертикальной черты через запятые, и сразу после вертикальных черт следует выражение, которое должно быть возвращено из замыкания. Типы параметров в замыканиях могут выписываться в явной форме с использованием синтаксиса, который реплицирует стандартный синтаксис Rust для функций. Однако для того, чтобы можно было аннотировать типы возвращаемых значений, возвращаемое выражение требуется заключать в фигурные скобки. Следующие ниже два замыкания функционально идентичны и могут использоваться как обычные функции:

```
fn main() {
    let add1 = |x: i32, y: i32| -> i32 {x + y};

    let add2 = |x: i32, y: i32| x + y; <--
```

```
    println!("{}", add1(3, 4));
    println!("{}", add2(3, 4));
```

```
}
```

Здесь не нужно сообщать компилятору о том, что данное замыкание возвращает значение типа `i32`, потому что при добавление значения типа `i32` к значению типа `i32` может приводить только к значению типа `i32`.

Хотя типы возвращаемых значений можно аннотировать в явной форме, ввиду природы замыканий, используемых в качестве аргументов для других функций, которые сами предоставляют компилятору подсказки о типе, на практике почти никогда не возникает необходимости писать типы для параметров замыкания или типов возвращаемых ими значений.

Теперь, объединив то, что вы узнали о `map_err`, с замыканиями, можно получить гораздо более компактную реализацию функции `validate_username`:

```
fn validate_username(username: &str) -> Result<(), UsernameError>
{
    validate_lowercase(username).map_err(
        |x| UsernameError::NotLowercase)?;
    validate_unique(username).map_err(
        |x| UsernameError::NotUnique)?;

    Ok(())
}
```

Если попытаться скомпилировать приведенный выше исходный код, то будет получено предупреждение о том, что параметр `X` в замыканиях не используется. Это предупреждение можно отключить, заменив `X` символом подчеркивания, который намекает компилятору о том, что значение параметра игнорируется и не используется:

```
fn validate_username(username: &str) -> Result<(), UsernameError>
{
    validate_lowercase(username).map_err(
        |_| UsernameError::NotLowercase)?;
    validate_unique(username).map_err(
        |_| UsernameError::NotUnique)?;

    Ok(())
}
```

Давайте соберем весь этот исходный код в одну программу, которая выполняет валидацию и показывает результат пользователю.

Листинг 2.36 Программа, выполняющая валидацию пользовательских имен

```
enum UsernameError {
    NotLowercase,
    NotUnique,
}

fn main() {
    match validate_username("user1") {
        Ok(_) => println!("Valid username"),
        Err(UsernameError::NotLowercase) => println!(
            "Username must be lowercase"),
    }
}
```

```

        Err.UsernameError::NotUnique) => println!(
            "Username already exists"),
    }
}

fn validate_username(username: &str) -> Result<(), UsernameError>
{
    validate_lowercase(username).map_err(
        |_| UsernameError::NotLowercase)?;
    validate_unique(username).map_err(
        |_| UsernameError::NotUnique)?;
    Ok(())
}

fn validate_lowercase(username: &str) -> Result<(), ()> {
    Ok(())
}

fn validate_unique(username: &str) -> Result<(), ()> {
    Ok(())
}

```

Функции `validate_lowercase` и `validate_unique` не реализованы, поскольку предполагается, что это библиотечные функции, которые уже существуют.



Иногда, вместо того чтобы передавать ошибку обратно источнику вызова, желательно констатировать отсутствие ошибки и завершать работу программы в целом, если она произошла. Для этого нужно взглянуть на то, что в языке Rust называется поднятием паники при появлении ошибок.

2.5.6 Поднятие паники при появлении ошибок

В языке Rust ошибки являются значениями. Это обычные значения, которые хранятся в переменных, таких как числа, строковые литералы или любые другие разновидности данных, с которыми программа может взаимодействовать. Они не страшны; у них нет какой-то своей особой логики потока управления (кроме явного досрочного возвращения с помощью оператора `?>`). Это просто значения, с которыми нужно работать. Способ работы с ними обычно делегируется источнику вызова на некотором уровне. Источник вызова может регистрировать ошибки в журнале событий и продолжать исполнение, повторять операцию до достижения успеха или полностью сдаться и выйти из программы с ошибкой.

Давайте вернемся к программе FizzBuzz. Теперь представьте себе, что требуется переписать функцию `print_fizzbuzz` таким образом, чтобы она никогда не возвращала значение ошибки и завершала работу всей программы в случае обнаружения ошибки. Это можно сделать, удалив синтаксис `?` из инструкции `match`, повторно введя сочетание `Ok/Err` из листинга 2.33 и заменив исходный код, который передает источнику вызова вариант `err`, на исходный код, который вызывает макрокоманду `panic!`.

Листинг 2.37 Поднятие паники, когда функция print_fizzbuzz видит ошибку

```

enum FizzBuzzValue {
    Fizz,
    Buzz,
    FizzBuzz,
    NotDivisible(i32),
}

enum Error {
    GotNegative,
}

fn main() {
    print_fizzbuzz(-1);           ← Место вызова в функции main
                                  изменено, чтобы обеспечить ис-
                                 полнение обработчика ошибок.
}

fn print_fizzbuzz(x: i32) {
    match fizzbuzz(x) {
        Ok(result) => match result {
            FizzBuzzValue::FizzBuzz => {
                println!("FizzBuzz");
            }
            FizzBuzzValue::Fizz => {
                println!("Fizz");
            }
            FizzBuzzValue::Buzz => {
                println!("Buzz");
            }
            FizzBuzzValue::NotDivisible(num) => {
                println!("{} {}", num);
            }
        },
        Err(Error::GotNegative) => {
            panic!("Got a negative number for fizzbuzz: {}", x);
        }
    }
}

fn fizzbuzz(x: i32) -> Result<FizzBuzzValue, Error> {
    if x < 0 {
        Err(Error::GotNegative)
    } else if x % 3 == 0 && x % 5 == 0 {
        Ok(FizzBuzzValue::FizzBuzz)
    } else if x % 3 == 0 {
        Ok(FizzBuzzValue::Fizz)
    } else if x % 5 == 0 {
        Ok(FizzBuzzValue::Buzz)
    } else {
        Ok(FizzBuzzValue::NotDivisible(x))
    }
}

```

Функция больше не возвращает значение типа Result. Возможность того, что функция может отказать, больше не видна в ее сигнатуре.

Завершающее выражение Ok() было удалено в конце этой ветви match, которое имелось в предыдущих листингах, так как функция больше не возвращает значение типа Result.

Макрокоманда `panic!` – это новая для вас макрокоманда, поэтому давайте вкратце остановимся на ее смысле. Макрокоманда `panic!` в языке Rust похожа на функцию `panic` в языке Go: она поднимает панику в текущем потоке исполнения и разматывает стек до тех пор, пока не будет достигнута вершина стека потока исполнения. Поскольку в программе есть только один главный поток исполнения, макрокоманда `panic!` будет выходить из программы с состоянием ошибки. Вызов макрокоманды `panic!` из фонового потока будет приводить к выходу именно из этого потока. Выход из программы при столкновении с единственной ошибкой может показаться странным, но макрокоманда `panic!` наиболее полезна для выполнения логических констатаций во время исполнения, которые гарантируют, что программа не находится в неработоспособном состоянии или не завершает работу, если обнаружена неустранимая ошибка. Если выполнить исходный код, то в главном потоке исполнения можно увидеть результаты поднятия паники:

```
$ cargo run
thread 'main' panicked at 'Got a negative
    number for fizzbuzz: -1', main.rs:35:7
note: run with `RUST_BACKTRACE=1` environment
      variable to display a backtrace
```

Компилятор Rust выдаст полезные результаты, в которых говорится о том, что вы можете предоставить переменную среды и получить информацию об обратной трассировке. Давайте попробуем:

```
$ env RUST_BACKTRACE=1 cargo run
thread 'main' panicked at 'Got a negative number
    for fizzbuzz: -1', main.rs:33:7
stack backtrace:
 0: rust_begin_unwind
    at /rustc/library/std/src/panicking.rs:475
 1: std::panicking::begin_panic_fmt
    at /rustc/library/std/src/panicking.rs:429
 2: chapter_02_listing_35::print_fizzbuzz
    at ./src/main.rs:33
 3: chapter_02_listing_35::main
    at ./src/main.rs:13
 4: core::ops::function::FnOnce::call_once
    at rustlib/src/rust/library/
      core/src/ops/function.rs:227
note: Some details are omitted, run with
`RUST_BACKTRACE=full` for a verbose backtrace.
```

Хотя это и не сразу бросается в глаза, просмотр пунктов 2 и 3 в трассе стека показывает, что функция `main` вызывает функцию `print_fizzbuzz` в строке 13, а функция `print_fizzbuzz` поднимает панику в строке 33. В более сложных программах на языке Rust трассировка стека оказывается очень полезной. По умолчанию Rust отключает создание отчетов о трассе стека для паник, но, как вы видите, ее можно легко включить.

Добавление поднятия паники в функцию `print_fizzbuzz` привнесло в чтение и написание исходного кода некоторое неудобство. Что, если бы потребовалось получить то же самое паническое поведение, не переписывая блочную инструкцию `match` – поведение, чья работа немного больше похожа на оператор `??`? Это можно сделать, используя метод `.unwrapping()` или `.expect()` на объекте типа `Result`, который поступает из функции `fizzbuzz`. Давайте посмотрим:

```
fn print_fizzbuzz(x: i32) {
    match fizzbuzz(x).unwrapping() {
        FizzBuzzValue::FizzBuzz => {
            println!("FizzBuzz");
        }
        FizzBuzzValue::Fizz => {
            println!("Fizz");
        }
        FizzBuzzValue::Buzz => {
            println!("Buzz");
        }
        FizzBuzzValue::NotDivisible(num) => {
            println!("{} {}", num);
        }
    }
}
```

Данная функция стала намного короче, но она по-прежнему поднимает панику при обнаружении ошибки. Прямо сейчас давайте попробуем ее выполнить:

```
$ cargo run
error[E0599]: no method named `unwrapping` found for enum
`std::result::Result<FizzBuzzValue, Error>` in the current scope
--> src/main.rs:17:21
 |
8 |     enum Error {
|     | ----- doesn't satisfy `Error: std::fmt::Debug`
...
17 |     match fizzbuzz(x).unwrapping() {
|     |         ^^^^^^ method not found in
|     |         `std::result::Result<FizzBuzzValue, Error>`
|     |
|= note: the method `unwrapping` exists but
the following trait bounds were
not satisfied:
`Error: std::fmt::Debug`
```

С этой интересной ошибкой компилятора раньше сталкиваться не приходилось! Расположенное внизу примечание сообщает о том, что метод `.unwrapping()` действительно существует, но его вызов неработоспособен, так как тип ошибки не реализует общую черту `Debug`. Общие черты более подробно обсуждаются в главе 3, а пока просто скажем, что типы, реализующие общую черту `Debug`, могут печатать-

ся в терминале в представлении, которое полезно для разработчиков. Общая черта `Debug` легко добавляется в тип ошибки, используя для этого специальную директиву компилятора под названием `derive`. Вот как она выглядит:

```
#[derive(Debug)]
enum Error {
    GotNegative,
}
```

Благодаря ей можно получить несколько разных общих черт, но `Debug` является одной из наиболее распространенных. По сути, этот исходный код сообщает компилятору Rust о том, что нужно сгенерировать код, который может превращать значение ошибки в строковое представление, чтобы можно было определять тип ошибки, к которому оно относится, при взгляде на него. Перечисления в языке Rust представляются во время исполнения числами, и распечатка числового значения перечисления, как правило, несет никакой пользы. Общая черта `Debug` очень похожа на метод `toString` в языке Java, но она может генерироваться компилятором автоматически с помощью директивы `derive`. В следующем ниже листинге показано, как должна выглядеть полная программа.

Листинг 2.38 Использование метода `.unwrap()` для поднятия паники при обнаружении ошибки

```
enumFizzBuzzValue {
    Fizz,
    Buzz,
    FizzBuzz,
    NotDivisible(i32),
}

#[derive(Debug)]
enum Error {
    GotNegative,
}

fn main() {
    print_fizzbuzz(-1);
}

fn print_fizzbuzz(x: i32) {
    match fizzbuzz(x).unwrap() {
        FizzBuzzValue::FizzBuzz => {
            println!("FizzBuzz");
        }
        FizzBuzzValue::Fizz => {
            println!("Fizz");
        }
        FizzBuzzValue::Buzz => {
```

```

        println!("Buzz");
    }
    FizzBuzzValue::NotDivisible(num) => {
        println!("{}", num);
    }
}

fn fizzbuzz(x: i32) -> Result<FizzBuzzValue, Error> {
    if x < 0 {
        Err(Error::GotNegative)
    } else if x % 3 == 0 && x % 5 == 0 {
        Ok(FizzBuzzValue::FizzBuzz)
    } else if x % 3 == 0 {
        Ok(FizzBuzzValue::Fizz)
    } else if x % 5 == 0 {
        Ok(FizzBuzzValue::Buzz)
    } else {
        Ok(FizzBuzzValue::NotDivisible(x))
    }
}

```

Теперь, когда тип ошибки реализует общую черту `Debug`, давайте попробуем выполнить программу, чтобы увидеть, как выглядит паника:

```
$ cargo run
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: GotNegative', src/main.rs:18:21
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Обратите внимание, что в сообщении об ошибке указано местоположение ошибки (строка 18 файла `main.rs` из листинга 2.38 и получено значение (`Debug`-представление ошибки `Error`, которое равно `GotNegative`). Если вы только начинаете программу на языке Rust, то простейшей формой обработки ошибок часто является добавление метода `.unwrap()` после всех функций, которые могут завершиться отказом, так как это бывает проще, чем настройка надлежащих типов возвращаемых значений объекта `Result` с помощью высокуюровневой обработки ошибок. В крупных программах очень важно иметь надлежащий исходный код обработки ошибок. Ведь совсем нежелательно, чтобы веб-сервер поднимал панику и аварийно отказывал во время выполнения из-за того, что кто-то отправил запрос с недействительными данными. Однако поднимать панику во время фазы инициализации веб-сервера абсолютно допустимо, если конфигурационные файлы содержат синтаксические или семантические ошибки, так как в таком сценарии нет работоспособного пути вперед.

Используя метод `.unwrap()`, можно выводить информацию на консоль, но иногда возникает потребность предоставлять чуть больше. Метод `.expect` очень похож на метод `.unwrap()` и позволяет писать небольшое сообщение, которое распечатывается вместе с паникой,

чтобы предоставлять пользователю дополнительный контекст вокруг ошибки. Давайте отредактируем функцию `print_fizzbuzz`, чтобы воспользоваться методом `expect` вместо `unwrap`:

```
fn print_fizzbuzz(x: i32) {
    match fizzbuzz(x).expect("Failed to run fizzbuzz") {
        FizzBuzzValue::FizzBuzz => {
            println!("FizzBuzz");
        }
        FizzBuzzValue::Fizz => {
            println!("Fizz");
        }
        FizzBuzzValue::Buzz => {
            println!("Buzz");
        }
        FizzBuzzValue::NotDivisible(num) => {
            println!("{} {}", num);
        }
    }
}
```

Выполнив исходный код сейчас, будет получено немного улучшенное сообщение об ошибке:

```
$ cargo run
thread 'main' panicked at 'Failed to run fizzbuzz:
GotNegative', main.rs:18:21
note: run with `RUST_BACKTRACE=1` environment variable
      to display a backtrace
```

Теперь, даже не заглядывая в исходный код, понятно, что ошибка была напрямую связана с функцией `fizzbuzz`. Источник ошибок в этой небольшой программе вполне очевиден, но в крупных программах метод `expect` оказывается гораздо полезнее, чем метод `unwrap`.

Краткий итог

- Системы владения и заимствования в языке Rust обеспечивают высокую производительность, снимая заботу об ошибках, возникающих при ручном управлении памятью.
- Владение значением позволяет компилятору Rust определять точку, когда оно будет создано, пригодно для использования и отброшено, еще до запуска программы.
- Все значения во всех языках программирования имеют времена жизни, но компилятор Rust применяет эти правила в явной форме.
- Система времен жизни в языке Rust позволяет компилятору знать, что ссылки всегда являются действительными и что вы никогда не будете читать данные из недопустимой памяти.
- В языке Rust есть несколько строковых типов, которые дают программисту полный контроль над резервированием памяти. Некоторые типы допускают возможность мутации после создания,

тогда как другие представляют собой доступные только для чтения виды на строковые данные.

- Перечисления используются для хранения объектов с предопределенным списком возможных значений.
- Функции, которые могут отказывать во время исполнения, возвращают значение типа `Result`, который представляет собой перечисление, содержащее показатель успешного выполнения или отказа, а также значение в случае успешного выполнения и значение ошибки в случае отказа.
- Значение успешного выполнения из объекта `Result` невозможно использовать, не обрабатывая возможность ошибки.
- Единичный тип, или `()`, – это тип и значение, которое ничего не представляет.
- Создание конкретно-прикладного типа ошибки – это образец наилучшей практики программирования на языке Rust.
- Оператор `?` используется для досрочного возвращения из функций, если результат содержит ошибку.
- Функция `map_err` используется для преобразования объекта `Result`, содержащего ошибку одного типа, в объект `Result`, содержащий ошибку другого типа.
- Замыкания используются в качестве аргументов функций, которые принимают другие функции в качестве параметров.
- Макрокоманда `panic!` используется для разматывания стека потока исполнения, когда программа находится в неработоспособном состоянии и должна завершиться.
- Методы `.unwrap()` и `.expect()` используются для поднятия паники, если результат содержит ошибку.

3

Введение в интерфейс с внешними функциями С и незащищенный язык Rust

Эта глава охватывает следующие ниже темы:

- понимание интерфейса с внешними функциями С и его взаимосвязи с незащищенным языком Rust;
- выполнение обычно запрещенных операций с помощью незащищенного языка Rust;
- переработка компонента программы на языке С на язык Rust.

В предыдущей главе был представлен общий обзор исходного кода на языке Rust и рассмотрены некоторые его элементы, которые могут показаться неожиданными или трудными для понимания начинающим разработчикам. Теперь, когда вы научились писать простые программы на языке Rust, в этой главе будет рассмотрен пример внедрения исходного кода на языке Rust в существующую программу на языке С.

Если требуется внедрить исходный код на языке Rust в существующее приложение, то нужна очень четко определенная семантика относительно того, как происходит обмен данными между двумя языками, как передаются значения между ними и как используется или не используется коллективная память между ними. В идеале такой

интерфейс между двумя языками должен хорошо поддерживаться на большом количестве разных языков и платформ во избежание переписывания исходного кода при выполнении конкретной интеграции.

Одним из хорошо поддерживаемых методов является написание функций, которые во время исполнения ведут себя идентично функциям языка С. В них используются одинаковые правила о вызовах; они передают параметры и возвращают значения одинаковым образом и используют типы, которые могут быть безопасно представлены на любом языке. Этот метод называется *интерфейсом с внешними функциями C* (C Foreign Function Interface, аббр. C FFI). В данной главе обсуждается вопрос о том, как писать такие функции на языке Rust и использовать поддержку указанного интерфейса в языке Rust при интеграции исходного кода на данном языке в приложение на языке С. Кроме того, будет затронут вопрос о том, как использовать незащищенные блоки и функции (блоки и функции `unsafe`), чтобы выполнять некоторые операции, которые не допускаются в обычном исходном коде на языке Rust, и когда и почему эти блоки необходимы при написании кода для интерфейса с внешними функциями.

3.1 Незащищенный язык Rust

Одним из главных преимуществ языка Rust является защищенность памяти, которую он обеспечивает разработчикам приложений. Однако иногда возникает потребность частично отказываться от этой защищенности памяти, чтобы улучшать производительность, упрощать работу или, что наиболее интересно, работать с типами, о которых компилятор Rust не может судить. Как вы знаете из обсуждения темы системы времен жизни и владения в главе 2, компилятор Rust способен делать выводы о том, когда память защищена и ее можно безопасно использовать, а когда ее можно обнулить, основываясь на соблюдении нескольких правил в исходном коде. Однако компилятор Rust не может делать никаких допущений о способах резервирования памяти, доступа к ней или высвобождения в любом другом исходном коде, кроме исходного кода на языке Rust. Если есть потребность работать с динамической памятью, которая не была создана в исходном коде на языке Rust, то нужно использовать *незащищенный* исходный код.

ПРИМЕЧАНИЕ. «Незащищенный» (`unsafe`) – это несколько неправильный термин, так как он не отменяет требований к защищенности, которые предъявляются к остальной части исходного кода на языке Rust. Он просто означает, что разработчик несет ответственность за соблюдение правил языка Rust в отношении защищенности без строгой проверки их разработчиком. Правильнее было бы использовать термин «непроверенный» (`unchecked`). Однако для обозначения этих блоков в языке используется ключевое слово `unsafe`, поэтому будем продолжать называть их незащищенными.

Незащищенные блоки кода позволяют выполнять несколько операций, которые запрещены в защищенном исходном коде Rust:

- брать значения по адресам, на которые указывают обычновенные указатели;
- вызывать функции, помеченные как `unsafe`;
- реализовывать функции, помеченные как `unsafe`;
- подвергать мутации статические значения;
- обращаться к полям типа `union`.

На самом деле, кроме этих пяти пунктов, больше нет ничего. Других секретных волшебных или опасных операций нет. Без сомнения, самой фундаментальной из всех этих незащищенных операций является взятие значений по обычновенным указателям.

3.1.1 Обыкновенные указатели

Как обсуждалось в главе 2, указатели – это значения, которые сообщают о расположении других значений в памяти. Если представить основную память компьютера в виде гигантского байтового массива, то указателями будут индексы в этом массиве. Значением указателя является адрес памяти, размер которого зависит от архитектуры компьютера. В большинстве современных систем адресация памяти осуществляется на байтовом уровне с использованием 64-битовых адресов, в силу чего указатели представляют собой 64-битовые числа, указывающие на отдельные байты в памяти компьютера.

Взятие значения по указателю (или снятие косвенности с указателя, англ. dereference) означает обращение к значению, на которое указывает указатель. На рис. 3.1 показана стековая память во время исполнения простой программы на языке С. Она включает в себя символьную переменную `x`, переменную `y`, указывающую на символьную переменную `x`, и символьную переменную `z`, которой присваивается результат взятия значения по `y`. Представьте себе, что вы выполняете эту программу С на компьютере с однобайтовыми адресами указателей. Стрелка слева представляет строку в программе, которая только что была выполнена, а диаграмма справа представляет объем стековой памяти на данный момент времени.

Причина, по которой эта операция должна быть скрыта за блоками `unsafe`, очень проста. Вспомните второе правило ссылок языка Rust из главы 2: ссылки всегда должны быть действительными. Во время исполнения ссылка и обычновенный указатель идентичны: они оба являются значениями, содержащими адрес в памяти, который используется для взятия в ней значения. Единственное различие заключается в их поведении во время компиляции. Поскольку ссылки Rust содержат дополнительную информацию, известную компилятору, например время их жизни, компилятор знает, что они всегда действительны и что операция взятия значений по ним всегда защищена. Если создается обычновенный указатель, то это просто адрес в памяти; к нему не прикреплена информация о времени жизни или владельце. У компилятора нет возможности удостовериться, что память, на которую он указывает, является допустимой, поэтому эта валидация зависит от программиста.

Точка
исполнения

```
→ int main() {
    char x = 'a';
    char *y = &x;
    char z = *y;
}
```

0x07	
0x06	
0x05	
0x04	
0x03	
0x02	
0x01	
0x00	

На этой диаграмме показано
состояние стековой памяти
программы после исполнения
указанной слева строки.

Никакой код еще не был
исполнен, поэтому стек
полностью пуст.

```
→ int main() {
    char x = 'a';
    char *y = &x;
    char z = *y;
}
```

0x07	
0x06	
0x05	
0x04	
0x03	
0x02	
0x01	
0x00	'a'

В первой позиции стека
сохраняется символ 'a'

```
→ int main() {
    char x = 'a';
    char *y = &x;
    char z = *y;
}
```

0x07	
0x06	
0x05	
0x04	
0x03	
0x02	
0x01	0x00
0x00	'a'

Следующее помещенное
в стек значение – это адрес
переменной x. В данном случае
значение адреса равно 0x00.

```
→ int main() {
    char x = 'a';
    char *y = &x;
    char z = *y;
}
```

0x07	
0x06	
0x05	
0x04	
0x03	
0x02	
0x01	0x00
0x00	'a'

Эта операция называется
взятием значения по указателю
или ссылке. Переменная
у содержит адрес памяти
0x00. Выполняется выборка
значения, которое хранится

Рисунок 3.1 Стековая память программы во время операций присвоения ссылке адреса
значения и взятия значения по ссылке/указателю

Одной из наиболее распространенных операций в исходном коде на языке Rust, работающем на разных языках, является чтение данных через буфер, такой как массив в стиле C.

Листинг 3.1 Чтение элементов вектора с использованием арифметических операций на указателях

Тип **Vec** в языке Rust – это динамически расширяемый сплошной блок памяти, содержащий множество смежно расположенных значений одного типа.

```
fn main() {
    let data: Vec<u8> = vec![5, 10, 15, 20];
    read_u8_slice(data.as_ptr(), data.len());
}
```

Метод **as_ptr** абсолютно защищен.

```

fn read_u8_slice(slice_p: *const u8, length: usize) {
    for index in 0..length {
        unsafe { // Блок unsafe обусловлен тем, что выполняются две незащищенные операции: вызывается незащищенная функция offset и затем по возвращаемому указателю берется значение.

            println!("slice[{}]= {}", index,
                     *slice_p.offset(index as isize)); // Функция offset выполняет арифметические операции на указателях; она нуждается в том, чтобы на ее вход поступали данные типа isize, поскольку она принимает отрицательные смещения.
        }
    }
}

```

Тип `Vec` аналогичен типу `std::vector` в языке C++ или `ArrayList` в языке Java и похож на `list` в языке Python, хотя списки там могут содержать значения разных типов. Тип `u8` – это 8-битовое целое число без знака, один байт. Если объединить их в виде `Vec<u8>`, то получится динамически расширяемый блок памяти, содержащий отдельные байтовые значения.

Метод `as_ptr` используется для получения указателя на буфер данных внутри `Vec`. Операция получения указателя абсолютно защищена. Вводить `unsafe` нужно только тогда, когда требуется взять значение по указателю.

Немутируемые указатели (`*const`) и мутируемые указатели (`*mut`) очень похожи на соответственно немутируемые и мутируемые ссылки. Если значение находится за `*const`, то оно не может быть подвергнуто мутации. Если нужно подвергнуть значение мутации, то необходимо использовать `*mut`. Одно из ключевых различий между указателями и ссылками в этом отношении заключается в том, что немутируемый указатель может быть приведен к мутируемому указателю. Разработчик несет ответственность за то, чтобы знать, когда это действие является защищенным, а когда нет.

3.2 Интерфейс с внешними функциями С

Разобравшись со взятием значений по указателям, теперь можно написать исходный код на языке Rust, который обменивается данными с исходным кодом на языке С. Для того чтобы читать и писать по указателям, которые исходный код на языке Rust принимает из С, необходимо знать, как работают операции на указателях.

Представьте себе, что существует приложение на языке С, которое решает простые арифметические выражения в формате обратной польской нотации (Reverse Polish Notation, аббр. RPN). В настоящее время эта программа принимает арифметические выражения, содержащие одну операцию. Перед вами была поставлена задача расши-

рить приложение за счет поддержки нескольких операций в одном выражении. Эта дополнительная функциональность должна быть написана на языке Rust; однако нынешний исходный код на С, который выполняет пользовательские операции, такие как ввод и вывод текста, должен оставаться на языке С.

Обратная польская нотация – это способ написания арифметических выражений, который устраниет необходимость в правилах приоритета операций. По сути, это простой язык программирования, работающий на стековой машине. Члены арифметического выражения разделены пробелами, и арифметические операторы работают не на предшествующем и последующем членах выражения, как в случае с более часто используемыми инфиксными операциями, а на двух предыдущих членах выражения. Вот несколько примеров арифметических выражений, написанных соответственно в инфиксной нотации, и их аналогов в обратной польской нотации:

```
Prefix: 3 + 4 * 12
RPN   : 4 12 * 3 +
        = 51
```

```
Prefix: (3 + 4) * 12
RPN   : 3 4 + 12 *
        = 84
```

На рис. 3.2 показан стек, который используется для вычисления результата второго арифметического выражения в формате обратной польской нотации (RPN).

Обратная польская нотация позволяет избегать двусмысленности инфиксной нотации, всегда оперируя строго в порядке слева направо. Порядок следования операций для первого и второго выражений в формате обратной польской нотации отличается, поскольку операции буквально записываются в другом порядке. Калькулятор, который выполняет разбор арифметического выражения в формате обратной польской нотации, написать гораздо проще, потому что можно избежать сложностей, связанных с упорядочиванием операций, и просто двигаться слева направо.

В настоящее время приложение на языке С принимает от пользователя целочисленные арифметические выражения, разделенные символами новой строки, в STDIN, выполняет разбор арифметического выражения, а затем вычисляет и показывает результат в STDOUT. От вас требуется добавить поддержку нескольких вложенных арифметических выражений; сейчас калькулятор выполняет только по одной операции за раз. Весь этот исходный код на языке С можно было бы оставить либо перенести исходный код разбора строковых литералов из языка С на язык Rust. Поскольку вы слышали много хорошего о языке Rust, вы решаете его применить для решения поставленной задачи. Сначала давайте посмотрим, как выглядит исходный код на языке С.

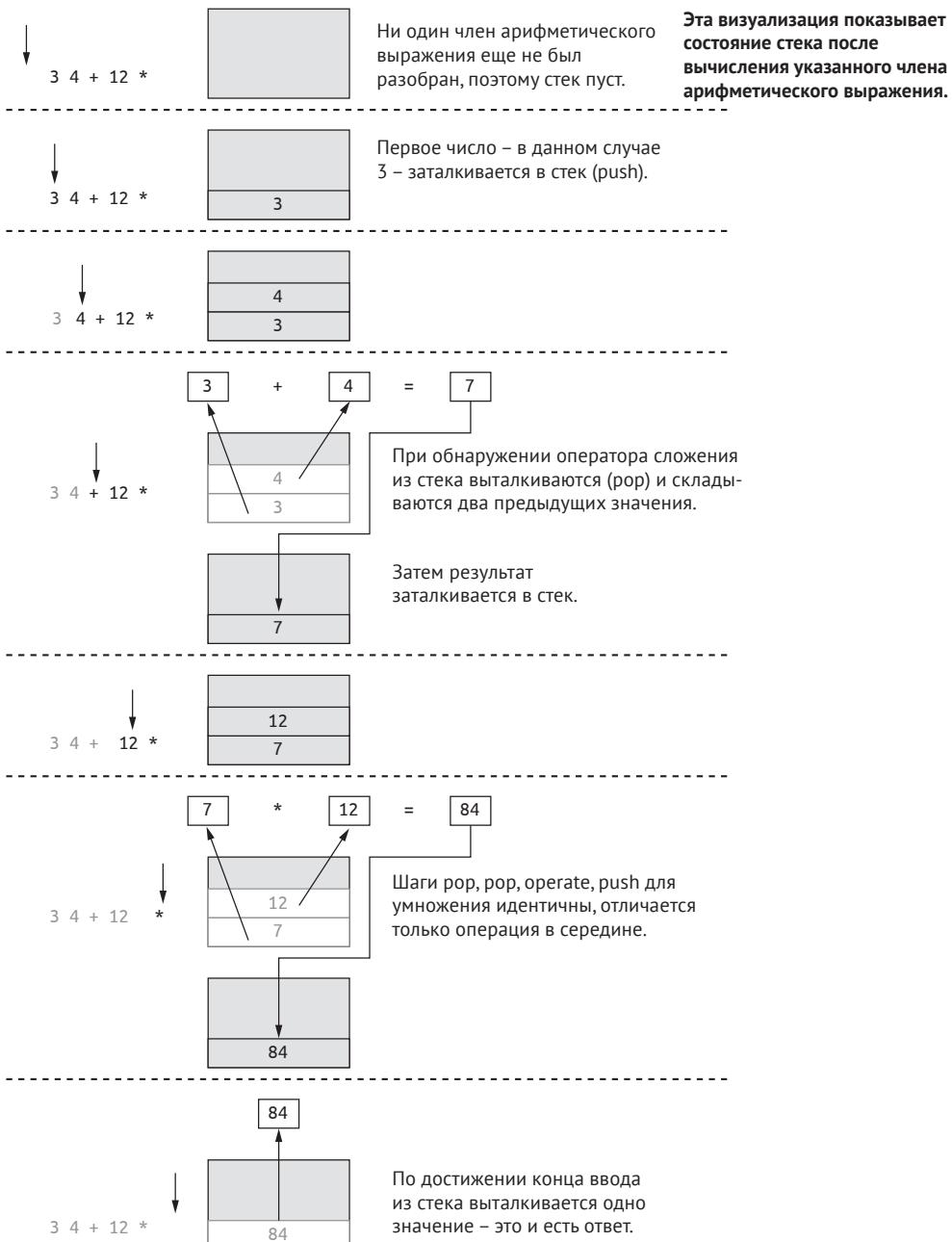


Рисунок 3.2 Стек в формате обратнойпольской нотации, используемый для вычисления арифметического выражения $3 \ 4 \ + \ 12 \ *$

Листинг 3.2 Программа простого арифметического калькулятора на языке С

```
#include <stdio.h>
#include <string.h>

int solve(char *line, int *solution);

int main() {
    char line[100]; ← Резервирует место
    int solution;   в стеке функции main.

    while (1) {
        printf("> ");
        if (fgets(line, 100, stdin) == NULL) { ←
            return 0;
        }

        if (solve(line, &solution)) {
            continue;
        }

        printf("%d\n", solution);
    }

    return 0;
}

int solve(char *line, int *solution) { ←
    int num1, num2;
    char operator;

    int values_read = sscanf(
        line, "%d %d %c", &num1, &num2, &operator); ←
    if (values_read != 3) { ←
        return 1;
    }

    switch (operator) { ←
        case '+':
            *solution = num1 + num2; ←
            return 0;
        case '-':
            *solution = num1 - num2;
            return 0;
        case '*':
            *solution = num1 * num2;
            return 0;
        case '/':
            *solution = num1 / num2;
            return 0;
    }

    return 1;
}
```

Функция `solve` принимает указатель на строку текста, читаемую из STDIN, и указатель на значение типа `int`, в которое `solve` пишет значение решения.

Здесь форматный строковый литерал будет отыскивать целое число, за которым следует один символ, а затем еще одно целое число. Эти значения будут использоваться для вычисления арифметического выражения.

В этой инструкции `switch` вычисляется результат предоставленного арифметического выражения и записывается в целое число, на которое указывает указатель `solution`. Вспомните, что `solution` указывает на переменную типа `int` в стеке функции `main`.

Выражение `char line[100]`; резервирует пространство в стеке функции `main` для хранения до 100 символов данных, которые будут считываться от пользователя. Поскольку обращаться к нескольким строкам текста одновременно не нужно, можно продолжить реиспользовать один и тот же буфер памяти снова и снова. Функция `fgets` его очистит, когда она прочитает данные из STDIN.

Функция `fgets` читает данные из STDIN и принимает в качестве первого аргумента указатель на символ – он должен указывать на зарезервированную память, в которую будут читаться данные из файла. В памяти должно быть зарезервировано место как минимум для такого же количества символов, как и во втором аргументе. Поскольку было зарезервировано пространство для 100 символов, в качестве второго аргумента указывается 100. Указатели в языке С и связанная с ними память не содержат данных о том, где заканчивается зарезервированная область памяти, поэтому разработчику необходимо явным образом конкретизировать размер областей памяти для многих функций, тем самым обеспечивая, чтобы `fgets` никогда не писал за пределами буфера.

Функция `solve` возвращает значение типа `int`, которое является кодом состояния. `0` означает, что функция сработала правильно, а `1` означает, что строковый литерал был обработан не так, как положено.

Если поместить этот исходный код в файл с именем `calculator.c` и исполнить его, то он будет решать простые арифметические задачи, как и ожидалось:

```
$ gcc calculator.c -o calculator
$ ./calculator
> 3 40 *
120
> 120 3 /
40
> 40 1345 *
53800
> 53800 3 /
17933
```

Он отлично справляется с приведенными выше простыми арифметическими выражениями, но что произойдет, если попытаться добавить дополнительные операции?

```
> 3 40 * 2 -
120
> 10 10 * 10 *
100
> 10 10 * hello!
100
```

Все, что идет после первых трех членов, игнорируется. Напомним, что была поставлена задача добавить в калькулятор поддержку нескольких операций в одном выражении. Давайте посмотрим, получится ли извлечь из него ключевой компонент и перенести его на язык Rust!

Первый шаг состоит в выявлении того, что нужно извлечь. Учитывая, что в данной программе есть только две функции, и одна из них является `main`, перенос на язык Rust необходимо начать с функции `solve`.

Давайте начнем новый проект Rust с помощью команды Cargo. В предыдущих примерах использовалась команда `cargo new PROJECT_NAME`, но она создает новый проект с точкой входа `main.rs` – с того, что может запускаться непосредственно как исполняемый файл. Здесь же исполняемый файл не создается; вместо него требуется создать библиотеку. Поэтому команде `cargo new` нужно предоставить дополнительный флаг, чтобы указать на это намерение:

```
cargo new --lib calculate
```

Откройте только что созданный файл `calculate/src/lib.rs`, и можно начинать. Вспомните, что при создании исполняемого файла вновь созданные файлы `main.rs` по умолчанию содержат программу «Hello world!». При создании библиотеки происходит аналогичное – Cargo заполнит файл `lib.rs` базовым каркасом модульного тестирования, который можно использовать для валидации функциональности программы. Межъязыковое тестирование будет рассмотрено подробнее в главе 7; а пока просто надо удалить содержимое этого файла.

При переносе функциональности функции `solve` из языка C на язык Rust нужно снабдить исходный код на C функцией, которая имеет ту же *сигнатуру*, что и старая функция `solve`. Сигнатуре функции относится к типам всех значений, которые функция принимает в качестве параметров на входе и возвращает на выходе, а также к семантическим смыслам этих значений. Вспомните сигнатуру функции на C:

```
int solve(char *line, int *solution)
```

Для того чтобы исходный код на языке C вызывал функцию на языке Rust, нужно написать функцию на Rust, которая принимает указатель на тип `char` и указатель на тип `int` в качестве параметров и возвращает значение типа `int`. Вот как будет выглядеть та же сигнатура на языке Rust:

```
fn solve(line: *const c_char, solution: *mut c_int) -> c_int
```

Из сигнатуры функции на Rust уже можно извлечь больше информации, чем из сигнатуры функции на C. Функция на языке Rust сообщает о том, что значение параметра `solution` может быть изменено внутри функции, а значение параметра `line` – нет. Исходный код на языке C не содержит никаких указаний на то, что параметр `solution` будет изменен функцией `solve`. Разумеется, разработчик всегда может добавить комментарии, но комментарии могут быть неточными или устаревшими.

Типы `c_char` и `c_int` в сигнатуре функции не встроены в стандартную библиотеку языка Rust; их необходимо импортировать из пакета `libc`. Принятый в языке Rust термин *crate* (дословно: упаковка) в языке Rust используется для обозначения пакетов или библиотек – коллекций функций и типов, которые могут использоваться другими

пользователями для выполнения определенных задач. Пакет `libc` предоставляет простые привязки через интерфейс с внешними функциями (FFI) к стандартной библиотеке С. Стандарт языка С предоставляет относительные гарантии в отношении размера. Например, тип `int` всегда имеет как минимум такой же размер, что и тип `short int`, но, сверх того, тип `int` в языке С зависит от платформы. Пакет `libc` частично абстрагируется от этой платформенно-специфичной природы, предоставляя типы Rust для примитивов языка С, размер которых определяется платформой, на которой они были скомпилированы. Поскольку многим программам на языке Rust не требуется взаимодействовать с библиотеками С, эта функциональность не включена в стандартную библиотеку, а находится во внешней библиотеке.

3.2.1 Включение пакета Rust

Ранее использование менеджера пакетов Cargo было ориентировано на создание новых пакетов Rust или на компиляцию и исполнение программы Rust. Однако Cargo может делать гораздо больше. Он также может скачивать, компилировать и связывать зависимости и выполнять многие другие функции, которые обычно требуют большой работы по настройке в программах на языке С или C++. Это универсальная программа для взаимодействия с языком Rust. На данный момент менеджер пакетов Cargo будет сообщено о необходимости включить `libc` при компиляции пакета `calculate`.

Конфигурационный файл Cargo называется `Cargo.toml`. Здесь содержится вся необходимая ему информация о том, как собирать пакет. Указанный файл содержит наборы подлежащих активации функциональных компонентов компилятора, подлежащие скачиванию/компиляции сторонние пакеты и их версии, флаги условной компиляции и информацию, которую необходимо включать, если создается пакет, который предполагается распространять среди других пользователей (например, контактную информацию, справку `readme`, информацию о версии и многое другое).

Откройте `calculate/Cargo.toml` в редакторе. Его содержимое должно быть уже заполнено командой `cargo new` и выглядеть примерно так, как показано в следующем ниже листинге.

Листинг 3.3 Дефолтный конфигурационный файл Cargo

```
[package]
name = "calculate"
version = "0.1.0"
authors = ["You <you@you.com>"]
edition = "2018"

# See more keys and their definitions at
# https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
```

Секция `[dependencies]` указанного файла используется большинством разработчиков на языке Rust наиболее часто. Под этой строкой набираются имя и номер версии пакета, который требуется включить. Впоследствии, когда команды Cargo будут применяться для компиляции программы Rust, менеджер Cargo будет скачивать соответствующую версию запрошенных пакетов, компилировать их и связывать с собираемым пакетом. При этом отпадает необходимость беспокоиться об установке флагов компилятора. Здесь нет отдельного шага; достаточно указать нужные пакеты, и Cargo их доставит. Для того чтобы найти имеющиеся пакеты, обратитесь к реестру [crates.io](#). Когда Cargo используется для создания и публикации пакетов, они (по умолчанию) отправляются в реестр [crates.io](#). Здесь можно ознакомиться со всеми публично доступными пакетами, которые можно использовать при создании приложений Rust, а также хранить свои собственные пакеты.

Для того чтобы включить `libc` в пакет `calculate`, добавьте строку в секции `[dependencies]`. Зависимости конкретизируются именем пакета, знаком равенства (`=`) и версией пакета, который требуется использовать. На момент написания этой книги последней версией пакета `libc` была версия `0.2.80`, поэтому давайте использовать эту версию. После указанного добавления файл `Cargo.toml` должен выглядеть следующим образом:

```
[package]
name = "calculate"
version = "0.1.0"
authors = ["You <you@you.com>"]
edition = "2018"

# See more keys and their definitions at
# https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
libc = "0.2.80"
```

Сюда можно включить столько зависимостей, сколько потребуется, но пока что нужен только `libc`.

После этого добавления откройте файл `calculate/src/lib.rs` еще раз, и давайте попробуем написать базовую функцию `solve`.

Листинг 3.4 Самая простая функция `solve` на языке Rust, которая компилируется

```
use libc::{c_char, c_int};
```

```
fn solve(line: *const c_char, solution: *mut c_int) -> c_int {
```

0

}

Последнее выражение в функции трактуется как возвращаемое значение, если после него нет точки с запятой, поэтому данная строка эквивалентна возвращению 0.

Инструкция `use` вставляет типы/функции/переменные из других пакетов или модулей Rust.

Система модулей будет рассмотрена в главе 5, а пока просто надо запомнить, что инструкция `use` вставляет элементы из других пакетов. Инструкция `use` не обязательна для каждого элемента, который требуется вставить, но если бы из этой инструкции был исключен `c_char`, то в сигнатуре функции на него пришлось бы ссылаться как `libc::c_char`. Неявное возвращение из функции без правила точки с запятой на первый взгляд может показаться странным, но в сочетании с некоторыми другими выражениями Rust оно становится бесценным.

Если этот исходный код скомпилировать, то можно будет увидеть, что Cargo вставляет пакет `libc`. Поскольку напрямую исполняемый файл создавать не нужно, применяется команда `cargo build`, чтобы скомпилировать пакет, не пытаясь его исполнить. Использовавшаяся в предыдущих примерах команда `cargo run` выполняет то же самое, что и команда `cargo build`, но она воспроизводит результирующий исполняемый файл, если пакет является исполняемым файлом:

```
$ cargo build
  Updating crates.io index
  Compiling libc v0.2.80
  Compiling calculate v0.1.0 (/home/you/calculate)
    Finished dev [unoptimized + debuginfo] target(s) in 5.81
```

Скомпилировав функцию `solve`, теперь давайте посмотрим, получится ли ее вызвать из исходного кода на языке C!

3.2.2 Создание динамической библиотеки с использованием языка Rust

Если вы много занимались программированием за пределами уровня программы «Hello world!», то вам уже приходилось взаимодействовать с библиотеками. Библиотеки – это коллекции функций, типов, переменных или других элементов, в зависимости от того, что поддерживает ваш язык программирования, которые упакованы вместе для выполнения некоторой функциональности, поэтому их не нужно реализовывать повторно всякий раз, когда в них возникает потребность. Например, если требуется выполнять HTTP-запросы на языке Python, то можно воспользоваться библиотекой `requests`, а на языке C – библиотекой `libcurl`. Гораздо проще импортировать библиотеку, чтобы выполнять HTTP-запросы, чем использовать необработанные сокеты и системные вызовы `read/write`.

В разных языках программирования библиотеки имеют разные форматы. Например, библиотеки языка Python – это просто коллекции файлов исходного кода Python, которые интерпретатор Python читает при импорте. В языке C существует несколько разных типов библиотек, но наиболее часто используемой в Unix-подобных операционных системах является динамическая библиотека, на которой ниже будет сосредоточено все внимание.

Прежде чем функцию Rust `solve` можно будет вызывать из программы на C, нужно выполнить несколько шагов:

- 1 Сообщить менеджеру пакетов Cargo, чтобы он скомпилировал пакет как динамическую библиотеку, понятную компоновщику С.
- 2 Добавить вновь созданную динамическую библиотеку в путь поиска, в котором компоновщик ищет библиотеки и объектные файлы.
- 3 Пометить функцию Rust `solve`, чтобы компилятор Rust знал, как ее скомпилировать с использованием правил С о вызовах.
- 4 Перекомпилировать программу на С, используя функцию `solve` из динамической библиотеки Rust.

Давайте пройдемся по этим шагам.

Создание динамической библиотеки

Когда менеджер пакетов Cargo компилирует пакет Rust, по умолчанию это не приводит к созданию чего-либо, что компилятор С знает, как использовать. Менеджер Cargo генерирует некий файл `rlib`, то есть тип файла, специфичный для компилятора Rust и используемый только в качестве промежуточного артефакта, который позже будет использован в какой-либо другой компиляции Rust. Вместо файла `rlib` желательно, чтобы Cargo генерировал динамическую библиотеку, которую компоновщик С знает, как использовать. В файл `Cargo.toml` нужно внести еще одно изменение. На этот раз надо сообщить ему о том, что нужно вывести что-то совместимое с языком С. Добавьте вот эти строки в свой файл `Cargo.toml` над секцией `[dependencies]`:

```
[lib]
crate-type = ["cdylib"]
```

Cargo может генерировать много разных типов пакетов, но наиболее распространенными являются дефолтные `rlib` и `cdylib`, которые позволяют Cargo собирать динамическую библиотеку, совместимую с программами на языке С. После внесения этого дополнения в файл `Cargo.toml` выполните команду `cargo build` повторно.

Добавление динамической библиотеки в путь поиска

Когда менеджер пакетов Cargo что-либо компилирует, он заходит в каталог с именем `target`. Внутри каталога `target` Cargo создает подкаталоги для разных профилей сборки. На данный момент просто `debug`, поскольку по умолчанию Cargo производит двоичные файлы с отладочной информацией и без оптимизаций, но позже будет рассмотрено, как создавать оптимизированные сборки. Если заглянуть в каталог `target/debug`, то можно будет увидеть несколько файлов и папок, но самая важная из них – это файл новой динамической библиотеки, `libcalculate.so`. Файл динамической библиотеки нужно поместить в папку, которую компилятор и компоновщик С будут искать при выполнении программы-калькулятора. Это можно сделать, создав ссылку в каталоге `/lib`, которая указывает на файл библиотеки. В каталоге `/lib` хранятся файлы динамических библиотек, и указанный каталог просматривается компилятором С, компоновщиком С и операционной системой при выполнении программы:

```
$ ln -s $(pwd)/target/debug/libcalculate.so /lib/libcalculate.so
```

Теперь, когда библиотечный файл находится в нужном месте, давайтесь попробуем скомпилировать с его помощью программу на С. Сначала удалите существующую функцию `solve`, показанную в листинге 3.2, из файла `calculator.c`. Новое содержимое файла показано в следующем ниже листинге.

Листинг 3.5 Программа-калькулятор на языке С без функции solve

```
#include <stdio.h>
#include <string.h>

int solve(char *line, int *solution); ← Предварительное объявление
                                         функции solve важно разместить перед функцией main.
                                         Этим компилятору С сообщается о том, что в конечном итоге функция, которая будет соответствовать заданной сигнатуре, будет определена.
                                         Указанное определение представляется путем связывания функции Rust solve.

int main() {
    char line[100];
    int solution;

    while (1) {
        printf("> ");
        if (fgets(line, 100, stdin) == NULL) {
            return 0;
        }

        if (solve(line, &solution)) {
            continue;
        }

        printf("%d\n", solution);
    }
    return 0;
}
```

Теперь должна появиться возможность скомпилировать программу на С и связать ее с библиотекой Rust. Аргумент `-lcalculate` сообщает компилятору о том, что ее нужно связать с библиотекой `libcalculate`:

```
$ gcc calculator.c -o bin -lcalculate
/usr/bin/ld: /tmp/ccwBuRCw.o: in function `main':
calculator.c:(.text+0x13f): undefined reference to `solve'
collect2: error: ld returned 1 exit status
```

Хм, не похоже, что это сработало. Ошибка говорит о том, что функция `solve` вызывается в функции `main`, но та не видит, где определена функция с именем `solve`. Следовательно, компоновщик С не может найти функцию Rust `solve`. Давайте посмотрим, как это исправить.

Пометка функции solve как допускающей связывание с языком С

Несмотря на то что компилятору Rust было поручено скомпилировать `calculate crate` как `cdylib`, он не экспортирует все функции и типы в формат, совместимый с языком С. Он экспортирует только те функ-

ции и типы, которые запрашиваются. Для того чтобы сделать функцию Rust доступной для вызова из C, требуется три шага. Нужно:

- отключить искажение имен;
- пометить функцию как публичную;
- сообщить компилятору Rust о том, что для этой функции нужно использовать правила С о вызовах.

В следующем ниже листинге показана надлежащая аннотированная функция.

Листинг 3.6 Функция solve на языке Rust, которую можно экспортовать как совместимую с языком С

```
#[no_mangle]
pub extern "C" fn solve(
    line: *const c_char, solution: *mut c_int) -> c_int {
    0
}
```

Здесь появляется ряд новых элементов, и все они имеют несколько иное назначение. Давайте рассмотрим их по очереди.

Первый из них, `#[no_mangle]`, является функциональной атрибутной макропомандой. Указанная макропоманда предписывает компилятору не искажать имя этой функции. Если вы много занимались разработкой на языке C++, то, возможно, знакомы с концепцией *искажения имен*. Если нет, то искажение имен относится к процедуре, которую компилятор использует для обеспечения уникальности имен функций и типов внутри системной библиотеки или исполняемого файла. В Unix-подобных системах исполняемые файлы и системные библиотеки не имеют пространств имен. Поэтому если в исполняемом файле определяется функция `solve`, то во всех используемых библиотеках и во всех файлах может быть только одна функция `solve`. Если в какой-либо библиотеке есть внутренняя функция с именем `solve`, то она будет конфликтовать с той, которую вы пытаетесь создать.

В целях преодоления этой проблемы компилятор Rust вносит дополнительную информацию в имя символов внутри нее, обеспечивая отсутствие наложения имен символов. Если оставить включенными искажение имен, то функции Rust `solve` будет дано имя наподобие `_zn9calculate5solve17h6ed7984646632de3fe`. Для целей изложения метод, который компилятор использует для создания этих уникальных имен, не имеет значения. Просто следует запомнить, что предсказывать эти искаженные имена очень сложно и громоздко. Следовательно, если ожидается, что будут вызываться какие-либо функции языка Rust из языка C, который не имеет понятия о схеме искажения имен в языке Rust, то необходимо использовать `no_mangle`, чтобы отключить искажение для этих конкретных функций.

Следующий новый фрагмент исходного кода, `pub`, – это ключевое слово, которое в языке Rust очень распространено. Он сообщает компилятору Rust о том, что символ должен быть экспортирован за пределы

лы модуля, в котором он определен. В языке Rust все символы по умолчанию являются приватными и не экспортируются. Для того чтобы экспортировать функцию или тип, надо добавить ключевое слово `pub` перед его определением, как это было сделано выше.

Наконец, идет выражение `extern "C"`, которое сообщает компилятору Rust о том, что нужно сгенерировать функцию `solve`, используя правила о вызовах, совместимые с С. По умолчанию правила компилятора Rust о вызовах не совсем совместимы с С. Компилятор Rust поддерживает целый ряд правил о вызовах, но наиболее часто используются дефолтные правила Rust, за которыми следует `"C"`.

На рис. 3.3 приводится детализация того, за что отвечает каждый из этих новых элементов синтаксиса.

Функциональная атрибутная макрокоманда

```
#[no_mangle]
```

Отключает искажение имени

Экспортирует функцию в публичный доступ

```
pub extern "C" fn solve
```

Использует правила С о вызовах

Рисунок 3.3 Анатомия объявления функции, совместимой с языком С

Получив возможность генерировать функцию, которая может вызываться из С, давайте сделаем так, чтобы библиотека Rust и приложение на С могли работать вместе.

Перекомпилирование программы С в динамическую библиотеку Rust

Можно начать с пересборки библиотеки Rust и перекомпиляции программы С:

```
$ cargo build
$ gcc calculator.c -o bin -lcalculate
```

Работает! Теперь давайте посмотрим, получится ли выполнить новую динамически связанную программу-калькулятор:

```
$ ./calculator
> 3 4 +
32686
> 4 10 +
32686
> 10 1000000 *
32686
> hello
32686
```

Ввиду того что данные читаются из памяти, в которую значения никогда не записывались, это число может отличаться при каждом запуске программы.

Итак, программа работает, но, похоже, была потеряна способность выполнять арифметические вычисления. Калькулятор все время выдает непредсказуемое число, потому что переменной `solution` ни разу не присваивается значение. Раз функция `solve` была заменена на функцию с ничего не делающей инструкцией `return 0`, это имеет смысл. Давайте напишем функцию `solve` на языке Rust! Прежде чем приступить к разбору строковых литералов, следует убедиться, что есть возможность обмениваться значениями между Rust и C, как положено. Поскольку функция `solve` принимает указатель на исходящий параметр `solution`, давайте попробуем записать в него значение. Так как для этой операции записи берется значение по указателю, операцию нужно будет обернуть в блок `unsafe`:

```
#[no_mangle]
pub extern "C" fn solve(
    line: *const c_char, solution: *mut c_int) -> c_int {
    if solution.is_null() {
        return 1;
    }

    unsafe {
        *solution = 1024; ←
    }
    0
}
```

Метод `is_null`.

`Синтаксис взятия значения по указателю
в языке Rust внутри блока unsafe такой же,
как и в языке C.`

Вспомните, что одна из причин, по которой Rust нуждается в том, чтобы взятия значений по указателям происходили внутри блоков `unsafe`, заключается в возможном наличии указателей на `null`. Прежде чем брать значения по ненадежным указателям, необходимо выполнить проверку на наличие указателей на `null`. В языке Rust взятие значения по указателю на `null` – это неопределенное поведение. Метод `is_null` встроен в примитивный тип указателей. Он не может отказывать или вызвать исключение, как это бывает при вызове метода для объекта `null` в Python или Java.

Если перекомпилировать исходный код Rust и выполнить исполняемый файл повторно, то теперь можно будет увидеть ожидаемые результаты:

```
$ cargo build
$ ./calculator
> 3 10 *
1024
> 1000 52 /
1024
> 1024 1 *
1024
```

Не то чтобы все они обязательно были правильными, но это все-таки результаты. Обратите внимание на отсутствие необходимости

перекомпилировать программу C, чтобы появились новые результаты. Ввиду того что `libcalculate.so` является динамической библиотекой, она загружается операционной системой при каждом выполнении программы `calculator`. Поэтому есть возможность обновлять исходный код Rust без необходимости повторного выполнения компилятора C.

Получив возможность писать в C, теперь нужно попытаться прочитать строковое значение, которое приходит из C. Строковые литералы C – это сплошные блоки, состоящие из смежно расположенных платформенно-специфичных типов символов с концевым символом `null`. Поскольку из строкового литерала C выполняется только чтение и его вообще невозможно изменить, можно создать только читаемый строковый срез типа `&str`, который указывает на ту же самую память, которая была создана в функции C `main`. Благодаря этому можно избежать двойного резервирования памяти под строковый литерал. Это одна из самых гибких возможностей использования многочисленных строковых типов в языке Rust. Если бы имелся только один тип `String`, то его можно было бы конструировать, резервируя под него память в куче, только в исходном коде Rust. Это означает, что всякий раз, когда требовалось бы использовать строковый литерал из C или любого другого языка, нужно было бы перераспределять под него память, что приводило бы к бесполезной трате памяти программы и потере времени.

Создание строковых срезов из ненадежных входных данных соединено с небольшими издержками; прежде чем их конструировать, приходится выполнять их проверку на соответствие кодировке UTF-8. В данной кодировке находятся все строковые литералы Rust, учитывая, что все конструкторы строковых литералов выполняют эту проверку либо являются незащищенными (`unsafe`) и ожидают, что разработчик применил какой-либо другой метод валидации. Поскольку у строковых литералов C кодировка UTF-8 может отсутствовать, эта проверка будет выполняться при их конструировании.

Нужно включить еще одну инструкцию `use`, чтобы ввести тип Rust под названием `CStr`. Тип `CStr` представляет строковый литерал C, который заимствует память у C. Вспомните расположение строкового литерала в памяти: это размещаемый в стеке массив типа `char`. Rust никогда не принимает это значение во владение, потому что если бы он попытался его высвободить, то память была бы высвобождена из стека программы C. А это невозможно и, вероятно, привело бы к ошибке сегментации. Вместо этого программа Rust просто заимствует только читаемый строковый литерал C, и все ссылки на него будут удалены, когда функция `solve` вернется. Таким образом, тип `CStr` используется как временное значение, чтобы облегчить создание значения типа `&str`:

```
use libc::{c_char, c_int};
use std::ffi::CStr;
```

```

#[no_mangle]
pub extern "C" fn solve(
    line: *const c_char, solution: *mut c_int) -> c_int {
    if line.is_null() || solution.is_null() {
        return 1;
    }
}

Функция from_ptr не защищена, так как источник вызова
несет ответственность за то, чтобы переданный указатель
не был равен null, а данные, на которые он указывает, соот-
ветствовали ожидаемой структуре строкового литерала С.

let c_str = unsafe { CStr::from_ptr(line) };
let r_str = match c_str.to_str() { ←
    Ok(s) => s,
    Err(e) => {
        eprintln!("UTF-8 Error: {}", e);
        return 1;
    },
};

println!("line: {}", r_str);

unsafe {
    *solution = 1024;
}
0
}

```

Выражение `match` в языке Rust является чрезвычайно мощным аналогом инструкции `switch`. В дополнение к сочетанию значений оно может выполнять операции деструктуризации, как это делается здесь. Функция `to_str` возвращает значение типа `Result`, которое является либо успешным значением `Ok`, либо значением `Err`. Для того чтобы извлечь вариант, нужно использовать `match`, как это сделано здесь.

Если выполнить программу-калькулятор сейчас, то можно будет увидеть, что строковый литерал `line` проникает в Rust:

```

$ cargo build
$ ./calculator
> 3 40 *
line: 3 40 *
1024

```

Даже появляется возможность удостовериться, что память под строковый литерал не перераспределяется, сравнив указатель `line`, который передается из С, с указателем на данные в `r_str`. Добавьте следующую ниже строку после создания `r_str`:

```
println!("r_str.as_ptr(): {:p}, line: {:p}", r_str.as_ptr(), line);
```

Местозаполнитель `{:p}` в форматном строковом литерале сообщает макрокоманде `println!` о необходимости отформатировать эти значения в виде адресов памяти:

```

$ cargo build
$ ./calculator
> 3 40 *
r_str.as_ptr(): 0x7fff78acb9b0, line: 0x7fff78acb9b0
line: 3 40 *

```

1024

Хорошо видно, что они оба имеют один и тот же адрес памяти, а значит, память в куче под строковый литерал `r_str` не перераспределялась; он полностью использует заимствованную память из исходного кода С. В простой программе это не будет иметь большого значения, но в более крупных программах с более крупным объемом передаваемых туда-сюда данных важно знать, что есть возможность эффективно делиться памятью между С и Rust.

Теперь, когда есть заготовка для обмена данными между исходным кодом на С и исходным кодом на Rust, можно перейти к решению задачи на языке Rust!

3.2.3 Решение арифметических выражений на языке Rust

В настоящее время имеется функция `solve` на языке Rust, которая выполняет большую работу с типами языка С, чего не делает обычная функция на языке Rust. Она преобразовывает строковый литерал С в строковый литерал Rust, пишет в указатель типа `int` в качестве исходящего параметра и сообщает о состоянии ошибки, возвращая значение типа `int`. В идеале желательно обособить исходный код, который выполняет эту работу по взаимодействию через интерфейс с внешними функциями (FFI) между С и Rust, от исходного кода, который содержит деловую логику. Если написать обычную функцию Rust, которая не будет иметь проблем с незащищенностью или интерфейсом с внешними функциями, то в дальнейшем ее можно было бы использовать для других целей. Например, вызывать из обычного исходного кода Rust или из других языков. Но если ее привязать непосредственно к функции `solve`, которая написана специально под работу с С, то ничего из этого нельзя будет сделать. Давайте начнем новую функцию в том же файле и дадим ей имя `evaluate`. Указанная функция будет принимать ссылку на строковый литерал и возвращать результат. Результат сообщает об успешном выполнении или отказе при вычислении арифметического выражения. Также давайте создадим для нее перечисление `Errgor`, которое пока будет оставаться пустым.

Листинг 3.7 Базовая функция evaluate

```
enum Errgor {
}

fn evaluate(problem: &str) -> Result<i32, Errgor> {
    Ok(1)
}
```

Функцию `solve` можно обновить под использование новой функции `evaluate`, чтобы получать результат, который она будет отправлять обратно в исходный код С. Сейчас также самое время, для того чтобы конвертировать Rust'овский тип `Result` в возвращаемый код типа `int`.

Листинг 3.8 Обновленная функция `solve`,
которая вызывает функцию `evaluate`

```
#[no_mangle]
pub extern "C" fn solve(
    line: *const c_char, solution: *mut c_int) -> c_int {
    if line.is_null() || solution.is_null() {
        return 1;
    }

    let c_str = unsafe { CStr::from_ptr(line) };
    let r_str = match c_str.to_str() {
        Ok(s) => s,
        Err(e) => {
            eprintln!("UTF-8 Error: {}", e);
            return 1;
        }
    };

    match evaluate(r_str) {
        Ok(value) => {
            unsafe {
                *solution = value as c_int;
            }
            0
        }
        Err(e) => {
            eprintln!("Error");
            1
        }
    }
}
```

Кроме того, необходимо убедиться, что программа по-прежнему функционирует, как положено. Поэтому давайте перекомпилируем библиотеку Rust и выполним калькулятор повторно. Вы должны увидеть, что вычисление всех арифметических выражений дает 1, так как именно это значение возвращается из функции `evaluate`:

```
$ cargo build
$ ./calculator
> 3 10 *
1
> 1000 52 /
1
> 1024 1 *
1
> hello
1
```

Разобравшись с этим вопросом, теперь давайте отложим функцию `solve` на некоторое время в сторону, сосредоточив все внимание на

реализации функции `evaluate`. Первое, что нужно сделать, – это разбить вводимые данные по пробельным символам и обследовать каждый фрагмент отдельно. Это легко достигается с помощью метода `.split`, доступного в языке Rust для значений типа `&str`:

```
fn evaluate(problem: &str) -> Result<i32, Error> {
    for term in problem.split(' ') {
        println!("{}", term);
    }

    Ok(1)
}
```

Если выполнить этот исходный код, то можно будет убедиться, что вводимые данные разбиваются по пробелам:

```
$ cargo build
$ ./calculator
> 3 4 *
3
4
*
```

```
1
```

Далее нужно определить, является ли рассматриваемый член арифметического выражения оператором, в каковом случае с ним нужно выполнить математические расчеты, или числом, в каковом случае нужно сохранить его где-нибудь для будущих математических расчетов. Пока что отложим вопрос о том, как «сохранить где-нибудь» до тех пор, пока не решим вопрос с правильным разбором входных данных. Для определения того, является ли строковый литерал в цикле оператором, можно задействовать выражение `match` способом, очень похожим на инструкцию `switch` в C. При этом можно добавить несколько простых инструкций печати, чтобы убедиться, что разбор членов арифметического выражения выполняется, как положено:

```
fn evaluate(problem: &str) -> Result<i32, Error> {
    for term in problem.split(' ') {
        match term {
            "+" => println!("ADD"),
            "-" => println!("SUB"),
            "*" => println!("MUL"),
            "/" => println!("DIV"),
            other => println!("OTHER {}", other),
        }
    }

    Ok(1)
}
```

Здесь благодаря использованию имени переменной вместо строкового литерала создается переменная с именем `other`.

Наличие переменной `other` внутри блока справа от «большой стрелки» (`=>`) в этой строке допустимо. `other` – это не ключевое слово, а просто имя создаваемой переменной. Блок `other` в выражении `match` будет выполнятьсь только в том случае, если никакие другие блоки не сочетаются с предоставленным значением. В данном случае блок `other` выполняется только в том случае, если значение `term` не равно ни одному из символов `+-*!`.

Если выполнить этот исходный код, то будет получено несколько неожиданных результатов:

```
$ cargo build
$ ./calculator
> 3 4 *
OTHER 3
OTHER 4
OTHER *
```

1

Если бы функция `evaluate` работала правильно, то можно было бы ожидать, что результат будет выглядеть следующим образом:

```
> 3 4 *
OTHER 3
OTHER 4
MUL
1
```

Но похоже, что программа неправильно выполняет разбор заключительного члена арифметического выражения; она разбирает оператор `*` только тогда, когда он не является заключительным членом выражения. Давайте добавим еще одну макрокоманду `println!`, на этот раз перед выражением `match`. До этого момента для печати всех значений использовался местозаполнитель `{}`. В нем задействуется форматировщик `Display`, который предназначен для вывода данных в форме, удобной конечному пользователю. Макрокоманда будет немного изменена, используя форматировщик `Display`, который обеспечивает более подробный вывод. Отладочное представление (`Debug`-представление) значения генерируется, используя местозаполнитель `{:?:}`:

```
fn evaluate(problem: &str) -> Result<i32, Error> {
    for term in problem.split(' ') {
        println!("Term - {:?}", term);
        match term {
            "+" => println!("ADD"),
            "-" => println!("SUB"),
            "*" => println!("MUL"),
            "/" => println!("DIV"),
            other => println!("OTHER {}", other),
        }
    }
    Ok(1)
}
```

Если выполнить программу еще раз, то проблема станет очевидной:

```
$ cargo build
$ ./calculator
> 3 4 *
Term - "3"
OTHER 3
Term - "4"
OTHER 4
Term - "*\n"
OTHER *
```

1

В последнем члене арифметического выражения есть замыкающий символ новой строки. Его можно удалить из строкового литерала `problem`, используя метод `.trim`, который удаляет начальные и замыкающие пробелы. Давайте посмотрим, даст ли добавление метода `.trim` ожидаемый результат. Теперь функция `evaluate` должна выглядеть следующим образом:

```
fn evaluate(problem: &str) -> Result<i32, Error> {
    for term in problem.trim().split(' ') {
        match term {
            "+" => println!("ADD"),
            "-" => println!("SUB"),
            "*" => println!("MUL"),
            "/" => println!("DIV"),
            other => println!("OTHER {}", other),
        }
    }
    Ok(1)
}
```

И вот результат:

```
$ cargo build
$ ./calculator
> 3 4 *
OTHER 3
OTHER 4
MUL
1
```

Поскольку на входном строковом литерале используется несколько вложенных методов, давайте быстро проверим, по-прежнему ли используется заимствованная память из стека С. Вспомните, что, как было подтверждено ранее, значение типа `&str`, которое передается в функцию `evaluate`, является коллективной памятью из стека С, и под это значение память в языке Rust не перераспределяется. Для получения адреса в памяти для `problem` и `term` можно применить форматировщик `{:p}` и метод `.as_ptr`:

```

fn evaluate(problem: &str) -> Result<i32, Error> {
    println!("problem: {:?}", problem.as_ptr());

    for term in problem.trim().split(' ') {
        println!("term: {:?} - {:?}", term.as_ptr(), term);
        match term {
            "+" => println!("ADD"),
            "-" => println!("SUB"),
            "*" => println!("MUL"),
            "/" => println!("DIV"),
            other => println!("OTHER {}", other),
        }
    }

    Ok(1)
}

```

Если память все еще используется из стека С коллективно, то `problem` и первое значение `term` должны указывать на одно и то же место в памяти, а последующие значения должны быть смещены на количество символов в подстроке. Выполнение этого действия подтверждает гипотезу о том, что память по-прежнему используется коллективно с C:

```

$ cargo build
$ ./calculator
> 3 4 *
problem: 0x7ffc117917b0
term : 0x7ffc117917b0
OTHER 3
term : 0x7ffc117917b2
OTHER 4
term : 0x7ffc117917b4
MUL
1

```

На вашем компьютере указанные в выводе адреса будут отличаться и могут меняться при каждом выполнении программы.

Адрес в памяти для `term` и `problem` одинаков, поэтому память для строковых буферов по-прежнему используется коллективно.

Объем памяти изменился на 2 байта: один байт для символа 3 и еще один байт для символа пробела.

Память по-прежнему используется коллективно! Память под строковый литерал из стека С ни разу не перераспределялась. Поскольку значение внутри строкового буфера изменять не нужно, а только просматриваемую часть строкового буфера, перераспределять под него память ни разу не пришлось. С типом `&str` в языке Rust можно выполнять столько операций на подстроках, сколько потребуется, и никогда не нужно перераспределять память. Эта способность является огромным преимуществом в части экономии памяти и времени. Хранение большого количества копий одних и тех же данных не отличается эффективностью и требует времени на перераспределение памяти и копирование строковых буферов, которые будут использоваться только один раз.

Далее нужно взять члены арифметического выражения, которые не являются операторами, и попытаться выполнить их разбор как целых чисел. Это делается с помощью метода `.parse`, доступного для

строковых литералов. Метод `.parse` является обобщенным по типу возвращаемого значения, то есть он может возвращать тип `int` разных размеров, число с плавающей точкой или большое число других типов. Методу `parse` нужно сообщить желаемый тип возвращаемого значения, который будет определять используемую им логику синтаксико-структурного разбора. Также нужно добавить вариант в перечисление `Error`, чтобы учесть возможный отказ метода `.parse`:

```
enum Error {
    InvalidNumber,
}

fn evaluate(problem: &str) -> Result<i32, Error> {
    for term in problem.trim().split(' ') {
        match term {
            "+" => println!("ADD"),
            "-" => println!("SUB"),
            "*" => println!("MUL"),
            "/" => println!("DIV"),
            other => match other.parse::<i32>() {
                Ok(value) => println!("NUM {}", value),
                Err(_) => return Err(Error::InvalidNumber),
            }
        }
    }
    Ok(1)
}
```

Выполнение этого исходного кода не приводит к неожиданностям:

```
$ cargo build
$ ./calculator
> 3 4 *
NUM 3
NUM 4
MUL
1
> 3 4 hello
NUM 3
NUM 4
Error
```

На данном этапе можно приступить к обследованию вопроса о том, как начать выполнять арифметические расчеты. Поскольку калькулятор выполняет разбор выражений в формате обратной польской нотации (RPN), нужна простая стековая структура данных, реализованная поверх двусторонней очереди. Стандартная библиотека Rust предоставляет двустороннюю очередь в виде типа `VecDeque`. Тип `VecDeque` – это двусторонняя очередь, поддерживаемая стандартным динамически расширяемым массивом `Vec`. Главное различие между более общим `Vec` и `VecDeque` заключается в том, что тип `VecDeque` предоставляет

ет двусторонние операции, такие как `push_front`, `push_back`, `pop_front` и `pop_back`. Для сравнения, тип `Vec` предоставляет только методы `push` и `pop`, которые обеспечивают порядок поступления «первым пришел – первым вышел» (FIFO). Ввиду того, что реализуется стек, нужно использовать методы `push_front` и `pop_front` из типа `VecDeque`, чтобы обеспечить порядок поступления «последним пришел – первым вышел» (LIFO). Для того чтобы обеспечить функциональность, соответствующую потребностям решателя в обратнойпольской нотации, будет создан тип-оболочка вокруг типа `VecDeque`. Этот тип называется `RpnStack`. Кроме того, поскольку тип `VecDeque` используется не так часто, как тип `Vec`, его нужно будет импортировать из стандартной библиотеки явным образом:

```
use std::collections::VecDeque;

#[derive(Debug)] //
struct RpnStack {
    stack: VecDeque<i32>,
}
```

`#[derive]` – это макрокоманда, которая поручает компилятору сгенерировать код для структуры или перечисления. В данном случае это реализация общей черты `Debug`, которая позволяет распечатывать `RpnStack`, используя представленный ранее форматировщик `Debug`. Хотя этот исходный код можно написать вручную, проще (в особенностях для типов с большим количеством полей) позволить компилятору сгенерировать его автоматически.

Давайте добавим несколько методов, чтобы выполнять стандартные операции `push` и `pop` на стеке: они соответственно добавляют новое число в начало стека и удаляют верхнее число из стека. Также будет добавлен вариант перечисления `Errgor`, чтобы отметить ошибку выталкивания из пустого стека:

```
enum Errgor {
    InvalidNumber,
    PopFromEmptyStack,
}

impl RpnStack {
    fn new() -> RpnStack {
        RpnStack {
            stack: VecDeque::new(),
        }
    }

    fn push(&mut self, value: i32) {
        self.stack.push_front(value);
    }

    fn pop(&mut self) -> Result<i32, Errgor> {
        match self.stack.pop_front() {
```

В блоке `impl` размещаются методы для структуры или перечисления.

Для создания экземпляра типа принято писать новый метод, который принимает все необходимые параметры. В языке Rust нет поддержки функций-конструкторов, как в языке C++ или Java; функция-конструктор – это просто обычная функция.

Методы, которые принимают параметр `self`, оперируют на отдельном экземпляре типа.

```
        Some(value) => Ok(value),
        None => Err(Errgog::PopFromEmptyStack),
    }
}
}
```

Блоки `impl` содержат методы, которые могут вызываться на заданном типе. Если вы пришли из таких языков, как Python или Java, в которых определения функций находятся в том же блоке, что и определение класса, это может показаться странным, но обеспечиваемая наличием отдельных блоков `impl` гибкость того стоит.

Обратите внимание на наличие в методах `push` и `pop` параметра `&mut self` и на отсутствие у `new`. `push` и `pop` – это *методы*, которые оперируют на конкретных экземплярах типа `RpnStack`, тогда как `new` – это *функция*, которая не принимает экземпляр на входе. Функции в блоках `impl` аналогичны статическим методам в Java или методам классов в Python. Блоки `impl` могут содержать как методы, так и функции; единственное различие заключается в наличии или отсутствии начального параметра `self`, аналогично методам Python, которые имеют начальный параметр `self`. В таких языках, как Java, JavaScript, Ruby и C++, переменная `self` или `this` бывает доступна в методах, но она не помечается как явный параметр. В языке Rust он требуется из-за явных правил языка Rust касательно мутации и контроля за владением. Параметры `self` могут принимать многочисленные формы: это могут быть вла-деемые значения `self`, немутируемые ссылки (`&self`) или, как видно здесь, мутируемые ссылки `self` (`&mut self`). Параметр `&mut self` не-обходи-м для обоих методов, так как они оба изменяют поле `stack` эк-земпляра типа `RpnStack`. Методы `push` или `pop` можно вызвать только в том случае, если есть мутируемая ссылка на экземпляр типа `RpnStack`.

С помощью этих методов можно будет реализовать функцию `evaluate`. Можно начать с заталкивания целых значений в стек и их распечатки позже. Кроме того, вместо того чтобы всегда возвращать `1`, можно начать возвращать верхнее значение в стеке:

```
fn evaluate(problem: &str) -> Result<i32, Errgog> {
    let mut stack = RpnStack::new(); ← Синтаксис Type::function()
    for term in problem.trim().split(' ') { ← используется для того,
        match term { ← чтобы вызывать функцию,
            "+" => println!("ADD"),
            "-" => println!("SUB"),
            "*" => println!("MUL"),
            "/" => println!("DIV"),
            other => match other.parse() { ← связанную с типом.
                Ok(value) => { ← Явно намекать на то, что
                    stack.push(value); ← синтаксико-структурный
                    println!("STACK: {:?}", stack); ← разбор должен возвращать
                }, ← значение типа i32, больше
                Err(_) => return Err(Errgog::InvalidNumber), ← нет необходимости.
            }
        }
    }
}
```

Синтаксис `instance.`
`method()` ис-
пользуется для
метода на кон-
кретном экзем-
пляре типа.

```
let value = stack.pop()?;
Ok(value)
}
```

Вспомните, что оператор `?` досрочно возвращает ошибку из функции, если выражение, к которому он применяется, является вариантом `Err`. Метод `pop` возвращает ошибку, когда стек пуст, поэтому оператор `?` необходим для передачи этой возможной ошибки источнику вызова.

Обратите внимание, что больше не нужно явно указывать, что метод `parse` должен возвращать значение типа `i32`. Берется возвращенная переменная `value` и немедленно передается в метод `push`. Этот метод принимает на входе только значение типа `i32`, поэтому компилятор посчитает, что синтаксико-структурный разбор должен возвращать значение типа `i32`, чтобы быть допустимым. Компилятор Rust прилагает все усилия, чтобы избавлять от необходимости писать типы снова и снова.

Давайте посмотрим, работает ли стек, как положено:

```
$ cargo build
$ ./calculator
> 3 4 *
STACK: RpnStack { stack: [3] }
STACK: RpnStack { stack: [4, 3] }
MUL
4
> *
MUL
Err: 0
```

Теперь, когда есть числовое хранилище, появляется возможность реализовать сложение. Вспомните, что в арифметике обратнойпольской нотации нужно извлекать два значения из стека, складывать их и помещать результат обратно в стек:

```
fn evaluate(problem: &str) -> Result<i32, Err: 0> {
    let mut stack = RpnStack::new();

    for term in problem.trim().split(' ') {
        match term {
            "+" => {
                let y = stack.pop()?;
                let x = stack.pop?();
                stack.push(x + y);
            }
            "-" => println!("SUB"),
            "*" => println!("MUL"),
            "/" => println!("DIV"),
            other => match other.parse() {
                Ok(value) => stack.push(value),
                Err(_) => return Err(Err: 0::InvalidNumber),
            }
        }
    }
}
```

Стек организован в порядке поступления «последним пришел – первым вышел» (LIFO), поэтому верхний элемент в стеке является вторым элементом в выражении. Следовательно, нужно выталкивать их из стека в «обратном» порядке, беря `y`, а затем `x`. Результаты сложения не меняются, но попробуйте эти строки поменять местами при вычитании или делении.

```

let value = stack.pop()?;
Ok(value)
}

```

Если выполнить эту программу сейчас, то можно будет вычислять произвольно вложенные выражения сложения:

```

$ cargo build
$ ./calculator
>34+
7
> 100 300 + 200 +
600

```

Аналогичные реализации достаточно просто предоставить и для других операторов.

Листинг 3.9 Использование функции evaluate для всех четырех арифметических операций

```

fn evaluate(problem: &str) -> Result<i32, Error> {
    let mut stack = RpnStack::new();

    for term in problem.trim().split(' ') {
        match term {
            "+" => {
                let y = stack.pop()?;
                let x = stack.pop?;

                stack.push(x + y);
            }
            "-" => {
                let y = stack.pop()?;
                let x = stack.pop?;

                stack.push(x - y);
            }
            "*" => {
                let y = stack.pop()?;
                let x = stack.pop?;

                stack.push(x * y);
            }
            "/" => {
                let y = stack.pop()?;
                let x = stack.pop?;

                stack.push(x / y);
            }
            other => match other.parse() {
                Ok(value) => stack.push(value),
                Err(_) => return Err(Error::InvalidNumber),
            }
        }
    }
}

```

```

    }
}

let value = stack.pop()?;
Ok(value)
}

```

И похоже, программа работает, как положено:

```

$ cargo build
$ ./calculator
> 3 4 * 10 + 20 -
2
> 3 4 *
12
> 3 4 + 10 * 20 -
50
> 100 2 /
50
> 100 5 /
20
> /
Error

```

Программа близка к завершению. На данный момент самый большой пробел в функциональности связан с тем, что сообщения об ошибках не выводятся пользователю за пределами структуры `Error`. Такая ситуация не очень помогает; нужно попробовать распечатывать сообщение с конкретной информацией об ошибке. В функцию `solve` можно было бы добавить еще одно выражение `match`, чтобы проверять вариант ошибки, но этот метод далек от идеала. Он может выглядеть нормальным для небольшой программы, но что, если функция `evaluate` вызывается в нескольких местах и все они хотят регистрировать одно и то же сообщение об ошибке при ее возникновении? Необходимо централизовать сообщения об ошибках, которые генерируются структурой `Error`. Для этого стандартным образом принято использовать общую черту `Display`.

3.2.4 Общая черта `Display`

Общие черты в языке Rust очень похожи на интерфейсы в языке Java или Go или абстрактные классы в языке C++. Это определения функциональности, которые могут быть реализованы любым типом, вследствие чего с этими типами можно обращаться схожим образом. Например, в стандартной библиотеке все числовые типы реализуют общую черту `Add`, указывая на возможность выполнения на них операции сложения. Ниже будет рассмотрена общая черта `Display`, которую вы использовали все это время, даже не осознавая! Реализация общей черты `Display` применялась при каждом использовании макрокоманды `println!` и местозаполнителя `{}`, чтобы напечатать значение.

Давайте посмотрим, как можно было бы написать программу «Hello world!», используя общую черту `Display`.

Листинг 3.10 «Hello World!» с общей чертой Display

Всякий раз, когда реализуется общая черта `Display`, необходимо реализовывать функцию `fmt` именно с такой сигнатурой. Также можно было бы импортировать `Result` из пакета `fmt`, чтобы сократить тип возвращаемого значения, но он часто конфликтует с обычным типом `Result`, поэтому обычно он не импортируется. Вместо этого используется полный путь.

```
use std::fmt::{Display, Formatter};
```

`struct Hello {}`

`impl Display for Hello {` Реализации общих черт всегда записываются как `impl` Общая_чертя `for` Тип.

```
    fn fmt(&self, f: &mut Formatter) -> std::fmt::Result {
        write!(f, "Hello world!")
    }
}
```

В макрокоманде `write!` используется тот же форматный строковый литерал с синтаксисом местозаполнителя, что и в макрокомандах `println!`/`format!` и подобных им. Указанная макрокоманда возвращает `std::fmt::Result`, поэтому точка с запятой в этой строке опускается, чтобы обеспечить возвращение результата из функции `fmt`.

```
fn main() {
    let x = Hello {};
    println!("{}", x);
}
```

Используется тот же местозаполнитель `{}`, который применялся на протяжении всей книги. Единственное отличие заключается в том, что теперь его можно использовать для своего собственного типа, а не только для стандартных библиотечных типов.

Реализация общей черты `Display` для конкретно-прикладных типов выполняется очень просто. Помимо сигнатуры типа для функции `fmt`, это, по сути, просто замена макрокоманды `println!` на макрокоманду `write!` и добавление начального аргумента `f`. Указанный аргумент представляет структуру `Formatter`, которая может содержать `stdout` (для `println!`), `stderr` (для `eprintln!`) или строковый литерал (для `format!`).

Теперь давайте реализуем общую черту `Display` для типа `Error`.

Листинг 3.11 Реализация общей черты Display для типа Error

```
use std::fmt::{Display, Formatter};
```

```
enum Error {
    InvalidNumber,
    PopFromEmptyStack,
}
```

```
impl Display for Error {
    fn fmt(&self, f: &mut Formatter) -> std::fmt::Result {
        match self {
            Error::InvalidNumber => write!(
                f, "Not a valid number or operator"),
            Error::PopFromEmptyStack => write!(
                f, "Tried to operate on empty stack"),
        }
    }
}
```

ПРИМЕЧАНИЕ. Для типов `Err` настоятельно рекомендуется предоставлять реализацию общей черты `Display`.

Затем можно обновить функцию `solve`, чтобы воспользоваться преимуществами этой новой реализации общей черты `Display`.

Листинг 3.12 Функция solve обновлена распечаткой сообщений об ошибках

```
#[no_mangle]
pub extern "C" fn solve(
    line: *const c_char, solution: *mut c_int) -> c_int {
    if line.is_null() || solution.is_null() {
        return 1;
    }

    let c_str = unsafe { CStr::from_ptr(line) };
    let r_str = match c_str.to_str() {
        Ok(s) => s,
        Err(e) => {
            eprintln!("UTF-8 Error: {}", e);
            return 1;
        }
    };

    match evaluate(r_str) {
        Ok(value) => {
            unsafe {
                *solution = value as c_int;
            }
            0
        }
        Err(e) => {
            eprintln!("Error: {}", e);
            1
        }
    }
}
```

Данная строка – единственная, которую нужно изменить. Значение ошибки распечатывается при помощи местозаполнителя {}.

Получилось! Теперь есть программа-калькулятор, которая обменивается данными с пользователем на C, решает уравнение на Rust и отправляет результат обратно в C. Для справки, в следующем ниже листинге приведено полное содержимое файла `lib.rs` пакета `calculate`. Библиотеку-калькулятор можно использовать из интерфейса с внешними функциями C (C FFI) или обычного исходного кода Rust.

Листинг 3.13 Библиотека-калькулятор

```
use libc::{c_char, c_int};
use std::collections::VecDeque;
use std::ffi::CStr;
use std::fmt::{Display, Formatter};
```

```

#[no_mangle]
pub extern "C" fn solve(
    line: *const c_char, solution: *mut c_int) -> c_int {
    if line.is_null() || solution.is_null() {
        return 1;
    }

    let c_str = unsafe { CStr::from_ptr(line) };
    let r_str = match c_str.to_str() {
        Ok(s) => s,
        Err(e) => {
            eprintln!("UTF-8 Error: {}", e);
            return 1;
        }
    };

    match evaluate(r_str) {
        Ok(value) => {
            unsafe {
                *solution = value as c_int;
            }
            0
        }
        Err(e) => {
            eprintln!("Error: {}", e);
            1
        }
    }
}

enum Error {
    InvalidNumber,
    PopFromEmptyStack,
}

impl Display for Error {
    fn fmt(&self, f: &mut Formatter) -> std::fmt::Result {
        match self {
            Error::InvalidNumber => write!(f, "Not a valid number or operator"),
            Error::PopFromEmptyStack => write!(f, "Tried to operate on empty stack"),
        }
    }
}

#[derive(Debug)]
struct RpnStack {
    stack: VecDeque<i32>,
}

impl RpnStack {

```

```

fn new() -> RpnStack {
    RpnStack {
        stack: VecDeque::new(),
    }
}

fn push(&mut self, value: i32) {
    self.stack.push_front(value);
}

fn pop(&mut self) -> Result<i32, Error> {
    match self.stack.pop_front() {
        Some(value) => Ok(value),
        None => Err(Error::PopFromEmptyStack),
    }
}

fn evaluate(problem: &str) -> Result<i32, Error> {
    let mut stack = RpnStack::new();

    for term in problem.trim().split(' ') {
        match term {
            "+" => {
                let y = stack.pop()?;
                let x = stack.pop()?;
                stack.push(x + y);
            }
            "-" => {
                let y = stack.pop()?;
                let x = stack.pop()?;
                stack.push(x - y);
            }
            "*" => {
                let y = stack.pop()?;
                let x = stack.pop()?;
                stack.push(x * y);
            }
            "/" => {
                let y = stack.pop()?;
                let x = stack.pop()?;
                stack.push(x / y);
            }
            other => match other.parse() {
                Ok(value) => stack.push(value),
                Err(_) => return Err(Error::InvalidNumber),
            },
        }
    }

    let value = stack.pop()?;
    Ok(value)
}

```

Давайте попробуем ее выполнить, чтобы убедиться, что все работает в сочетании с новым исходным кодом обработки ошибок:

```
$ cargo build
$ ./calculator
> 3 4 *
12
> 19 8 /
2
> hello
Error: Not a valid number or operator
> 4 *
Error: Tried to operate on empty stack
> 30 2 -
28
> 30 4 +
34
> 4
4
```

Все работает именно так, как задумано.

На рис. 3.4 показан график времен жизни для программы-калькулятора, работающей через интерфейс с внешними функциями (FFI).

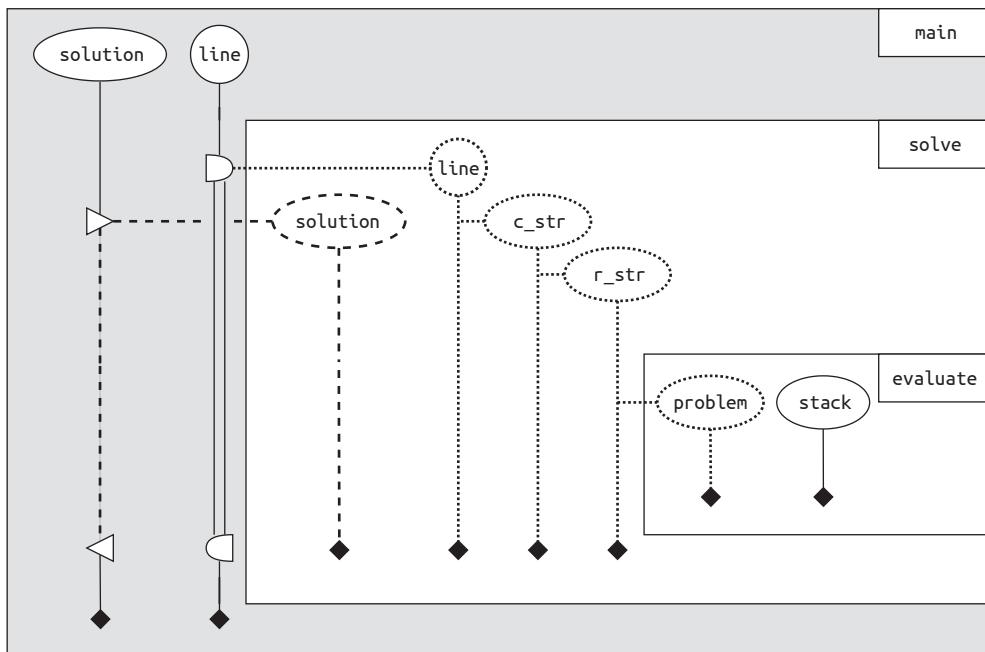


Рисунок 3.4 График времен жизни для программы-калькулятора, работающей через интерфейс с внешними функциями FFI

Краткий итог

- Незащищенные (`unsafe`) функции и блоки могут использоваться для выполнения операций, которые запрещены в обычном исходном коде Rust, например для взятия значений по обычновенным указателям.
- `unsafe` означает, что компилятор не проверяет некоторые правила, и разработчик несет ответственность за соблюдение правил защищенности памяти Rust.
- Существует возможность писать обычные функции Rust со своей деловой логикой и функции-обертки, которые обрабатывают обмен данными с C через границы интерфейса с внешними функциями (FFI).
- Пакет Rust `cdylib` можно связывать с обычной программой на C, а функции Rust, аннотированные для работы через интерфейс с внешними функциями (FFI), можно вызывать из C.
- Тип `CStr` используется для преобразования строкового литерала языка C с концевым символом `null` в тип `&str` языка Rust.
- Обычные типы Rust, такие как `&str`, могут обеспечивать защищенные и простые в использовании абстракции коллективной памяти с использованием исходного кода C.
- Тип `&str` не нуждается в перераспределении памяти для выполнения операций на подстроках.
- Выражения `match` используются подобно инструкциям C `switch` для выполнения нескольких операций сравнения с одним значением.
- Отладочное форматирование (`Debug`) может предоставлять информацию, например, о скрытых экранирующих последовательностях внутри строкового литерала или о внутренних элементах структуры данных.
- Общая черта `Display` используется для печати значений с местозаполнителем `{}`.
- Реализация общей черты `Display` для типов ошибки считается образцом наилучшей практики.

Продвинутый интерфейс с внешними функциями

Эта глава охватывает следующие ниже темы:

- создание модуля расширения для веб-сервера NGINX с помощью языка Rust;
- генерирование привязок языка Rust к существующей кодовой базе на языке C;
- использование встроенного в С менеджера памяти из языка Rust;
- коллективное использование функций между пакетами языка Rust.

В предыдущей главе был рассмотрен простой пример вызова функции, написанной на языке Rust, из исходного кода на языке C. При этом из исходного кода на языке Rust использовалось одиночное строковое значение, размещенное в стеке C, но исходный код Rust не отправлял в исходный код C никаких значений, размещенных в куче, и не вызывал никаких функций C. Поверхность API программы-калькулятора на C была очень мала, и поэтому добавить в нее Rust было довольно просто. Эта глава является продолжением примера с калькулятором, приведенного в предыдущей главе. Вместо добавления функции калькулятора в простое консольное приложение будет написан модуль расширения для веб-сервера NGINX, который отвечает на HTTP-запросы результатами вычислений. Эта глава не предназначена в качестве об-

щего руководства по написанию расширений для веб-сервера NGINX; веб-сервер NGINX – это просто типичный представитель достаточно сложной кодовой базы на языке C, к которой будет добавлено немного исходного кода на языке Rust.

Цель состоит в создании модуля для веб-сервера NGINX, который решает арифметические выражения в формате обратнойпольской нотации (RPN), используя библиотеку `calculate`, которая была создана в главе 3.

Модуль должен уметь читать выражения из тела запроса POST. Поэтому если исходить из того, что веб-сервер NGINX работает на порту 8080, то его можно будет использовать следующим образом:

```
$ curl -X POST -d '3 4 +' http://localhost:8080/calculate  
7  
$ curl -X POST -d '3 4 * 2 -' http://localhost:8080/calculate  
10
```

Веб-сервер NGINX – это популярный балансировщик HTTP-нагрузки и обратный прокси-сервер, написанный на языке C. В настоящее время он используется на более чем 400 млн веб-сайтов во всей Всемирной паутине. В веб-сервере NGINX есть модульная система, позволяющая разработчикам писать исходный код на языке C, который может управлять его поведением или добавлять совершенно новые функциональные возможности. Этот C-интерфейс будет использован как из языка C, так и из языка Rust, чтобы создать обработчик HTTP, задействующий тот же калькулятор в формате обратнойпольской нотации, который был создан в главе 3. В главе 3 был представлен интерфейс STDIN/STDOUT для использования калькулятора, но в этой главе будет создан HTTP-интерфейс. Поскольку веб-сервер NGINX намного сложнее, чем описанная в главе 3 программа STDIN/STDOUT, для выполнения поставленной задачи нужно будет предпринять ряд шагов:

- 1 Скачать исходный код веб-сервера NGINX.
- 2 Написать склеивающий исходный код на C, который соединяет веб-сервер NGINX с языком Rust.
- 3 Связать исходный код модуля на C с функцией-обработчиком HTTP на Rust.
- 4 Извлечь сведения о запросе из структуры запроса к веб-серверу NGINX.
- 5 Вызвать библиотеку-калькулятор, о которой писалось в главе 3.
- 6 Вернуть результат вычисления в HTTP-ответе.

4.1 Скачивание исходного кода веб-сервера NGINX

Скачивание исходного кода веб-сервера NGINX – самый простой из всех шагов. Здесь будет использована версия 1.19.3 веб-сервера, которую можно бесплатно скачать с веб-сайта NGINX (<https://nginx.org>). Он представлен в виде архивного файла в сжатом виде, и его можем

легко извлечь после скачивания. Давайте также создадим новый пакетный каталог с помощью менеджера пакетов Cargo, чтобы поместить все эти файлы в него:

```
$ cargo new --lib ngx_http_calculator_rs
$ cd ngx_http_calculator_rs
$ wget https://nginx.org/download/nginx-1.19.3.tar.gz
$ tar -xvf nginx-1.19.3.tar.gz
```

Теперь все готово к тому, чтобы приступить к написанию исходного кода!

ПРИМЕЧАНИЕ. В следующих далее разделах содержится большое количество путей к файлам и команд. Давайте будем исходить из того, что все пути к файлам относятся к пакету `ngx_http_calculator_rs`, который только что был создан, а также из того, что все сеансы работы с командной оболочкой начинаются в этом каталоге, и, при необходимости, сеанс работы с командной оболочкой будет в самом начале содержать команду `cd`, указывающую, в каком подкаталоге следует выполнять команды.

4.2 Создание модуля для веб-сервера NGINX

Веб-сервер NGINX имеет большую и сложную поверхность С-интерфейса, и эта глава не предназначена для того, чтобы быть руководством по написанию плагина для данного веб-сервера. В этом разделе представлен стартовый исходный код модуля C для веб-сервера NGINX, который вызывает функцию Rust, чтобы та предоставила обработчик HTTP.

Веб-сервер NGINX позволяет разработчикам создавать динамические модули, которые загружаются в память с помощью двоичного файла веб-сервера после его запуска. В этом примере будет создан динамический модуль, который позволит обновлять модуль путем перекомпиляции исходного кода Rust без необходимости каждый раз перекомпилировать весь двоичный файл веб-сервера. Для того чтобы создать новый динамический модуль, давайте начнем с создания каталога под названием `module` и размещения в нем двух новых файлов. Первый файл, `module/config`, должен выглядеть следующим образом:

```
ngx_module_type=HTTP
ngx_module_name=ngx_http_calculator
ngx_module_srcs="$ngx_addon_dir/ngx_http_calculator.c"
ngx_module_libs=""

auto/module

ngx_addon_name=$ngx_module_name
```

Этот файл представляет собой скрипт командной оболочки, который устанавливает некоторые переменные среды, используемые веб-

сервером NGINX в своих конкретно-прикладных шагах сборки для модулей. Переменные, чьи значения предположительно будут установлены в этом файле, задокументированы на странице веб-сервера NGINX (<https://mng.bz/oKWp>).

Прочитав переменные, заданные в скрипте командной оболочки, вы, возможно, смогли угадать путь ко второму файлу, который будет создан далее. Продолжите и создайте файл `module/ngx_http_calculator.c`. Этот файл исходного кода на С задает некоторые глобальные переменные и предоставляет функции, необходимые для инициализации модуля NGINX. Указанные переменные и функции можно написать на языке Rust, что позволяет отказаться от написания исходного кода на С. Однако эти функции инициализации просты и в значительной степени опираются на макрокоманды предобработчика, которые нелегко переложить на язык Rust. В данной главе их перенос на язык Rust не обсуждается, но это может стать хорошим упражнением для самостоятельной работы!

Добавьте следующее ниже содержимое в файл `module/ngx_http_calculator.c`:

Листинг 4.1 Стартовый исходный код модуля NGINX

```
#include <ngx_config.h>
#include <ngx_core.h>
#include <ngx_http.h>

typedef struct {
    ngx_flag_t enable_calculation;      Предварительное объявление функции, которая будет определена в библиотеке Rust.
} ngx_http_calculator_loc_conf_t;

ngx_int_t ngx_http_calculator_handler(ngx_http_request_t *r);           ◀

static void *ngx_http_calculator_create_loc_conf(ngx_conf_t *cf);
static char *ngx_http_calculator_merge_loc_conf(
    ngx_conf_t *cf, void *parent, void *child);                           ◀

Этот блок позволяет написать calculate on в конфигурационном файле веб-сервера NGINX, чтобы сообщить веб-серверу о том, что эта библиотека должна обрабатывать специфичные HTTP-запросы.                                ◀

static ngx_command_t ngx_http_calculator_commands[] = {
    {ngx_string("calculate"),
        NGX_HTTP_LOC_CONF | NGX_CONF_FLAG,
        ngx_conf_set_flag_slot, NGX_HTTP_LOC_CONF_OFFSET,
        offsetof(ngx_http_calculator_loc_conf_t,
            enable_calculation), NULL},          ◀
    ngx_null_command};                  Переменная ngx_http_calculator совпадает с именем модуля в файле module/config. Она позволяет веб-серверу NGINX узнавать, какой символ загружать из динамической библиотеки при открытии модуля.                                ◀

static ngx_http_module_t ngx_http_calculator_module_ctx = {           ◀
    NULL, NULL, NULL, NULL, NULL, NULL,
```

```

    ngx_http_calculator_create_loc_conf,
    ngx_http_calculator_merge_loc_conf};

ngx_module_t ngx_http_calculator = {
    NGX_MODULE_V1,
    &ngx_http_calculator_module_ctx,
    ngx_http_calculator_commands,
    NGX_HTTP_MODULE,
    NULL, NULL, NULL, NULL,
    NULL, NULL, NGX_MODULE_V1_PADDING};

```

←

→

Эта макрокоманда версии V1 позволяет веб-серверу NGINX немного версионировать свой С-интерфейс. В настоящее время для этого API существует только версия V1, и на данный момент нужно включать константу V1 в начало модуля и макрокоманду выравнивания по V1 (V1 padding) в его конец.

Эта макрокоманда сообщает веб-серверу NGINX о том, что модуль будет управлять подсистемой HTTP. Веб-сервер имеет ряд подсистем, и многие из них имеют перехватчики для модулей.

```

static void *ngx_http_calculator_create_loc_conf(ngx_conf_t *cf)
{
    ngx_http_calculator_loc_conf_t *conf;

    conf = ngx_pcalloc(cf->pool, sizeof(
        ngx_http_calculator_loc_conf_t));
    if (conf == NULL) {
        return NULL;
    }

    conf->enable_calculation = NGX_CONF_UNSET;

    return conf;
}

static char *ngx_http_calculator_merge_loc_conf(
    ngx_conf_t *cf, void *parent, void *child)
{
    ngx_http_calculator_loc_conf_t *prev = parent;
    ngx_http_calculator_loc_conf_t *conf = child;

    ngx_conf_merge_value(conf->enable_calculation,
        prev->enable_calculation, 0);

    if (conf->enable_calculation) {
        ngx_http_core_loc_conf_t *clcf;

        clcf = ngx_http_conf_get_module_loc_conf(
            cf, ngx_http_core_module);
        clcf->handler = ngx_http_calculator_handler;
    }

    return NGX_CONF_OK;
}

```

Сообщает веб-серверу NGINX о том, что при вызове обработчика HTTP нужно вызвать функцию Rust. Если в конфигурации веб-сервера указан аргумент calculate_on, то функция-обрабочник HTTP устанавливается равной функции-обрабочнику на языке Rust.

Пусть вас не пугает большое количество значений `NULL`! Система модулей веб-сервера NGINX содержит большое количество перехватчиков, и многие из них не требуются для решения задачи, которую вы пытаетесь решить.

Теперь, когда есть исходный код на С, необходимый для модуля NGINX, давайте попробуем его скомпилировать! Перейдите в созданный ранее каталог исходного кода веб-сервера и выполните скрипт `configure` с созданным ранее каталогом `module`:

```
$ cd nginx-1.19.3  
$ ./configure --add-dynamic-module=../module
```

Учитывая путь к `../module`, скрипт `configure` выполнит файл `../module/config`, чтобы сообщить процессу сборки метаданные о том, как он должен собирать модуль. Далее можно скомпилировать веб-сервер NGINX и прикладной модуль с помощью одной команды `make`:

```
$ cd nginx-1.19.3  
$ make -j16 build modules
```

Цель сборки `build` – главный исполняемый файл `nginx`, а `modules` представляет все сконфигурированные модули-плагины (такие как данный прикладной модуль). Эти модули производят много выходных данных и могут занимать некоторое время. Рекомендуется использовать опцию `-j` (которая расшифровывается как `jobs`, то есть заявки на обработку заданий) в `make` для распараллеливания сборки. На нашем компьютере использовалась опция `-j16`, так как процессор имеет 16 ядер.

После того как `make` завершит компиляцию модуля и двоичного файла веб-сервера NGINX, в выходном каталоге `objs`, куда процесс сборки NGINX помещает двоичные файлы и библиотеки после их сборки, должно появиться несколько новых файлов. Поиск исполняемых файлов в этом каталоге позволяет обнаружить два важных файла:

```
$ cd nginx-1.19.3  
$ find objs -executable -type f  
objs/nginx_http_calculator.so ←  
objs/nginx ←  
|  
| Сам двоичный файл  
| веб-сервера NGINX.
```

Файл динамической библиотеки для прикладного модуля. Он содержит определение переменной `ngx_http_calculator`, которая сообщает веб-серверу о том, что собственно нужно делать при загрузке модуля.

Теперь, когда есть скомпилированный веб-сервер NGINX и скомпилированный модуль, давайте попробуем запустить веб-сервер NGINX с загруженным модулем! Однако сначала веб-серверу нужен рабочий каталог для размещения своих временных файлов, конфигурационных файлов и журналов регистрации событий. Сейчас они будут созданы. Давайте назовем его `nginx-run`. В дополнение к папке верхнего уровня в ней должен быть подкаталог `logs`:

```
$ mkdir nginx-run  
$ mkdir nginx-run/logs
```

ПРИМЕЧАНИЕ. Во время исполнения веб-сервер NGINX будет использовать каталог `nginx-run` в качестве резервного. За исключением каталога `logs` и конфигурационного файла, беспокоиться о структуре этого каталога не следует.

Теперь создайте файл `nginx-1.19.3/nginx.conf` и добавьте в него следующее:

```
load_module ..../nginx-1.19.3/objs/ngx_http_calculator.so;
worker_processes 1;
daemon off;
error_log /dev/stderr info; ← Дает команду веб-серверу NGINX загрузить динамический модуль по указанному пути к файлу.

events {
    worker_connections 1024;
} ← Направляет информацию об ошибке непосредственно на консоль. Обычно NGINX проглатывает эту строку и добавляет ее в журнальные файлы регистрации событий. Хотя это и идеально подходит для производственных задач, но значительно усложняет отладку в реальном времени.

http {
    access_log /dev/stdout; ← Аналогичным образом направляет зарегистрированные запросы не в файл, а в STDOUT.

    server {
        listen      8080;

        location /calculate {
            calculate on; ← Веб-сервер NGINX сообщает, что запросы, маршрутизованные в /calculate, должны обрабатываться библиотекой calculate.
        }
    }
}
```

Теперь, когда есть конфигурационный файл для веб-сервера NGINX, давайте его запустим! В этой главе следующая ниже команда запуска экземпляра веб-сервера NGINX будет использоваться много раз:

```
$ ./nginx-1.19.3/objs/nginx -c nginx.conf -p ngx-run
nginx: [emerg] dlopen() "ngx_http_calculator.so" failed
(ngx_http_calculator.so: undefined symbol:
 ngx_http_calculator_handler)
in nginx.conf:1
```

Веб-сервер NGINX не запускается! Но почему? Неужели вся работа пошла насмарку? Все дело в том, что веб-серверу NGINX была сообщена только часть информации. В файле на C есть предварительное объявление, которое сообщает веб-серверу о том, что «в какой-то момент будет определена функция `ngx_http_calculator_handler`», но это определение еще нигде не было предоставлено. В следующем разделе будет рассмотрено создание этой функции на языке Rust и ее использование в существующем исходном коде на языке C.

4.3

Связывание языка C с языком Rust

В предыдущем разделе было написано предварительное объявление для HTTP-обработчика, которое выглядит следующим образом:

```
ngx_int_t ngx_http_calculator_handler(ngx_http_request_t *r);
```

И было выяснено, что указанную функцию нужно добавить позже в библиотеку Rust. Указанное выше объявление функции Слегко перекладывается в объявление функции Rust. Давайте взглянем:

```
#[no_mangle]
pub unsafe extern "C" fn ngx_http_calculator_handler(
    r: *mut ngx_http_request_t
) -> ngx_int_t {
    0
}
```

Эта функция должна существовать, чтобы ее можно было вызывать из веб-сервера NGINX, но сначала необходимо выполнить несколько действий. Возможно, вы заметили, что некоторые типы в сигнатуре функции начинаются с префикса `ngx_`. Эти типы выставляются наружу API-интерфейсом модулей веб-сервера NGINX в его заголовочных файлах. Обычно при написании модуля на C достаточно вставлять заголовочные файлы в собственный исходный код на C, и типы будут доступны для разработчика. Поскольку функция-обработчик пишется не на C, нужно немного поработать, чтобы перенести эти типы на язык Rust.

Понадобится сгенерировать *привязки* языка Rust к типам C в NGINX. Привязка – это, по сути дела, метаданные об API, который существует у библиотеки, реализованной на другом языке программирования¹. Это метаданные обо всех существующих в библиотеке функциях, типах и глобальных переменных, без их реализации. В главе 3 были созданы привязки языка C к библиотеке Rust `calculate` с C-совместимой функцией `solve`, являющейся составной частью этой библиотеки. Привязки не всегда существуют как составная часть самой библиотеки; они нередко предстаивают обособленными библиотеками. Например, библиотека `openssl` написана на C; для того чтобы напрямую взаимодействовать с функциями C из Rust, используется пакет Rust `openssl-sys`. Этот пакет предоставляет привязки языка Rust к библиотеке C `openssl`. На рис. 4.1 показано, как высокоуровневые привязки языка Rust в пакете `openssl` обращаются вниз к прямым привязкам в пакете `openssl-sys`, которые затем пересекают границу межъязыкового FFI-интерфейса (интерфейса с внешними функциями) и попадают в библиотеку C `openssl` (рис. 4.1).

Для создания этих привязок к C потребуется ввести новую концепцию языка Rust – *скрипты сборки*.

¹ Иными словами, привязки языка Rust – это интерфейсный слой, который позволяет исходному коду на языке Rust взаимодействовать с библиотеками, написанными на других языках программирования. – *Прим. перев.*

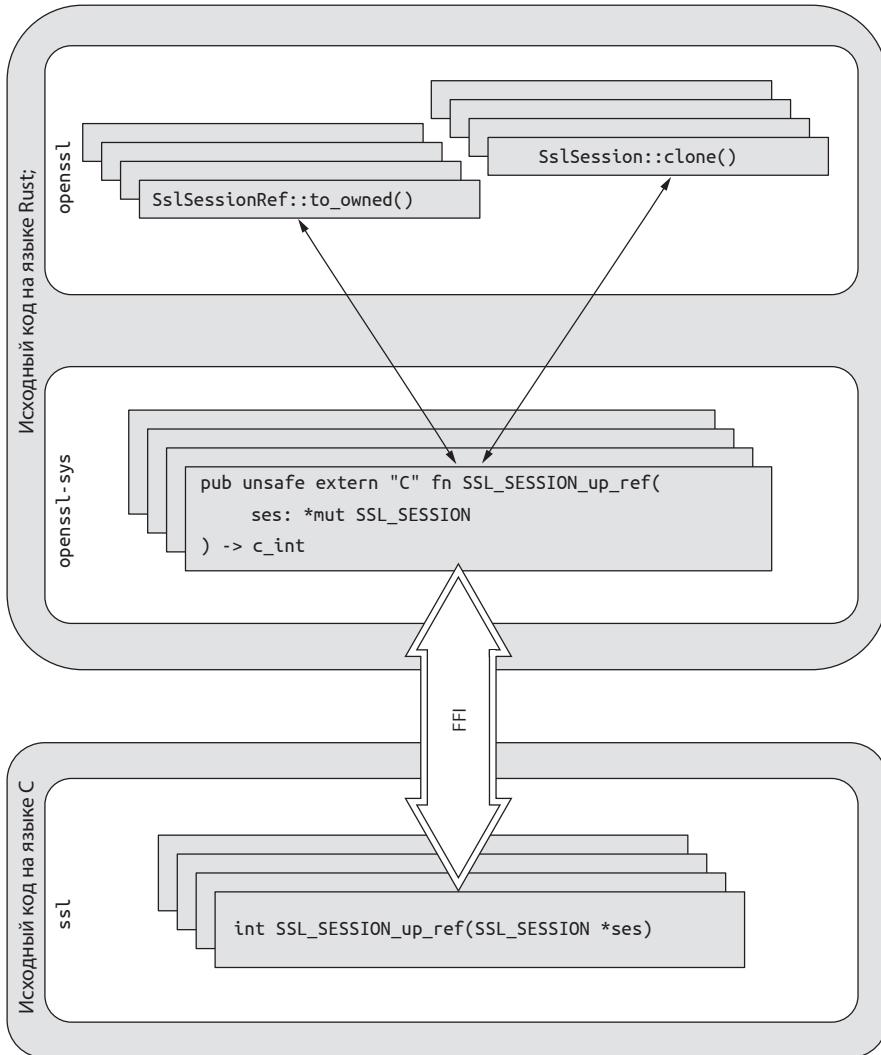


Рисунок 4.1 Высокоуровневые и низкоуровневые привязки языка Rust к библиотеке openssl на языке С

4.3.1 Скрипты сборки

Скрипт сборки – это небольшая программа на языке Rust, которая автоматически компилируется и исполняется непосредственно перед компиляцией более крупной библиотеки или исполняемого файла. Он может выполнять все, что делала бы обычная программа на языке Rust. Его полезность состоит в том, что он позволяет динамически генерировать исходный код на языке Rust во время сборки, который передается обратно в компилятор. Давайте ненадолго отложим в сторону обсуждение веб-сервера NGINX, чтобы рассмотреть упрощенный пример.

Представьте себе, что вы разрабатываете приложение для приветствий и хотите наделить вашу программу способностью приветствовать людей на нескольких языках. Однако вы не хотите выпускать одно масштабное приложение, содержащее все языки мира. Вы решаете, что хотели бы выполнить свою задачу, используя передаваемую компилятору переменную среды, чтобы определять язык, который приложение приветствия должно поддерживать. Вы предоставите соответствующим образом скомпилированные версии для разных регионов. Давайте начнем!

ПРИМЕЧАНИЕ. Этот пример создан для того, чтобы научиться строить скрипты, и является не самым лучшим способом интернационализации. В языке Rust доступны гораздо более оптимальные механизмы интернационализации. Поэтому не рекомендуем применять этот метод в реальной жизни.

С помощью менеджера пакетов Cargo создайте новый пакетный каталог (за пределами пакетного каталога веб-сервера NGINX):

```
$ cargo new build-script-test
```

ПРИМЕЧАНИЕ. В этом подразделе все пути указаны относительно корня нового пакетного каталога `build-script-test`.

Перейдите в свой новый каталог, создайте и откройте файл `build.rs`. По умолчанию менеджер пакетов Cargo будет искать файл под названием `build.rs` в корне пакетного каталога¹ и обрабатывать его как скрипт сборки, если он присутствует. Поскольку скрипты сборки выполняются как обычные программы Rust, им нужно предоставлять функцию `main`. Указанную функцию можно дополнить двумя наиболее важными заявками (jobs) на исполнение заданий, которые указанный скрипт сборки будет выполнять: читать переменную среды и писать в файл.

Листинг 4.2 Базовый скрипт сборки, который пишет данные в файл

```
use std::fs::File;
use std::io::Write; ← Импортирует общую черту Write, чтобы можно было вызывать метод file.write_all в последней строке функции main.

fn main() {
    let language = std::env::var("GREET_LANG").unwrap(); ← Функция std::env::var выполняет поиск значений переменных среды во время исполнения. Она возвращает экземпляр типа Option<String>, так как запрашиваемая переменная может быть не задана. Поэтому прежде чем использовать экземпляр типа Option, его нужно развернуть.
```

¹ Корень пакетного каталога в Rust состоит из папки верхнего уровня, содержащей файл `Cargo.toml`, который содержит метаданные пакета, зависимости и другие параметры конфигурации. Он также может содержать каталог `src`, в котором хранятся файлы исходного кода, такие как `main.rs` для двоичных файлов или `lib.rs` для библиотек. – Прим. перев.

```
let mut file = File::create("src/greet.rs").unwrap();
file.write_all(language.as_bytes()).unwrap();
}
```

Пишет содержимое языковой переменной. Метод `write_all` ожидает на входе получения байтов, так как файлы не всегда могут содержать текстовые данные, поэтому метод `.as_bytes` используется на строковом литерале, чтобы получить лежащие под ним байтовые данные.

Создает файл на диске (или воссоздает заново, если он уже существует).

Давайте сейчас попробуем выполнить скрипт сборки с помощью Cargo:

```
$ cargo run
Compiling build-script-test v0.1.0
error: failed to run custom build command for
`build-script-test v0.1.0`
```

Caused by:
process didn't exit successfully:
--- stderr
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: NotPresent'
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

Похоже, скрипт сборки поднял панику, потому что ему не была предоставлена новая ожидаемая переменная `GREET_LANG` среды. Давайте попробуем еще раз:

```
$ env GREET_LANG=en cargo run
Compiling build-script-test v0.1.0
Finished dev [unoptimized + debuginfo] target(s) in 0.25s
Running `build-script-test`
Hello, world!
```

Скрипт сборки успешно выполнен! Давайте посмотрим, привело ли это к ожидаемому результату. Теперь вы должны увидеть, что файл под названием `src/greet.rs` содержит все, что было передано компилятору в качестве переменной `GREET_LANG` среды:

```
$ ls src
main.rs greet.rs

$ cat src/greet.rs
en
```

В файл можно написать строковый литерал, но `en`, безусловно, не является допустимым файлом Rust. Скрипт сборки нужно немного подредактировать, чтобы записывать другой код Rust, в зависимости от видимого им значения переменной `GREET_LANG` среды.

Листинг 4.3 Скрипт сборки, пишущий код с помощью переменных среды

```
use std::fs::File;
use std::io::Write;

fn main() {
    let language = std::env::var("GREET_LANG").unwrap();

    let greeting = match language.as_ref() {
        "en" => "Hello!",
        "es" => "¡Hola!",
        "el" => "?ε?α σα?",
        "de" => "Hallo!",

        x => panic!("Unsupported language code {}", x),
    };

    let rust_code = format!("fn greet() {{
        println!("{}");
    }}", greeting);

    let mut file = File::create("src/greet.rs").unwrap();
    file.write_all(rust_code.as_bytes()).unwrap();
}
```

Метод `.as_ref` используется в связи с тем, что функция `std::env::var` возвращает значение типа `String`. Для того чтобы использовать выражение `match` со строковыми литералами (которые имеют тип `&strs`), необходимо конвертировать `String` в `&str`, используя метод `.as_reference`.

Необходимость пары `{}` вызвана тем, что макрокоманда `format!` использует фигурные скобки в качестве местозаполнителей для целей форматирования. `{}` используется для получения буквального символа фигурной скобки, необходимого для создания тела функции. Кавычки в макрокоманде `println!` экранируются аналогичным образом, чтобы преждевременно не завершать строковый литерал `rust_code`.

Теперь если повторно выполнять скрипт сборки, компилируя библиотеку несколько раз с разными языковыми опциями, то можно будет увидеть, что текст в `src/greet.rs` будет изменяться:

```
$ env GREET_LANG=en cargo run
hello!

$ cat src/greet.rs
fn greet() { println!("hello!"); }

$ env GREET_LANG=el cargo run
?ε?α σα?

$ cat src/greet.rs
fn greet() { println!("?ε?α σα?"); }
```

Таким образом, получилось написать исходный код Rust, но нужно обновить исполняемый файл, чтобы воспользоваться его преимуществами. В настоящее время исполняемый файл содержит только базовый код «Hello world!», предоставленный менеджером пакетов Cargo.

Листинг 4.4 Программа приветствия, использующая генерированный файл greet.rs

```
include!("greet.rs");    ← Вставляет (include!) текстовое содержимое файла src/greet.rs, выполняет его разбор как исходный код Rust и добавляет в файл src/main.rs. Префикс src/ в пути не требуется, потому что относительные пути макрокоманды include! строятся относительно файла исходного кода, в котором они используются.

fn main() {
    greet();    ←
}
```

Здесь можно вызвать функцию greet, потому что она была определена в src/greet.rs, а затем была использована макрокоманда include!, чтобы добавить текст из src/greet.rs в src/main.rs.

Выше была введена новая макрокоманда – `include!`. Она работает аналогично директиве C/C++ `#include`, то есть берет текстовое содержимое файла, выполняет его разбор как исходный код Rust и вставляет его в то место, где вызывается `include!`. На рис. 4.2 показано, как программа работает между скриптом сборки и файлом src/main.rs.

ПРИМЕЧАНИЕ. Макрокоманду `include!` не следует использовать для импорта файлов Rust в общем смысле. Система модулей языка Rust рассматривается в главе 5. Макрокоманду `include!`, как правило, следует использовать только с файлами исходного кода, генерируемыми динамически во время сборки.

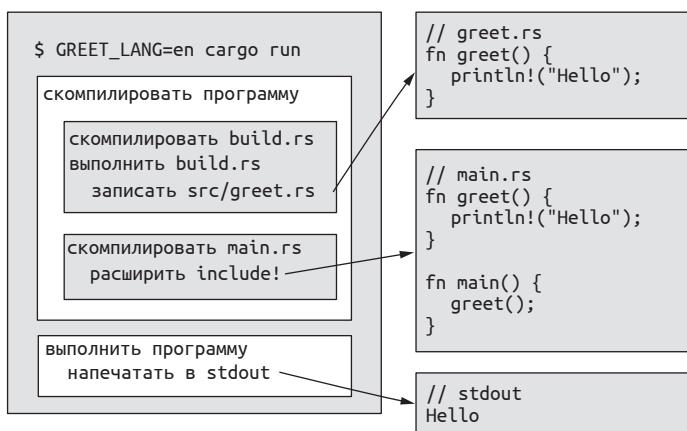


Рисунок 4.2 Компиляция и выполнение программы с помощью скрипта сборки

Немного разобравшись с тем, как скрипты сборки используются для генерации исходного кода Rust, теперь давайте вернемся к исходному коду NGINX. Вспомните, что требуется сгенерировать привязки языка Rust к С-интерфейсу веб-сервера NGINX. Для того чтобы сгенерировать эти привязки, можно написать кучу кода на языке Rust либо задействовать скрипт сборки, который сделает это автоматически.

Последний вариант предпочтительнее. Далее будет создан скрипт сборки, который использует библиотеку на языке Rust под названием (как и полагается) `bindgen`.

4.3.2 Инструмент `bindgen`

Инструмент `bindgen` – это библиотека на языке Rust, которая выполняет разбор исходного кода на языках C/C++ и автоматически генерирует привязки языка Rust. В своей простейшей форме инструмент `bindgen` генерирует совместимые с языком Rust определения типов и функций C/C++, загружаемых из одного заголовочного файла. Давайте начнем с добавления пакета `bindgen` в файл `Cargo.toml`:

```
[package]
name = "ngx_http_calculator_rs"
version = "0.1.0"
authors = ["You <you@you.com>"]
edition = "2018"

[dependencies]

[build-dependencies]
bindgen = "0.56.0"
```

Обратите внимание на секцию, куда включен пакет `bindgen`. Он находится не в секции `dependencies`, а в совершенно новой секции `build-dependencies`. Поскольку пакет `bindgen` будет использоваться только из скрипта сборки для генерации исходного кода Rust, его не нужно включать в окончательный двоичный файл как обычную зависимость; его нужно включить только в зависимости скрипта сборки.

Скрипт сборки нужен, чтобы сгенерировать привязки языка Rust к веб-серверу NGINX с помощью пакета `bindgen`. Инструмент `bindgen` выполняет разбор заголовочного файла C/C++ (следуя всем директивам `include`) с объявлениями типов, переменных и функций и выводит исходный код на языке Rust, совместимый с этими объявлениями.

Перед тем как применить инструмент `bindgen`, нужно создать этот заголовочный файл. Он должен включать (`#include`) все заголовки, к которым модулю Rust может потребоваться доступ. Давайте начнем с добавления заголовков, которые используются внутри нашего модуля C. Поместите следующее ниже содержимое в файл под названием `wrapper.h`:

```
#include <ngx_config.h>
#include <ngx_core.h>
#include <ngx_http.h>
```

Этот код представляет собой обычный заголовочный файл на языке C, но вместо того, чтобы использоваться для компиляции кода на языке C, он будет задействован для генерации кода на языке Rust. Поэтому теперь, когда заголовочный файл готов, давайте создадим файл `build.rs` и откроем его, чтобы посмотреть, как использовать инструмент `bindgen` для создания привязок.

Листинг 4.5 Скрипт сборки, создающий привязки языка Rust к веб-серверу NGINX

```

fn main() {
    let nginx_dir = "nginx-1.19.3";

    let bindings = bindgen::builder()
        .header("wrapper.h")           ←
        .clang_args(vec![
            format!("-I{}/src/core", nginx_dir),
            format!("-I{}/src/event", nginx_dir),
            format!("-I{}/src/event/modules", nginx_dir),
            format!("-I{}/src/os/unix", nginx_dir),
            format!("-I{}/objs", nginx_dir),
            format!("-I{}/src/http", nginx_dir),
            format!("-I{}/src/http/v2", nginx_dir),
            format!("-I{}/src/http/modules", nginx_dir),
        ])
        .generate()                   ←
        .unwгар();                   ←

    bindings
        .write_to_file("nginx.rs")   ←
        .unwгар();
}

```

wrapper.h – это только что созданный заголовочный файл. Инструмент `bindgen` принимает на входе только один заголовочный файл, и поскольку нужны типы из трех разных заголовочных файлов `NGINX`, нужно написать свой собственный заголовочный файл, который включает (`#include`) их все.

Этот список представляет собой аргументы командной оболочки, передаваемые компилятору C/C++ `clang`, когда тот используется для разбора заголовочного файла `wrapper.h`. Ему предоставляются каталоги, необходимые для расширения всех директив `#include` вплоть до дерева зависимостей заголовочных файлов внутри `NGINX`.

Конкретизирует местоположение вывода результатов работы инструмента `bindgen`. Привязки будут записываться в файл `nginx.rs`.

Давайте выполним скрипт сборки, перекомпилировав библиотеку. На этот раз весь процесс может занять чуть больше времени, так как сейчас компилятор выполняет большую работу по инспекции заголовочных файлов `NGINX` при запуске. По завершении этапа сборки вы должны увидеть, что в корень пакетного каталога `nginx.rs` помещен новый файл. Откройте этот файл и ознакомьтесь с ним. Просмотрев несколько фрагментов сгенерированного кода Rust для работы с битовыми полями, вы, возможно, заметите, что многие представленные в этом файле типы и функции имеют мало общего с самим веб-сервером `NGINX`. На минуточку, здесь описана вся стандартная библиотека C! Эта поверхность API, вероятно, намного больше, чем понадобится для интеграции, и ее включение только увеличит время компиляции. Указанный файл содержит более 51 000 строк, и любые усилия по уменьшению этого размера были бы напрасными. Указанный файл можно ограничить, воспользовавшись функциональностью `whitelist` пакета `bindgen`.

ПРИМЕЧАНИЕ. Если вы получите сообщение об ошибке, связанной с отсутствием файлов `libclang.so`, то из менеджера пакетов операционной системы необходимо установить библиотеку `libclang`. В инструменте `bindgen` для разбора передаваемых ему файлов на языках C и C++ используется библиотека `libclang`.

Внимательные читатели, возможно, заметили, что типы и функции в API модуля веб-сервера NGINX начинаются с префикса `ngx_`. Для того чтобы включить только те типы, функции и глобальные переменные, которые начинаются с этого префикса, игнорируя все остальные, можно воспользоваться регулярным выражением. Давайте вернемся к файлу `build.rs` и добавим эти правила.

Листинг 4.6 Скрипт сборки `bindgen` принимает только элементы с префиксом `ngx_`

```
fn main() {
    let nginx_dir = "nginx-1.19.3";

    let bindings = bindgen::builder()
        .header("wrapper.h")
        .whitelist_type("ngx_.*")
        .whitelist_function("ngx_.*")
        .whitelist_var("ngx_.*") | Эти методы whitelist_ принимают
        .clang_args(vec![
            format!("-I{}/src/core", nginx_dir),
            format!("-I{}/src/event", nginx_dir),
            format!("-I{}/src/event/modules", nginx_dir),
            format!("-I{}/src/os/unix", nginx_dir),
            format!("-I{}/objs", nginx_dir),
            format!("-I{}/src/http", nginx_dir),
            format!("-I{}/src/http/v2", nginx_dir),
            format!("-I{}/src/http/modules", nginx_dir),
        ])
        .generate()
        .unwarp();

    bindings
        .write_to_file("nginx.rs")
        .unwarp();
}
```

После повторной сборки теперь получится файл `nginx.rs` с 30 000 строк кода. Это не идеально, но, безусловно, выглядит лучше по сравнению с предыдущим шагом. Достаточно мотивированный разработчик мог бы пройти весь путь и в явной форме разрешить каждый отдельный тип, необходимый для того, чтобы его интеграция с использованием интерфейса с внешними функциями (FFI) заработала, но на данном этапе в этом нет необходимости.

В скрипте сборки нужно еще кое-что изменить: до сих пор файл `nginx.rs` размещался в корне пакетного каталога. Однако на самом

деле ему там не место. При генерации файлов в рамках скрипта сборки, которые должны будут вставлены в последующих шагах компиляции, они должны помещаться в выходной каталог. Cargo управляет выходным каталогом, который уникalen для каждого запуска компилятора. Именно туда следует помещать все сгенерированные файлы, поскольку, вероятно, нежелательно сохранять изменения в 30 000 строк сгенерированного кода в системе контроля версий!

Местоположение выходного каталога можно узнать, только проверив переменные среды, задаваемые менеджером пакетов Cargo. Для скриптов сборки Cargo задает ряд переменных среды при выполнении скрипта, и эти же переменные среды передаются в главный пакет во время компиляции. Давайте посмотрим, как сослаться на эту переменную среды, чтобы поместить файл `nginx.rs` в выходной каталог. Замените последние три строки в нижней части функции `main` в файле `build.rs` следующими ниже строками:

```
let out_dir = std::env::var("OUT_DIR").unwrap(); ← Cargo автоматически задает переменную OUT_VAR для скриптов сборки.

bindings
    .write_to_file(format!("{}/nginx.rs", out_dir))
    .expect("unable to write bindings");
```

Теперь, когда сгенерированный код находится в нужном месте, необходимо его добавить в библиотеку Rust, используя макрокоманду `include!`, которая обсуждалась ранее в этой главе. Поскольку исходный файл находится в `$OUT_DIR/nginx.rs`, нужен способ извлечения переменных во время компиляции. Можно было бы использовать функцию `std::env::var`, как это делалось в скрипте сборки, но она применяется для проверки значений во время исполнения. Здесь же нужно проверить значение этой переменной во время компиляции. Вместо этого можно применить макрокоманду `env!`. Указанная макрокоманда расширяется в строковый литерал, содержащий значение переменной среды на момент компиляции программы. Если переменная не указана, то возникнет ошибка компилятора. В нашем примере переменная `OUT_DIR` среды проверяется с помощью

```
env!("OUT_DIR")
```

Таким образом, имеется выходной каталог, и известно, что внутри этого каталога нужен `nginx.rs`, но как совместить эти две вещи? Во время исполнения достаточно было бы применить макрокоманду `format!`, чтобы скрепить их вместе с помощью разделителя путей посередине, но как сделать то же самое во время компиляции? Ответом является макрокоманда `concat!`. Указанная макрокоманда выполняет простые операции конкатенации строковых литералов, известных во время компиляции. Поскольку требуется сгенерировать путь, который выглядит как `$OUT_DIR/nginx.rs`, макрокоманду `concat!` можно применить следующим образом:

```
concat!(env!("OUT_DIR"), "/nginx.rs")
```

Этот метод немного отличается от того, как этот же путь строился в скрипте сборки, но следует помнить, что время исполнения скрипта сборки по сути совпадает со временем компиляции кода приложения. К сожалению, для выполнения той же задачи требуется немного другая семантика. Теперь, когда имеются все детали, давайте соберем их вместе.

Откройте файл `src/lib.rs` и добавьте в начало файла следующее:

```
include!(concat!(env!("OUT_DIR"), "/nginx.rs"));
```

Многовато макрокоманд, если честно! Давайте рассмотрим их по очереди:

- `include!` представляет собой операцию вставки исходного кода, аналогично директиве `#include` в C/C++;
- `concat!` выполняет конкатенацию строковых литералов во время компиляции;
- `env!` проверяет значение переменной `OUT_DIR` среды во время компиляции.

На рис. 4.3 показан наглядный вид каждой из этих частей.

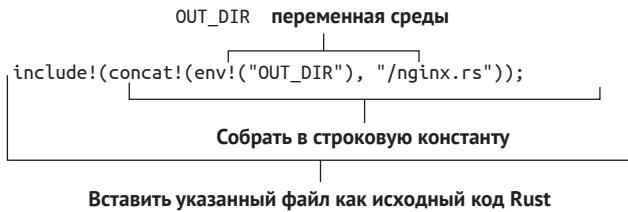


Рисунок 4.3 Диаграмма нового синтаксиса

Теперь, когда есть понимание того, как вставлять сгенерированный код NGINX, наконец, можно вернуться к той функции-обработчику HTTP, которая была объявлена уже давно. Если вставить ее в `src/lib.rs` с помощью макрокоманды `include!`, которая была только что написана, и дополнительное сообщение «Hello from Rust!», то она должна выглядеть следующим образом.

Листинг 4.7 Полностью сформированная минимальная функция-обработчик веб-сервера NGINX на языке Rust

```
include!(concat!(env!("OUT_DIR"), "/nginx.rs"));

#[no_mangle]
pub unsafe extern "C" fn ngx_http_calculator_handler(
    r: *mut ngx_http_request_t
) -> ngx_int_t {
    eprintln!("Hello from Rust!");
    0
}
```

Если сейчас попытаться скомпилировать этот исходный код, то он будет работать! Будет выдано большое количество предупреждений из-за имен в стиле C, которые инструмент `bindgen` генерирует и которые не соответствуют руководящим стилем принципам языка Rust. Эти предупреждения можно заглушить с помощью нескольких директив компилятора, но пока давайте продолжим.

Вспомните из главы 3, что при связывании к исходному коду на Rust из исходного кода на С нужно сообщить менеджеру пакетов Cargo о необходимости генерировать динамическую библиотеку, совместимую с С, вместо библиотеки, совместимой с Rust, то есть вместо обычного генерируемого им формата. Откройте файл `cargo.toml` и добавьте следующие ниже строки:

```
[lib]
crate-type = ["cdylib"]
```

Собрав пакет, теперь в каталоге `build` можно будет найти динамическую библиотеку:

```
$ cargo build
$ ls target/debug/*
target/debug/libnginx_http_calculator_rs.so
```

Ввиду того что эта динамическая библиотека содержит функцию-обработчик HTTP, к ней нужно привязаться из С-модуля NGINX, добавив дополнительную конфигурационную переменную в файл `module/config`:

```
ngx_module_type=HTTP
ngx_module_name=nginx_http_calculator
ngx_module_srcs="$ngx_addon_dir/nginx_http_calculator.c"
ngx_module_libs="/path/to/your/libnginx_http_calculator_rs.so" ↴
.
auto/module
ngx_addon_name=$ngx_module_name
```

Только что добавленная строка. Абсолютный путь нужен для того, чтобы никакие различия в урегулировании относительного пути не вызывали проблем при попытке загрузить модуль во время выполнения веб-сервера NGINX.

Раз была обновлена конфигурация модуля, его необходимо перекомпилировать. К сожалению, после обновления конфигурационных файлов модуля процесс сборки веб-сервера NGINX нуждается в повторном выполнении скрипта `configure` и пересборке двоичного файла. Это нужно будет сделать в последний раз:

```
$ cd nginx-1.19.3
$ ./configure --add-dynamic-module=../module
$ make -j16 build modules
```

Теперь, после всех этих шагов, наконец, все готово к тому, чтобы запустить веб-сервер NGINX, и должно появиться сообщение «Hello from Rust!».

Прежде всего давайте запустим веб-сервер NGINX, используя ту же команду, что и ранее. Он должен распечатать несколько со-

общений уровня «уведомлений», а затем ничего не делать, ожидая получения HTTP-запросов. Используйте отдельный терминал для отправки HTTP-запроса в конечную точку `/calculate`, для которой был активирован модуль в файле `nginx.conf`. Сам HTTP-запрос должен завершиться ошибкой, но более интересным является то, что появляется в записях регистрации событий (логах) веб-сервера NGINX:

```
$ ./nginx-1.19.3 objs/nginx -c nginx.conf -p ngx-run
....
Hello from Rust!

# Concurrently, in a separate window after NGINX is started
$ curl -X POST -d '3 4 * 2 -' http://localhost:8080/calculate
<html>
<head><title>400 Bad Request</title></head>
<body>
<center><h1>400 Bad Request</h1></center>
<hr><center>nginx/1.19.3</center>
</body>
</html>
```

Получилось! HTTP-запрос был успешно перенаправлен из исходного кода NGINX, написанного на языке C, в функцию-обработчик HTTP на языке Rust. Теперь, когда есть некоторый уровень обмена данными между двумя системами, нужно перейти к реализации деловой логики обработчика HTTP.

4.4

Чтение запроса веб-сервера NGINX

Процедура получения данных из тела POST-запроса веб-сервера NGINX не так уж и сложна. Она очень похожа на метод, который использовался в главе 3 для чтения данных из зарезервированного в стеке буфера STDIN. Однако вместо обращения к буферу как простому аргументу `*const u8` функции веб-сервер NGINX предоставляет параметр `ngx_http_request_t`, который содержит целый ряд полей. Это значение нужно преобразовать во что-то понятное исходному коду на языке Rust.

В HTTP-стек веб-сервера NGINX встроены самые разные модули обработки запросов, и не для всех из них требуется чтение содержимого тела HTTP-запроса. Поэтому передаваемая функциям-обработчикам HTTP структура запроса на самом деле еще не содержит загруженного тела запроса. Для того чтобы получить эти данные, нужно вызывать метод из библиотеки HTTP, который выполняет синтаксико-структурный разбор тела. Потребуется функция `request_body`. Она принимает указатель на запрос и указатель на функцию, которая будет вызвана, когда тело запроса будет прочитано в память. Давайте посмотрим, как можно было бы ее использовать для загрузки тела запроса.

Листинг 4.8 Обработчик запроса, который может читать тело запроса

```

include!(concat!(env!("OUT_DIR"), "/nginx.rs"));

#[no_mangle]
pub unsafe extern "C" fn ngx_http_calculator_handler(
    r: *mut ngx_http_request_t,
) -> ngx_int_t {
    let rc = ngx_http_read_client_request_body(
        r, Some(read_body_handler)); ←
    if rc != 0 {
        return rc;
    }
    0
}

unsafe extern "C" fn read_body_handler(
    r: *mut ngx_http_request_t) {
    if r.is_null() {
        eprintln!("got null request in body handler");
        return;
    } ←
    let request = &*r;

    let body = match request_body_as_str(request) {
        Ok(body) => body,
        Err(e) => {
            eprintln!("failed to parse body: {}", e);
            return;
        }
    }; ←
    eprintln!("Read request body: {:?}", body);
}

unsafe fn request_body_as_str<'a>(
    request: &'a ngx_http_request_t,
) -> Result<&'a str, &'static str> {
    if request.request_body.is_null()
        || (*request.request_body).bufs.is_null()
        || (*(*request.request_body).bufs).buf.is_null()
    {
        return Err("Request body buffers
                  were not initialized as expected");
    }
}

```

ngx_http_calculator_handler – это точка входа, которую NGINX вызывает при получении запроса.

ngx_http_read_client_request_body считывает тело запроса из сети и добавляет его в буфер структуры запроса. Поскольку чтение из сети может занимать некоторое время, необходимо предоставить функцию обратного вызова, которую NGINX будет вызывать по завершении.

read_body_handler – это функция обратного вызова, которую NGINX вызывает, когда читает тело запроса в память из сети.

Распечатывает тело запроса, после того как его разбор был выполнен из структуры запроса NGINX.

request_body_as_str читает тело запроса из структуры запроса NGINX и пытается его интерпретировать как строковый срез Rust. Она не резервирует никакой дополнительной памяти, а просто переинтерпретирует существующие байты.

```

let buf = (*(*request.request_body).bufs).buf;

let start = (*buf).pos;
let len = (*buf).last.offset_from(start) as usize;

let body_bytes = std::slice::from_raw_parts(start, len);

let body_str = std::str::from_utf8(body_bytes)
    .map_err(|_| "Body contains invalid UTF-8")?;

Ok(body_str)
}

```

В этом примере исходного кода можно выделить несколько моментов, но давайте начнем с трех определенных в нем функций. Сначала обратите внимание на различные уровни аннотаций, которые появляются в этих функциях. Давайте посмотрим на сигнатуры функций без каких-либо параметров или исходного кода тел функций. В дополнение к стандартному ключевому слову `fn` все три функции содержат дополнительные аннотации, но ни одна из них не содержит совершенно одинаковых аннотаций:

```

#[no_mangle]
pub unsafe extern "C" fn ngx_http_calculator_handler

unsafe extern "C" fn read_body_handler

unsafe fn request_body_as_str

```

На рис. 4.4 все эти элементы показаны наглядно.

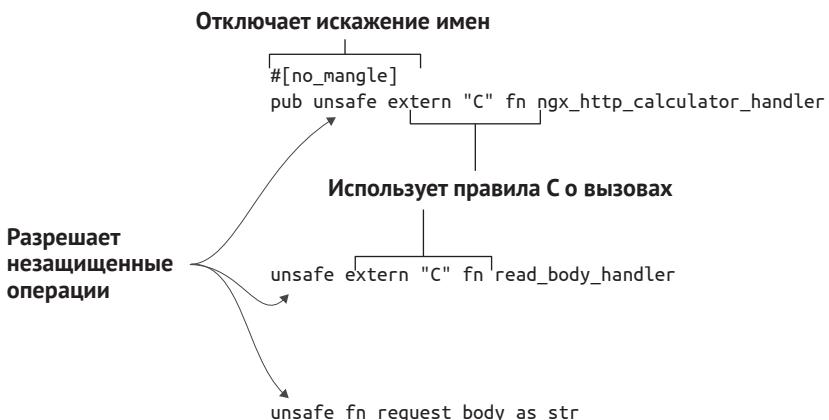


Рисунок 4.4 Анатомия разных элементов сигнатур функций

Первая функция называется `ngx_http_calculator_handler`. Этую функцию необходимо вызывать по имени из исходного кода на C, и она должна выполнять незащищенные операции внутри нее. Ей нужны `#[no_mangle]` и `pub`, чтобы передавать свое имя через границу межъ-

языкового FFI-интерфейса исходному коду на C, и ей нужен `extern "C"`, чтобы ее можно было защищенно вызывать из исходного кода на C. В дополнение к этому, ввиду того что искажение имен отключено, для функции нужно использовать пространство имен в стиле C, отсюда наличие префикса `ngx_http_calculator` во избежание конфликтов с другими функциями C.

Далее идет функция обратного вызова `read_body_handler`. Эта функция должна быть доступна для вызова из исходного кода на C, но исходному коду на C не нужно знать ее имя, только ее местоположение в памяти. Следовательно, предоставлено `extern "C"`, чтобы использовались правила C о вызовах и чтобы функция могла применяться за пределами межъязыкового FFI-интерфейса. Поскольку имя функции всегда будет использоваться только в исходном коде на Rust, отключать искажение имени или публиковать эту функцию не нужно. Внутри этой функции выполняются незащищенные операции, поэтому ключевое слово `unsafe` также добавляется в сигнатуру.

Наконец, последней идет функция `request_body_as_str`. Эта функция вызывается только из обычного исходного кода на Rust; она никогда не будет вызываться из C. Это очевидно из-за отсутствия аннотации `extern "C"`. Поэтому будут использоваться правила Rust о вызовах, и вызывать эту функцию из исходного кода на C небезопасно.

Теперь, когда есть понимание сигнатур этих трех функций, давайте немного углубимся в их реализацию. Начнем с функции `ngx_http_calculator_handler`:

```
#[no_mangle]
pub unsafe extern "C" fn ngx_http_calculator_handler(
    r: *mut ngx_http_request_t,
) -> ngx_int_t {
    let rc = ngx_http_read_client_request_body(
        r, Some(read_body_handler));
    if rc != 0 {
        return rc;
    }
    0
}
```

Эта функция выполняет только три действия: она вызывает функцию `ngx_http_read_client_request_body`, чтобы настроить цепочку событий для чтения в теле HTTP-запроса POST, проверяет возвращаемый код и возвращает ноль, сообщая веб-серверу NGINX о том, что ошибок нет. Поскольку эта функция вызывается только самим веб-сервером NGINX, она должна соответствовать довольно строгому определению того, что делает функция-обработчик HTTP в рамках NGINX. Она должна принимать в качестве параметра одну структуру запроса и возвращать код состояния типа `int`. В NGINX многие функции возвращают коды состояния типа `int`, где ноль означает статус успешного выполнения.

Давайте немного подробнее рассмотрим функцию `ngx_http_read_client_request_body`. Если открыть автоматически созданный файл `$OUT_DIR/nginx.rs`, то можно будет увидеть определение этой функции на Rust, а если заглянуть в файл `nginx-1.19.3/src/http/ngx_http_request_body.c`, то можно будет сравнить ее с сигнатурой С:

```
pub fn ngx_http_read_client_request_body(          ← Сигнатура функции
    r: *mut ngx_http_request_t,                   ← Rust, автоматически
    post_handler: ngx_http_client_body_handler_pt, ← сгенерированная ин-
) -> ngx_int_t;                                ← струментом bindgen.

ngx_int_t ngx_http_read_client_request_body(          ← Сигнатура
    ngx_http_request_t *r,                         ← функции С.
    ngx_http_client_body_handler_pt post_handler,
)
```

Сигнатуры двух функций практически идентичны. Помимо этого, также включены определения типа `body_handler`, который требуется обеим функциям:

```
pub type ngx_http_client_body_handler_pt =          ← Тип Rust, автоматически
    Option<unsafe extern "C" fn(r: *mut ngx_http_request_t)>; ← сгенерированный
                                                               ← инструментом bindgen.

→     typedef void (*ngx_http_client_body_handler_pt)      ← Тип Rust ngx_http_client_
            (ngx_http_request_t *r);                         ← body_handler_pt оберты-
                                                               ← вает дескриптор функции
                                                               ← в тип Option, чтобы иметь
                                                               ← возможность чисто обра-
                                                               ← батывать случай указате-
                                                               ← ля null на функцию.
```

Тип С.

Хорошо видно, что, обернув параметр функции в тип `Option`, инструмент `bindgen` сделал возможность данного параметра использовать с `null` немного более очевидной. Поэтому при передаче функции `read_body_handler` в качестве обратного вызова в функцию `ngx_http_read_client_request_body` ее нужно обернуть в конструкцию `Some`. Таким способом инструмент `bindgen` генерирует в исходном коде Rust типы указателей на функции, поступающие из исходного кода на С. Кроме того, посмотрев на определение типа Rust, можно также заметить, что сигнатура функции в типе `Option` совпадает с сигнатурой определенной ранее функции обратного вызова. Ниже они представлены обе:

```
pub type ngx_http_client_body_handler_pt =
    Option<unsafe extern "C" fn(r: *mut ngx_http_request_t)>

unsafe extern "C" fn read_body_handler(
    r: *mut ngx_http_request_t)
```

Тип указывает на то, что необходимо предоставить функцию обратного вызова, которая принимает указатель на запрос и ничего не возвращает. Этот обратный вызов представлен в виде функции

`read_body_handler`. Теперь, когда есть понимание точки входа в обработчик, давайте посмотрим на реализацию указанного обратного вызова:

```
unsafe extern "C" fn read_body_handler(r: *mut ngx_http_request_t)
{
    if r.is_null() {
        eprintln!("got null request in body handler");
        return;
    }

    let request = &*r;

    let body = match request_body_as_str(request) {
        Ok(body) => body,
        Err(e) => {
            eprintln!("failed to parse body: {}", e);
            return;
        }
    };

    eprintln!("Read request body: {:?}", body);
}
```

Большая часть исходного кода в этой функции вполне предсказуема. Новинкой является только то, что непосредственно перед вызовом `request_body_at_str` стоит вот эта строка:

```
let request = &*r;
```

Вы уже знаете, что `&` используется для получения ссылки на объект, но что означает `*?` В языке Rust этот символ называется оператором *взятия значения по ссылке*. Как следует из названия, взятие значения по ссылке означает использование ссылки для получения объекта, на который ссылка указывает. Он очень похож на оператор снятия косвенности (дереференции) в таких языках, как C, C++ и Go.

Совместное использование этих двух операторов на обыкновенном указателе называется *перезамствованием*. По сути, перезамстование – это конвертация обычного указателя в ссылку Rust. Разница между ними, возможно, будет немного неясной, но это потому, что во время исполнения они абсолютно одинаковы!

Ссылка Rust – это просто указатель, о котором у компилятора есть чуть больше информации. Если подумать об указателе в языке C или C++, то у компилятора нет абсолютно никакой информации о том, откуда берется память под указатель, как долго он будет действителен или инициализировано ли лежащее под ним значение. Ссылка Rust позволяет компилятору знать всю эту информацию. Поскольку все ссылки связаны со временем жизни, известна продолжительность времени, в течение которого ссылка будет действительна. Предполагается, что все ссылки выровнены, не равны `null` и указывают на инициализированные значения.

Потребность в конвертации указателя в ссылку возникает по некоторым причинам:

- большая часть исходного кода на Rust написана для работы со ссылками, а не с указателями, поэтому использование ссылок вместо указателей значительно упрощает реиспользование исходного кода;
- можно выполнить однократную проверку на `null` перед конвертацией и больше никогда об этом не беспокоиться, потому что ссылки Rust *всегда должны быть не-null*;
- не нужно использовать ключевое слово `unsafe` для доступа к данным, находящимся за ссылкой. Хотя все функции в этом примере являются `unsafe`, как вы увидите в главе 5, большая часть исходного кода не обязательно должна быть такой;
- доступ к полям через указатель на структуру затруднен, потому что в языке Rust нет оператора доступа к полю через указатель, как в C или C++;
- наличие ссылки позволяет увязывать соотносящиеся времена жизни вместе, как вы вскоре увидите в объявлении функции `request_body_as_str`.

При этом при конвертации из указателя в ссылку нужно придерживаться нескольких рекомендаций:

- поскольку ссылки Rust трактуются всем исходным кодом как не равные `null`, в этом необходимо удостоверяться перед выполнением конвертации. Как вы видите, эта проверка на `null` – первое, что делается в `read_body_handler`;
- хранящийся в указателе объект должен быть допустимым экземпляром типа. Например, многие функции резервирования памяти в C возвращают неинициализированную память; заимствование этой памяти как `&mut T`, а затем инициализация памяти, используя ссылку, является незащищенной операцией. Память должна быть инициализирована с использованием операций на указателях;
- если объект является ссылкой, то он должен соответствовать принятым в языке Rust правилам заимствования. Поскольку создается немутируемая ссылка, компилятор Rust будет исходить из того, что никакой другой код не изменит содержимое указателя. Если фоновый поток исполнения пишет данные в этот указатель, а Rust поддерживает немутируемую ссылку на него, то создается неопределенное поведение.

По завершении проверок на `null` важно учесть время жизни. Раз берется указатель, у которого нет информации о времени жизни, и превращается в ссылку, у которой такая информация есть, то откуда берется это время жизни? Короткий ответ таков: оно всегда было там; компилятор просто не знал об этом!

Поскольку известно, что исполняемый файл NGINX не изменяет этот запрос в фоновом режиме, проводится проверка на `null` и можно

обоснованно полагать, что память инициализирована. Следовательно, можно безопасно превратить этот указатель в ссылку.

Давайте рассмотрим еще одну функцию в обработчике – `request_body_as_str`. Эта функция принимает ссылку на структуру запроса веб-сервера NGINX и возвращает строковый срез, содержащий тело HTTP-запроса, либо ошибку, если его не удалось прочитать. В этой функции появилось несколько новых элементов, и они все будут проинспектированы:

```
unsafe fn request_body_as_str<'a>(
    request: &'a ngx_http_request_t,
) -> Result<&'a str, &'static str> {
    if request.request_body.is_null()
        || (*request.request_body).bufs.is_null()
        || (**request.request_body).bufs.buf.is_null()
    {
        return Err("Request body buffers
                    were not initialized as expected");
    }

    let buf = (**request.request_body).bufs.buf;

    let start = (*buf).pos;
    let len = (*buf).last.offset_from(start) as usize;

    let body_bytes = std::slice::from_raw_parts(start, len);

    let body_str = std::str::from_utf8(body_bytes)
        .map_err(|_| "Body contains invalid UTF-8")?;

    Ok(body_str)
}
```

Первое, что бросается в глаза как новинка, находится очень близко к началу сигнатуры функции. Там расположена новая разновидность аргумента функции – обобщенный аргумент времени жизни! Какова его цель?

```
unsafe fn request_body_as_str<'a>(
    request: &'a ngx_http_request_t,
) -> Result<&'a str, &'static str>
```

Разобраться в этом будет трудновато, поэтому давайте ненадолго отойдем от примера с веб-сервером NGINX и его сложностей, чтобы рассмотреть гораздо более простую программу.

4.4.1 Аннотации времен жизни

Для того чтобы эффективно делиться памятью в программах на языке Rust, иногда нужно помогать компилятору понимать, как ссылки взаимосвязаны между собой. Компилятор нередко достаточно умен, чтобы неявно выявлять эти взаимосвязи, но иногда ему нужна помощь. Эта помощь может быть предоставлена в виде *аннотаций времен жизни*.

Листинг 4.9 Простая программа на языке Rust

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    let value = &numbers[0];

    println!("value: {}", value);
}
```

Приведенная выше программа создает экземпляр типа `Vec`, содержащий пять чисел, берет первое число и распечатывает его. Представьте себе, что нужно перенести стержневую функциональность этой программы, ту часть, которая получает число из списка, в обособленную функцию. Это делается очень просто.

Листинг 4.10 Программа на языке Rust с использованием вспомогательной функции

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    let value = get_value(&numbers);

    println!("value: {}", value);
}

fn get_value(numbers: &Vec<i32>) -> &i32 {
    &numbers[0]
}
```

Приведенный выше исходный код компилируется, но почему он компилируется? Как компилятор узнает из сигнатуры функции `get_value` о допустимости указанных здесь времен жизни? Вспомните, что происходит, когда делается попытка вернуть ссылку на локальную переменную? Разумеется, функция не компилируется.

Листинг 4.11 Функция, пытающаяся вернуть ссылку на локальную переменную

```
fn get_value() -> &i32 {
    let x = 4;
    &x
}
```

Причина, по которой исходный код в листинге 4.10 компилируется, а исходный код из листинга 4.11 – нет, заключается в способности компилятора заключить, что выходное время жизни в листинге 4.10 совпадает с входным временем жизни. На рис. 4.5 показан график времен жизни для этой программы.

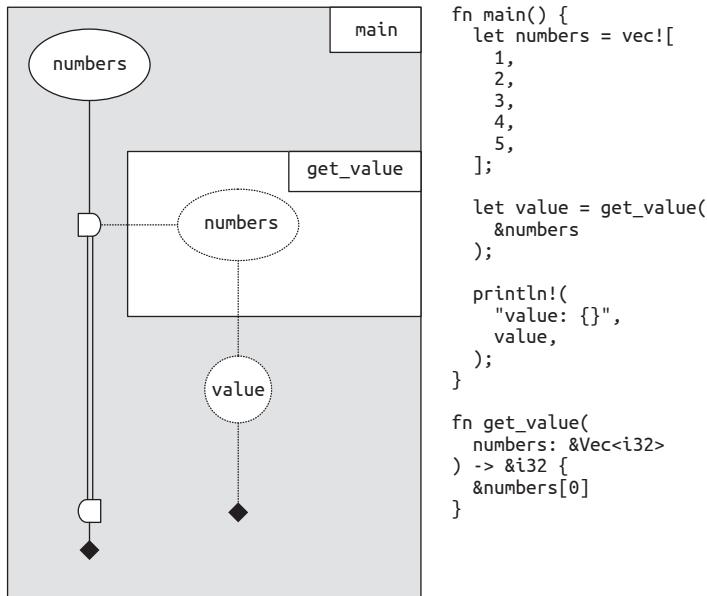


Рисунок 4.5 График времен жизни для листинга 4.10

Приведенный выше график времен жизни показывает, что ссылка, исходящая из функции `get_value`, напрямую произошла от ссылки, которая в нее входит. Обе ссылки имеют одинаковое время жизни. Если попытаться использовать ссылку, возвращенную из `get_value`, после лишения ее способности ссылаться¹, то можно будет увидеть последствия:

```

fn main() {
    let mut numbers = vec![1, 2, 3, 4, 5];

    let value = get_value(&numbers);

    numbers.push(6);

    println!("value: {}", value);
}

fn get_value(numbers: &Vec<i32>) -> &i32 {
    &numbers[0]
}

```

Компилятор Rust не примет эту программу. Будет выдано следующее ниже сообщение об ошибке:

```

$ cargo run
error[E0502]: cannot borrow `numbers` as mutable
  because it is also borrowed as immutable
--> src/main.rs:6:3

```

¹ Так же принято использовать термин «инвалидация ссылки» как лишение ее способности ссылаться на действительный объект в памяти. – *Прим. перев.*

```

4 |     let value = get_value(&numbers);
|           ----- immutable borrow occurs here
5 |
6 |     numbers.push(6);
|     ^^^^^^^^^^^^^^^^ mutable borrow occurs here
7 |
8 |     println!("value: {}", value);
|           ----- immutable borrow later used here

```

Компилятор жалуется, что подвергать `numbers` мутации нельзя в связи с тем, что переменная `value` содержит немутуируемое заимствование `numbers`. Поскольку компилятор знает, что `value` ссылается на память внутри `numbers`, он не позволит подвергать `numbers` мутации. Опытные разработчики на C и C++, возможно, сталкивались с лишением указателей способности указывать из-за перераспределения памяти под буфер, что невозможно в защищенном языке Rust ввиду означенного правила, предотвращающего мутацию уже заимствованной памяти.

В данном случае компилятор Rust достаточно умен и выяснил, что входные и выходные времена жизни ссылок совпадают, но в функцию можно внести очень небольшое изменение, которое лишит компилятор возможности эффективно судить об этом.

Листинг 4.12 Возвращение ссылки на аргумент

```

fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    let value = get_value(&numbers, "Getting the number");

    println!("value: {}", value);
}

fn get_value(numbers: &Vec<i32>, s: &str) -> &i32 {
    println!("{}", s);
    &numbers[0]
}

```

Если попытаться исполнить исходный код, приведенный в листинге 4.12, то будет получена новая ошибка компилятора:

```

$ cargo run
error[E0106]: missing lifetime specifier
--> src/main.rs:9:46
|
9 | fn get_value(numbers: &Vec<i32>, s: &str) -> &i32 {
|           ----- ----- ^ expected named
|                           lifetime parameter
|
= help: this function's return type contains a borrowed value,
      but the signature does not say whether it is borrowed
      from `numbers` or `s`

```

```
help: consider introducing a named lifetime parameter
|
9 | fn get_value<'a>(numbers: &'a Vec<i32>, s: &'a str) -> &'a i32
{ |
    ^^^^           ^^^^^^^^^^           ^^^^^^           ^^^
```

Ошибка компилятора дает отличную подсказку о том, в чем проблема и как ее можно исправить. Новая функция `get_value` содержит две ссылки в качестве входных параметров. Однако выходной параметр может иметь только одно время жизни, поэтому компилятору необходимо знать, какое время жизни назначить выходному параметру. У кого заимствуется число, которое возвращает `get_value`: у `numbers` или у `s`? В данном случае заимствуется у `numbers`, но компилятор должен знать до того, как он будет определять работоспособность или неработоспособность программы. Компилятору об этом сообщают, используя *аннотации времен жизни*. Вы немного ознакомились с ними в ошибке компилятора, но нужно внести одно небольшое изменение.

Листинг 4.13 Возвращение ссылки на аргумент

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    let value = get_value(&numbers, "Getting the number");

    println!("value: {}", value);
}

fn get_value<'a>(numbers: &'a Vec<i32>, s: &str) -> &'a i32 {
    println!("{}", s);
    &numbers[0]
}
```

Единственная измененная строка: добавлена явная аннотация времени жизни ('a).

Перед списком параметров по значению появляется новый синтаксис (`<'a>`). В этих угловых скобках Rust помещает аргументы с обобщенным типом, аналогично тому, как аргументы с обобщенным типом форматируются в языках Java и Typescript. Но что такое '`a` в угловых скобках? Это аннотация времени жизни. Вспомните, что когда времена жизни рассматривались в первый раз, было показано, что времена жизни `'static` использовалось для ссылок, которые были действительны в течение всего времени выполнения программы и никогда не высвобождались. Теперь, как вы видите, можно создавать другие именованные времена жизни, чтобы ссыльаться на отдельные не-'`static` времена жизни. На рис. 4.6 более подробно показан принцип работы этого синтаксиса в данном примере.

Давайте также посмотрим на график времен жизни для этой новой программы, чтобы увидеть, как аннотации времен жизни помогают компилятору принимать решения о том, как взаимодействуют различные заимствования. График показан на рис. 4.7.

```
fn get_value<'a>(numbers: &'a Vec<i32>, s: &str) -> &'a i32
```

Существует некое время жизни, которое будем называть '*a*'.

Время жизни '*a*' описывает время жизни ссылки на *numbers*, которая была передана в эту функцию.

'*a*' также описывает время жизни ссылки на *i32*, возвращаемой из *get_value*.

Совместив эти два факта, можно заключить, что *get_value* возвращает ссылку на элемент внутри *numbers* типа *Vec*.

Рисунок 4.6 Более пристальный взгляд на синтаксис аннотации времени жизни

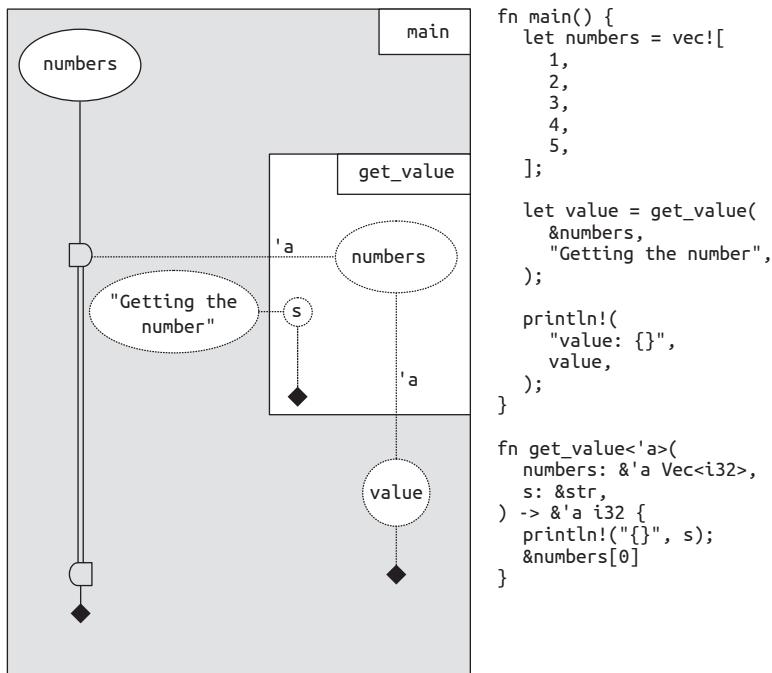


Рисунок 4.7 График времен жизни для листинга 4.13

Эти аннотации времен жизни всегда можно предоставлять в любое время, когда у функции есть ссылочный аргумент, но в большинстве случаев они не нужны, потому что компилятор может безопасно делать вывод о них сам. Также верно и то, что сигнатуру для функции *get_value* можно написать следующим образом:

```
fn get_value<'a, 'b>(numbers: &'a Vec<i32>, s: &'b str) -> &'a i32
```

Эта сигнатура явно указывает время жизни ссылки *s* как '*b*', но компилятору не нужно знать эту информацию, так как '*b*' не взаимодей-

ствует ни с одним значением, кроме как с `s`. Компилятор автоматически вставляет эти дополнительные ненужные правила времен жизни в исходный код, когда они не предоставляются разработчиком, но в техническом смысле будет правильно указывать все времена жизни в параметрах функции явным образом (хоть и нежелательно в стилистическом смысле).

В тех случаях, когда компилятор не способен сделать вывод о временах жизни только из сигнатуры типа, например в функциях с несколькими ссылочными параметрами и возвращаемым ссылочным значением, ему необходима информация от разработчика. Он не способен вывести ее логически из функции, так как выходное время жизни фактически является частью контракта функции в отношении публичного API. Если бы компилятор должен был определять выходное время жизни, просматривая код в теле функции, то появлялась бы возможность вносить радикальные изменения в API без изменения сигнатур функций. Гораздо менее опасно просить разработчика самому писать аннотации, с тем чтобы функции со сложными временами жизни в их публичных API не подвергались радикальным изменениям.

Немного разобравшись с предназначением и использованием аннотаций, теперь давайте вернемся к исходному коду плагина для веб-сервера NGINX.

4.4.2 Аннотации времен жизни в плагине для веб-сервера NGINX

В частности, будет рассмотрена функция `request_body_as_str`. Напомним, что рассматривается следующая ниже сигнатура функции:

```
unsafe fn request_body_as_str<'a>(
    request: &'a ngx_http_request_t,
) -> Result<&'a str, &'static str>
```

Теперь, когда есть понимание принципа работы аннотаций времен жизни, вы знаете, что эта сигнатура указывает на то, что возвращаемый из этой функции строковый литерал на самом деле заимствован из той же памяти, что и у переменной `request`. И следовательно, можно заключить, что функция не перераспределяет память ни под какие строковые литералы и просто переинтерпретирует память, положенную в основу структуры запроса NGINX.

Возвращаемый строковый срез гарантированно будет жить ровно столько, сколько переданная в него ссылка на запрос. Это имеет смысл, поскольку возвращаемый функцией строковый срез указывает на память, которая находится во владении структуры запроса NGINX. Было бы неправильно высвобождать запрос и удерживать ссылки на строковый литерал тела запроса. Применяемая в языке Rust система времен жизни используется здесь для валидации свойства исходного кода, которую в противном случае было бы трудно выразить, – свойства, касающегося того, как эти две части памяти напрямую взаимосвязаны между собой в иерархии. Тело запроса не может пережить

структуре запроса, и мы защищены от каких-либо допущений о том, что оно сможет это сделать.

Теперь давайте посмотрим на тело функции. Сначала рассмотрим вызов функции `std::slice::from_raw_parts` в середине функции, потому что она информирует обо всем остальном, что происходит. Прежде чем начать обследовать принцип работы этой функции, нужно поговорить о *срезах*.

Срез – это сплошной блок памяти, состоящий из смежно расположенных элементов одинакового типа, подобно массиву или вектору. Однако представление среза – это всего лишь указатель и длина, поэтому он может выступать в качестве дешевого «вида на» многочисленные опорные хранилища. По сути, это то же самое, что разница между типом `String` (владеемым, динамически расширяемым, мутируемым) и типом `&str` (только читаемым видом, возможно, на `String` или на `&'static str`). На рис. 4.8 показан строковый литерал типа `String` и несколько срезов типа `&str`, которые указывают на его подстроки.

```
let message = String::from("Hello world and all who inhabit it!");
```

```
let start = &message[0..11];
// start == "Hello world"
```

```
let end = &message[16..35];
// end == "who inhabit it!"
```

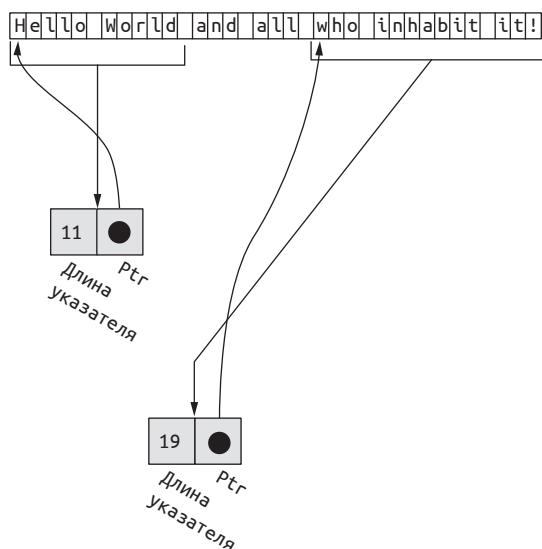


Рисунок 4.8 Использование нескольких срезов одного и того же строкового литерала

Вспомните, что в главе 3 удалось создать ссылку на строковый литерал (более правильно именуемую *строковым срезом*) из строкового литерала C (с концевым символом `null`). Память под тот срез компилятор Rust не перераспределял; просто был получен только читаемый вид на байты, переданные из C. Поскольку веб-сервер NGINX передает указатели в буферы тел запросов, строковый срез можно создать аналогичным образом, и он будет содержать тело запроса. Для этого сначала необходимо создать срез из необработанных байтов (тип для них записывается как `&[u8]`). Затем этот байтовый срез можно преобразовать

в строковый, предварительно убедившись, что он соответствует кодировке UTF-8 (требуется для всех строковых литералов Rust).

Для того чтобы сконструировать срез, используется метод, немнogo отличающийся от исходного кода конструирования строкового среза, который был написан в главе 3. В том исходном коде предполагалось, что будет передаваться строковый литерал с концевым символом `null`, и использовалась вспомогательная структура `CStr`. Однако NGINX не использует строковые литералы с концевым символом `null`; вместо этого он раздает начальный и конечный указатели. Следовательно, нужно использовать функцию немного более низкого уровня, `std::slice::from_raw_parts`. Эта функция берет начальный указатель и длину и конвертирует их в срез Rust.

Теперь, начиная с верхней части функции первое, что тут имеется, – это группа проверок на `null`. Однако в этих проверках на `null` можно заметить кое-что странное. Давайте посмотрим:

```
if request.request_body.is_null()
    || (*request.request_body).bufs.is_null()
    || (**request.request_body).bufs.buf.is_null()
{
    return Err("Request body buffers were
               not initialized as expected");
}
```

Первая проверка выглядит достаточно нормальной, но последующие проверки имеют несколько странный синтаксис. Скобки и звездочка – это то, как в языке Rust осуществляется доступ к полям структуры, находящейся за указателем. Это эквивалентно оператору `->` в языке C или C++; для такого доступа в языке Rust просто нет специального оператора!

Возможно, будет полезно взглянуть на структуру этих типов. Ниже приведен упрощенный взгляд на структуру, так как реальные используемые тут типы содержат огромное количество полей. Одна только `ngx_http_request_t` содержит до 144 полей (в зависимости от флагов компилятора)!

```
struct ngx_http_request_t {
    request_body: *mut ngx_http_request_body_t,
    ...
}

struct ngx_http_request_body_t {
    bufs: *mut ngx_chain_t,
    ...
}

struct ngx_chain_t {
    buf: *mut ngx_buf_t,
    ...
}
```

```
struct ngx_buf_t {
    last: *mut u_char,
    ...
}
```

В приведенном ниже примере показаны эквивалентные операции по созданию размещаемой в стеке структуры и распечатке элемента на основе указателя как на языке C, так и на языке Rust:

```
typedef struct { ← Код на языке C.
    x int
} foo_t;
foo_t foo = { 1 };
foo_t *foo_p = &foo;
printf("%d\n", foo_p->x);

struct Foo { ← Код на языке Rust.
    x: i32,
}
let foo = Foo { x: 1 };
let foo_p: *const Foo = &foo;
unsafe {
    println!("{}", (*foo_p).bar);
}
```

Приведенный выше исходный код достаточно разумен для доступа к одному полю, но он может стать немного громоздким при работе с более крупной структурой C, содержащей много вложенных полей-указателей. Заключительная проверка на `null` в функции-получателе тела имеет только два вложенных обращения к полю-указателю, и ее уже довольно сложно разобрать:

```
(**request.request_body).bufs.buf.is_null()
```

После проверки на `null` имеется новый вызов метода, и этот вызов раньше не встречался:

```
let len = (*buf).last.offset_from(start) as usize;
```

При конструировании строковых срезов из обычновенных указателей сначала необходимо создавать срез из байтов, используя функцию Rust `std::slice::from_raw_parts`. Эта функция принимает два аргумента: указатель на начало среза и длину среза. Веб-сервер NGINX предоставляет начальный и конечный указатели для своих строковых типов. Для получения длины занимаемой строковым литералом области памяти используется метод `offset_from` на любом указателе, чтобы получить смещение в памяти между конечным и начальным указателями. Если бы эта информация была нужна в C, то можно было бы использовать простые арифметические операции на указателях, но предоставляемые в языке Rust функции с указателями немного более наглядны. Следующие ниже функции C и Rust добиваются одной и той же цели отыскания размера блока памяти между двумя указателями:

```

ptrdiff_t offset(char *start, char *end) {
    end - start
}

fn offset(start: *const u8, end: *const u8) -> usize { ← Код на языке C.
    end.offset_from(start) as usize
} ← Код на языке Rust.

```

Возможно, вы заметили, что в исходном коде на языке Rust также используется тип `usize`, потому что если `start` больше `end`, то метод `offset_from` может возвращать отрицательное число, поэтому он возвращает `isize`. Тип `usize` – это беззнаковый тип размера указателя, а `isize` – его знаковый эквивалент. Функция `std::slice::from_raw_parts` нуждается в том, чтобы аргумент длины имел тип `usize`, так как конструирование среза памяти с отрицательной длиной не имеет никакого смысла. Поэтому необходимо конвертировать `isize` в `usize`, используя выражение приведения `as usize`. Поскольку `isize` гарантированно совпадает с размером `usize`, это приведение ничего не делает (но-оп) и никогда не приведет к отказу.

Функция `std::slice::from_raw_parts` уже обсуждалась; единственное, что осталось, – это исходный код, который преобразовывает байтовый срез в строковый. Функция `std::str::from_utf8` выполняет проверку на соответствие кодировке UTF-8 для среза байтов, и если проверка проходит, то возвращает строковый срез Rust.

После выполнения всего этого исходного кода и при условии, что ошибок не возникает, получится строковый срез, содержащий тело запроса, полученное HTTP-обработчиком веб-сервера NGINX. Разобравшись в том, как работает функция-обработчик, теперь давайте проверим, получится ли извлечь ожидаемые детали:

```

$ cargo build
$ ./nginx-1.19.3 objs/nginx -c nginx.conf -p ngx-run
....
Read request body: "3 4 * 2 -"

# Concurrently, in a separate window after NGINX is started
$ curl -X POST -d '3 4 * 2 -' http://localhost:8080/calculate
# this command will block forever

```

Получилось! Rust читает тело HTTP-запроса из веб-сервера NGINX. Еще не добавлен исходный код написания HTTP-ответа, поэтому консольный инструмент `curl` будет блокировать до тех пор, пока вы из него не выйдете, но вы приближаетесь к решению арифметических задач с помощью веб-сервера NGINX.

4.5 Использование библиотеки-калькулятора

Библиотека-калькулятор, которая рассматривалась в главе 3, уже написана, и ее можно использовать для решения такого же рода арифметических задач в формате обратной польской нотации, которые,

как мы ожидаем, эта конечная точка будет получать. Давайте попробуем добавить ее в проект обработчика NGINX и выполним несколько арифметических расчетов! Сперва нужно добавить пакет `calculate` в качестве зависимости для пакета обработчика. Откройте файл `Cargo.toml` в проекте обработчика и добавьте новую строку в секцию `[dependencies]`:

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

Обычно при управлении зависимостью с помощью Cargo зависимость извлекается из реестра crates.io. Поскольку публиковать библиотеку `calculate` пока нежелательно, данный пакет можно настроить как зависимость от пути. Таким образом, в качестве местоположения для поиска пакета менеджер пакетов Cargo будет искать по указанному пути, а не в реестре crates.io. Указанный здесь путь предусматривает, что уже имеется структура папок, которая выглядит следующим образом:

```
some_directory/
  calculate/
    Cargo.toml
    src/
      lib.rs
  ngx_http_calculator_rs/
    Cargo.toml
    src/
      lib.rs
```

Если ее нет, то для пакета `calculate` можно указать путь в кавычках как относительный либо как абсолютный путь к пакетному каталогу, сообразно обстоятельствам.

Далее можно вызвать функцию `evaluate` из пакета `calculate` изнутри функции-обработчика HTTP веб-сервера NGINX. Давайте посмотрим, как бы это выглядело:

```
unsafe extern "C" fn read_body_handler(r: *mut ngx_http_request_t)
{
    if r.is_null() {
        eprintln!("got null request in body handler");
        return;
    }

    let request = &*r;

    let body = match request_body_as_str(request) {
        Ok(body) => body,
        Err(e) => {
            eprintln!("failed to parse body: {}", e);
            return;
        }
    };
}
```

```

match calculate::evaluate(body) {
    Ok(result) => eprintln!("{} = {}", body, result),
    Err(e) => eprintln!("{} => error: {}", body, e),
}
}

```

Теперь давайте скомпилируем функцию-обработчик и попробуем ее выполнить:

```

$ cargo build
warning: The package `calculate` provides no linkable target.
The compiler might raise an error while compiling
`ngx_http_calculator_rs`. Consider adding 'dylib' or 'rlib' to
key `crate-type` in `calculate`'s Cargo.toml. This warning
might turn into a hard error in the future.

```

```

Compiling ngx_http_calculator_rs v0.1.0
error[E0433]: failed to resolve: use of undeclared type or
  module `calculate`
--> src/lib.rs:35:9
  |
35 | match calculate::evaluate(body) {
  |         ^^^^^^^^^ use of undeclared type or module `calculate`

```

Исходный код не компилируется! Почему? Если вы помните, в главе 3 менеджеру пакетов Cargo сообщалось о том, что пакет Rust **calculate** нужно компилировать как совместимую с языком С динамическую библиотеку. Это отлично работает для связывания с исходным кодом на C, но, как оказалось, не так хорошо работает для связывания с исходным кодом на Rust. Эту ошибку можно устраниить, сообщив Cargo о необходимости генерировать совместимый с Rust файл **rlib** в дополнение к файлу **dylib**. По умолчанию Cargo генерирует только файлы **rlib**, но если эту настройку переопределить, то указанная функциональность будет потеряна. Откройте файл **Cargo.toml** в пакете Rust **calculate** и отредактируйте поле **crate-type** под заголовком **[lib]**:

```

[lib]
crate-type = ["rlib", "cdylib"]

```

При такой настройке Cargo будет генерировать библиотечные файлы обоих типов, поэтому беспокоиться об утере какой-либо функциональности не нужно. Давайте попробуем выполнить результат этой компиляции еще раз:

```

$ cargo build
Compiling ngx_http_calculator_rs v0.1.0
error[E0603]: function `evaluate` is private
--> src/lib.rs:35:20
  |
35 |     match calculate::evaluate(body) {
  |             ^^^^^^^^^ private function
  |

```

```
note: the function `evaluate` is defined here
--> calculate/src/lib.rs:73:1
|
73 | fn evaluate(problem: &str) -> Result<i32, Errgor> {
| ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Теперь компиляция не проходит по другой причине: `evaluate` – это приватная функция. Вспомните, что при выставлении функции `solve` из Rust в C в объявление функции нужно было добавить ключевое слово `pub`, чтобы сообщить компилятору о том, что она должна быть видна за пределами пакета Rust. То же самое нужно сделать и с функцией `evaluate`. Определение нужно изменить, и оно должно выглядеть следующим образом:

```
pub fn evaluate(problem: &str) -> Result<i32, Errgor> {
```

Перезапуск компилятора выдает еще одну новую ошибку:

```
$ cargo build
   Compiling calculate v0.1.0
error[E0446]: private type `Errgor` in public interface
--> calculate/src/lib.rs:73:1
|
35 | enum Errgor {
|   - `Errgor` declared as private
...
73 | pub fn evaluate(problem: &str) -> Result<i32, Errgor> {
| ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
| can't leak private type
```

Когда элемент (функция, структура или перечисление) выставляется наружу в публичное пространство, компилятор пытается предотвратить создание непригодного для использования API. Например, эта функция помечается как публичная, но часть возвращаемого ею типа является приватной. Если бы кто-то захотел использовать эту функцию и произошла ошибка, то он не смог бы определить, что это была за ошибка. Такой результат был бы неблагоприятным, поэтому хорошо, что компилятор ее предотвратил.

Как вы, возможно, уже догадались, в целях устранения этой ошибки нужно пометить перечисление `Errgor` тоже как публичное. Определение перечисления `Errgor` теперь становится таким:

```
pub enum Errgor {
    InvalidNumber,
    PopFromEmptyStack,
}
```

После внесения этой правки можно перекомпилировать исходный код без единой ошибки:

```
$ cargo build
   Compiling calculate v0.1.0
   Compiling ngx_http_calculator_rs v0.1.0
Finished dev [unoptimized + debuginfo] target(s) in 6.75s
```

Если сравнивать с главой 3, то подавляющая часть файла `lib.rs` в пакете `calculate` остается неизменной; измененные строки показаны в следующем ниже листинге.

Листинг 4.14 Изменения, необходимые для пакета calculate

```
...
pub enum Error {
    InvalidNumber,
    PopFromEmptyStack,
}
...

pub fn evaluate(problem: &str) -> Result<i32, Error> {
    ...
}
```

В следующем ниже листинге показано, как должна выглядеть функция `read_body_handler` после завершения работы.

Листинг 4.15 Обработчик HTTP, печатающий результат арифметического выражения

```
unsafe extern "C" fn read_body_handler(r: *mut ngx_http_request_t)
{
    if r.is_null() {
        eprintln!("got null request in body handler");
        return;
    }

    let request = &*r;

    let body = match request_body_as_str(request) {
        Ok(body) => body,
        (e) => {
            eprintln!("failed to parse body: {}", e);
            return;
        }
    };

    match calculate::evaluate(body) {
        Ok(result) => eprintln!("{} = {}", body, result),
        Err(e) => eprintln!("{} => error: {}", body, e),
    }
}
```

Теперь, когда вы научились собирать обработчик HTTP вместе с библиотекой `calculate`, можно запустить веб-сервер NGINX с новой версией модуля:

```
$ cargo build
$ ./nginx-1.19.3/objs/nginx -c nginx.conf -p ngx-run
....
3 4 * 2 - = 10

# Concurrently, in a separate window after NGINX is started
$ curl -X POST -d '3 4 * 2 -' http://localhost:8080/calculate
# this command will block forever
```

Успех уже совсем близок! Вы связали язык C с языком Rust, вычи-
тали тело запроса из структуры HTTP-запроса веб-сервера NGINX,
реиспользовали существующую библиотеку `calculate` и решили ариф-
метическую задачу. Осталось только написать результат вычислений
в HTTP-ответ.

4.6 Написание HTTP-ответа

HTTP-ответ будет содержать результат арифметического выражения
в текстовой форме. В языке Rust переход от `i32` к `String` делается до-
статочно просто с помощью макрокоманды `format!`:

```
match calculate::evaluate(body) {
    Ok(result) => {
        eprintln!("{} = {}", body, result)

        let response_body = format!("{}", result);
    },
    Err(e) => eprintln!("{} => error: {}", body, e),
}
```

Немного сложнее перейти от этого строкового литерала к телу от-
вета веб-сервера NGINX. Нужно написать еще одну функцию, которая
создает несколько промежуточных структур и копирует память из
типа `String` в тип NGINX. Полное содержимое этой функции выгля-
дит следующим образом:

```
unsafe fn write_response(
    request: &mut ngx_http_request_t,
    response_body: &str,
    status_code: ngx_uint_t,
) -> Result<(), &'static str> {
    let headers = &mut request.headers_out;

    headers.status = status_code;

    let response_bytes = response_body.as_bytes();
    headers.content_length_n = response_bytes.len() as off_t;

    let rc = ngx_http_send_header(request); ←
    if rc != 0 {                                | Записывает код
        return Err("failed to send headers");
    }
}
```

```

let buf_p =
    ngx_pcalloc(request.pool, std::mem::size_of::<
        ngx_buf_t>() as size_t) ← Создает «буфер»
        as *mut ngx_buf_t;
if buf_p.is_null() {
    return Err("Failed to allocate buffer");
}

let buf = &mut (*buf_p);

buf.set_last_buf(1); ← Конфигурирует буфер
.set_last_in_chain(1); ← под уборку.
buf.set_memory(1);

let response_buffer =
    ngx_pcalloc(request.pool, response_bytes.len() as size_t); ← Резервирует
if response_buffer.is_null() { память под
    return Err("Failed to allocate response buffer"); строковый бу-
} фер для хране-
ния ответа.

std::ptr::copy_nonoverlapping( ← Копирует тело
    response_bytes.as_ptr(),
    response_buffer as *mut u8,
    response_bytes.len(),
);
buf.pos = response_buffer as *mut u8;
buf.last = response_buffer.offset(
    response_bytes.len() as isize) as *mut u8;

let mut out_chain = ngx_chain_t {
    buf,
    next: std::ptr::null_mut(),
}; ← Передает ответ в обра-
botчики вывода веб-
сервера NGINX.

if ngx_http_output_filter(request, &mut out_chain) != 0 {
    return Err("Failed to perform http output filter chain");
}

Ok(())
}

```

Сейчас эта функция выполняет много разных вещей, поэтому давайте рассмотрим их все. Указанная функция выполняет следующие высокоуровневые действия:

- 1 записывает код статуса HTTP и заголовок Content-Length (длина содержимого);
- 2 создает «буферный» объект NGINX;
- 3 конфигурирует буфер NGINX таким образом, чтобы он правильно высвобождался веб-сервером NGINX;
- 4 резервирует память под строковый буфер под хранение тела ответа;

- 5 копирует байты тела ответа из строкового среза Rust в буфер NGINX;
- 6 передает буфер с телом ответа в обработчики HTTP-вывода в рамках веб-сервера NGINX.

Порядок действий для операций с HTTP-ответами довольно стандартен. Сначала необходимо записать заголовки ответа:

```
let headers = &mut request.headers_out;
headers.status = status_code;
```

`headers_out` – это поле переменной `request`, содержащее информацию о заголовках, которые будут выводиться клиенту вместе с HTTP-ответом.

```
let response_bytes = response_body.as_bytes();
headers.content_length_n = response_bytes.len() as off_t;

let rc = ngx_http_send_header(request);
if rc != 0 {
    return Err("failed to send headers");
}
```

`off_t` – это тип смещения указателя, который берется из автоматически генерированных привязок к NGINX; это не стандартный тип Rust.

Каждый строковый литерал Rust (и строковый срез) является коллекцией байтов, которые образуют допустимый текст в кодировке UTF-8. Переход от строкового представления к байтовому срезу осуществляется, используя метод `as_bytes`.

Каждый HTTP-ответ начинается со строки, содержащей версию протокола и код статуса, за которой следует ряд строк, содержащих данные заголовков. Заголовок `Content-Length` всегда должен быть задан, когда предоставляется тело ответа, в котором не используется фрагментированная кодировка ответа. Поэтому прежде чем можно будет что-либо сделать с текстом тела ответа, необходимо записать код статуса и длину содержимого. Код статуса предоставляется этой функции в качестве аргумента, а длина содержимого может быть рассчитана на основе количества байтов в строковом литерале тела ответа. После установки этих двух значений вызывается функция `ngx_http_send_header`, которая записывает данные заголовков в сетевое соединение.

Далее резервируется память под буфер `ngx_buf_t` для хранения информации о буфере ответа. Давайте посмотрим на эту часть исходного кода:

```
let buf_p =
    ngx_pcalloc(request.pool, std::mem::size_of::<
        ngx_buf_t>() as size_t)
    as *mut ngx_buf_t;

if buf_p.is_null() {
    return Err("Failed to allocate buffer");
}

let buf = &mut (*buf_p);
```

```
buf.set_last_buf(1);
buf.set_last_in_chain(1);
buf.set_memory(1);
```

Сначала применяется функция `ngx_malloc`. Это предоставляемая веб-сервером NGINX функция резервирования памяти аналогична стандартной функции `malloc` в языке С. Для резервирования запрашиваемого объема памяти используется пул памяти, локальный для каждого объекта-запроса.

Указанные пулы памяти обеспечивают механизм, очень похожий на применяемую в языке Rust систему владения, но они специфичны для NGINX и требуют больше работы во время исполнения. При высвобождении каждого пула тот высвобождает свое содержимое, поэтому по завершении обработки запроса все созданные в пуле временные буферы будут высвобождены. Это высвобождение позволяет авторам плагинов резервировать память с тем же временем жизни, что и у самого запроса, не слишком беспокоясь о настройке дополнительного кода уборки.

Здесь представлено несколько новых концепций языка Rust; первая из них – функция `std::mem::size_of<T>`. Эта функция возвращает размер в байтах любого типа, который ей передается в позиции аргумента типа. За счет этого есть возможность сообщать менеджеру памяти NGINX о том, сколько байтов он должен резервировать под безопасное хранение буфера. После проверки на `null` выполняется муттируемое перезаимствование только что размещенного в памяти указателя, поэтому отпадает необходимость брать значение по этому указателю всякий раз, когда требуется его использовать.

Наконец, используется несколько функций `set_`, чтобы инициализировать несколько настроек, которые сообщают веб-серверу NGINX о том, как должен обрабатываться буфер. Точный смысл этих функций довольно специфичен для NGINX, но в них есть кое-что интересное для данного контекста. Для того чтобы понять, нужно посмотреть на определение следующих ниже полей у типа `ngx_buf_t`:

```
struct ngx_buf_t {
    ... (несколько полей опущено)
    unsigned memory:1;
    unsigned last_buf:1;
    unsigned last_in_chain:1;
};
```

Три вызываемые функции `set_` (`set_last_buf`, `set_memory` и `set_last_in_chain`) соответствуют битовым полям (`last_buf`, `memory` и `last_in_chain`) в конце типа `ngx_buf_s`. Инструмент Rust `bindgen` генерирует функции `set` и `get` для этих битовых полей, так как Rust изначально их не поддерживает. Кроме этих функций, адекватный способ взаимодействия с указанными битовыми полями отсутствует.

Следующая часть функции довольно проста: резервируется блок памяти под хранение тела ответа, и в этот блок копируются данные

из строкового среза Rust. Технически есть возможность просто передать указатель на этот срез веб-серверу NGINX, но Rust высвободит строковый литерал, когда выйдет за пределы диапазона доступности, и указатель станет недействительным. Этот строковый литерал нужно переразместить в буфере, которым владеет NGINX, так как это проще, чем пытаться координировать применяемую в языке Rust систему владения с NGINX (это вполне возможно, но здесь данная тема рассматриваться не будет):

```
let response_buffer =
    ngx_pcalloc(request.pool, response_bytes.len() as size_t);

if response_buffer.is_null() {
    return Err("Failed to allocate response buffer");
}

std::ptr::copy_nonoverlapping(
    response_bytes.as_ptr(),           ←
    response_buffer as *mut u8,        | Использует метод as_ptr для получения
    response_bytes.len(),            | указателя на первый элемент в срезе.
);

```

Как и раньше, здесь используется та же функция `ngx_pcalloc`, но на этот раз не нужно использовать `std::mem::size_of`, потому что память резервируется под известное количество байтов, а не под экземпляры сложного типа. Функция `std::ptr::copy_nonoverlapping` работает так же, как функция `memcpuy` стандартной библиотеки C, с зеркальным порядком указателей на источник и адресата. Она копирует каждый байт из строкового среза Rust в только что зарезервированный буфер.

После копирования данных выполняется заключительная настройка, а затем завершенный запрос передается обратно веб-серверу NGINX, чтобы он мог выполнить необходимые операции ввода-вывода для отправки данных по сети:

```
buf.pos = response_buffer as *mut u8;
buf.last = response_buffer.offset(
    response_bytes.len() as isize) as *mut u8;

let mut out_chain = ngx_chain_t {
    buf,
    next: std::ptr::null_mut(),
};

```

Здесь устанавливаются соответствующие поля структуры `ngx_buf_t` в качестве начального и конечного указателей блока памяти, под который только что была зарезервирована память. Для того чтобы получить конечный указатель для блока памяти, нужен новый метод `.offset`. Указанный метод в основном противоположен методу `offset_from`, который возвращает разницу между двумя указателями. Метод `.offset` принимает указатель и число N и возвращает новый указатель, который находится на расстоянии N указателей от базово-

го указателя. На рис. 4.9 показано дерево решений, которое можно использовать для выбора метода, наиболее подходящего для варианта использования.

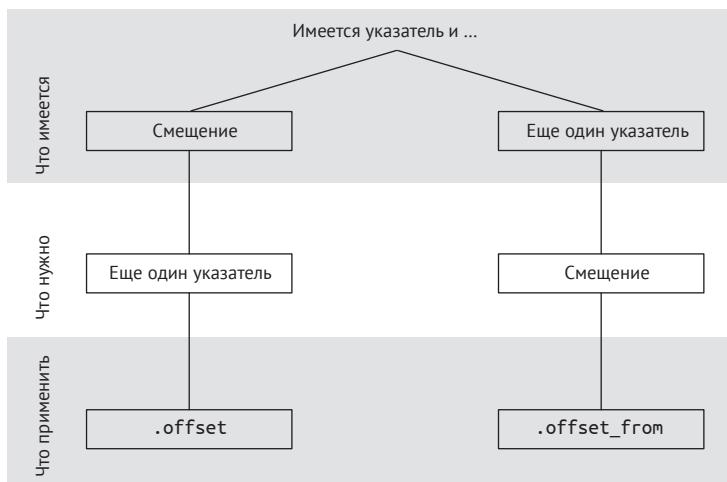


Рисунок 4.9 Выбор между методами конвертации указатель/смещение

Буфер помещается в `ngx_chain_t`. Этот тип, по сути, представляет собой связный список блоков памяти. Поскольку есть только один блок, инициализируется цепочка с одним-единственным буфером и указателем `null` в слоте, который в противном случае указывал бы на следующий элемент в цепочке. Наконец, когда все конфигурационные настройки сделаны и буферы заполнены данными, можно сообщить веб-серверу NGINX о том, чтобы тот начал писать данные ответа клиенту:

```

if ngx_http_output_filter(request, &mut out_chain) != 0 {
    return Err("Failed to perform http output filter chain");
}
Ok(())
  
```

Функция `ngx_http_output_filter` принимает указатель на запрос и указатель на `ngx_chain_t` и обрабатывает процесс записи данных ответа клиенту. После вызова этой функции возвращается `Ok()`, чтобы сообщить источнику вызова, что все прошло так, как положено.

Теперь можно вызвать ее из функции `read_body_handler`:

```

unsafe extern "C" fn read_body_handler(r: *mut ngx_http_request_t)
{
    if r.is_null() {
        eprintln!("got null request in body handler");
        return;
    }
  
```

```

let request = &mut *r; ←
    Получатель указателя запроса должен пре-
    вратиться в мутируемое перезаместова-
    ние, что позволит позже подвергать поля
    запроса мутации.

let body = match request_body_as_str(request) {
    Ok(body) => body,
    Err(e) => {
        eprintln!("failed to parse body: {}", e);
        return;
    }
};

match calculate::evaluate(body) {
    Ok(result) => {
        let response_body = format!("{}", result);

        match write_response(request, &response_body, 200) {
            Ok(_) => {}
            Err(e) => {
                eprintln!("failed to write HTTP response: {}", e);
            }
        }
    }
    Err(e) => eprintln!("{} => error: {}", body, e),
}
}

```

Давайте перекомпилируем приведенный выше исходный код и пробуем применить HTTP-обработчик:

```

$ cargo build
$ ./nginx-1.19.3/objs/nginx -c nginx.conf -p ngx-run
....
# Конкурентно, в отдельном окне после запуска веб-сервера NGINX
$ curl -X POST -d '3 4 * 2 -' http://localhost:8080/calculate; echo
10

```

Дополнительная команда echo здесь присутствует в связи с тем, что в конце HTTP-ответа нет символа новой строки, поэтому без команды echo, которая добавляет символ новой строки, будет трудно увидеть результат работы консольного инструмента curl.

Получилось! Вы успешно создали пакет Rust, который предоставляет HTTP-обработчик NGINX, выполняющий арифметические расчеты. Для этого потребовалось выполнить немало шагов и внести немало изменений в файлы исходного кода. В следующем ниже листинге представлена окончательная версия того, как должен выглядеть файл `lib.rs` в проекте.

Листинг 4.16 Полный HTTP-обработчик калькулятора

```

include!(concat!(env!("OUT_DIR"), "/nginx.rs"));

#[no_mangle]

```

```

pub unsafe extern "C" fn ngx_http_calculator_handler(
    r: *mut ngx_http_request_t,
) -> ngx_int_t {
    let rc = ngx_http_read_client_request_body(
        r, Some(read_body_handler));
    if rc != 0 {
        return rc;
    }

    0
}

unsafe extern "C" fn read_body_handler(r: *mut ngx_http_request_t)
{
    if r.is_null() {
        eprintln!("got null request in body handler");
        return;
    }

    let request = &mut *r;

    let body = match request_body_as_str(request) {
        Ok(body) => body,
        Err(e) => {
            eprintln!("failed to parse body: {}", e);
            return;
        }
    };

    match calculate::evaluate(body) {
        Ok(result) => {
            let response_body = format!("{}", result);

            match write_response(request, &response_body, 200) {
                Ok(_) => {}
                Err(e) => {
                    eprintln!("failed to write HTTP response: {}", e);
                }
            }
        }
        Err(e) => eprintln!("{} => error: {}", body, e),
    }
}

unsafe fn request_body_as_str<'a>(
    request: &'a ngx_http_request_t,
) -> Result<&'a str, &'static str> {
    if request.request_body.is_null()
        || (*request.request_body).bufs.is_null()
        || (*(*request.request_body).bufs).buf.is_null()
    {
        return Err("Request body buffers were not
                  initialized as expected");
    }
}

```

```
let buf = (*(*request.request_body).bufs).buf;

let start = (*buf).pos;
let len = (*buf).last.offset_from(start) as usize;

let body_bytes = std::slice::from_raw_parts(start, len);

let body_str = std::str::from_utf8(body_bytes)
    .map_err(|_| "Body contains invalid UTF-8")?;

Ok(body_str)
}

unsafe fn write_response(
    request: &mut ngx_http_request_t,
    response_body: &str,
    status_code: ngx_uint_t,
) -> Result<(), &'static str> {
    let headers = &mut request.headers_out;

    headers.status = status_code;

    let response_bytes = response_body.as_bytes();
    headers.content_length_n = response_bytes.len() as off_t;

    let rc = ngx_http_send_header(request);
    if rc != 0 {
        return Err("failed to send headers");
    }

    let buf_p =
        ngx_pcalloc(request.pool, std::mem::size_of::<
            ngx_buf_t>() as size_t)
        as *mut ngx_buf_t;
    if buf_p.is_null() {
        return Err("Failed to allocate buffer");
    }

    let buf = &mut (*buf_p);

    buf.set_last_buf(1);
    buf.set_last_in_chain(1);
    buf.set_memory(1);

    let response_buffer =
        ngx_pcalloc(request.pool, response_bytes.len() as size_t);
    if response_buffer.is_null() {
        return Err("Failed to allocate response buffer");
    }

    std::ptr::copy_nonoverlapping(
        response_bytes.as_ptr(),
```

```

    response_buffer as *mut u8,
    response_bytes.len(),
);

buf.pos = response_buffer as *mut u8;
buf.last = response_buffer.offset(
    response_bytes.len() as isize) as *mut u8;

let mut out_chain = ngx_chain_t {
    buf,
    next: std::ptr::null_mut(),
};

if ngx_http_output_filter(request, &mut out_chain) != 0 {
    return Err("Failed to perform http output filter chain");
}

Ok(())
}

```

В этих 127 строках исходного кода на языке Rust содержится много новых идей, но в нем также можно найти много устаревших идиом языка С. Временные буферы, незащищенные вызовы функций и ряд других вещей, которые не появляются в обычном исходном коде на языке Rust, встроены непосредственно в функции-обработчики. В следующей далее главе рассматриваются методы, которые используются для организации более крупных файлов исходного кода на языке Rust в обособленные модули.

Краткий итог

- Инструмент `bindgen` используется для генерирования привязок языка Rust к исходному коду на языках С и С++.
- Скрипты сборки позволяют разработчикам писать исходный код на Rust, который исполняется во время компиляции.
- Макрокоманда `include!` вставляет текстовый файл в файлы исходного кода Rust во время компиляции и компилирует его как исходный код Rust.
- Не все функции `extern "C"` должны помечаться как `#[no_mangle]`.
- Перезаимствование позволяет трактовать обычновенные указатели как стандартные для Rust ссылки.
- Метод `.offset_from` возвращает разницу в байтах между двумя указателями.
- Функция `std::slice::from_raw_parts` конструирует вид на сплошной блок памяти на основе указателя и длины.
- Зависимости от пути используются менеджером пакетов Cargo для включения пакетов, которые находятся на компьютере, а не закачиваются в реестр crates.io.
- Пакеты могут компилироваться как `rlib` (для Rust) и как `cdylib` (для С).

- Если пометить элемент как `pub`, то компилятор будет ожидать, что все типы, которые являются частью его публичного API, также являются `pub`.
- Инструмент `bindgen` автоматически создает функции `get_` и `set_` для битовых полей С.
- Срезы – это сплошные заимствованные виды на область памяти.
- Метод `.as_ptr` возвращает указатель на первый элемент в срезе.
- Метод `.offset` возвращает указатель, который находится на расстоянии N элементов от базового указателя в сплошном блоке.
- Функция `std::mem::size_of` – это применяемый в языке Rust эквивалент функции `sizeof`.
- Функция `std::ptr::copy_nonoverlapping` – это применяемый в языке Rust эквивалент функции `memcpy`.

5

Структурирование библиотек языка Rust

Эта глава охватывает следующие ниже темы:

- организация исходного кода на языке Rust посредством модулей;
- понимание принципа работы путей в отношении модулей Rust;
- работа с правилами видимости.

Практически все языки программирования имеют функциональные компоненты, позволяющие делить исходный код на группы составляющих. До сих пор во всех примерах исходного кода, которые вы видели, использовалось единое пространство имен. В этой главе будет рассмотрена принятая в языке Rust мощная система модулей и способы ее использования для структурирования своих пакетов.

5.1 Модули

В языке Rust *модуль* – это контейнер для содержания в них элементов. Элемент – это компонент пакета, такой как функция, структура, перечисление или тип (есть и другие, но давайте пока остановимся на них). Вы уже использовали модули из стандартной библиотеки, когда импортировали общую черту `Display` из модуля `fmt` в пакете `std`. Пакет `std` – это стандартная библиотека Rust, а модуль `fmt` содержит

элементы, которые помогают с форматированием текста, такие как общие черты `Display` и `Debug`.

Представьте себе, что требуется организовать небольшую программу, которая получает имя пользователя, а затем приветствует его и прощается с ним. Создайте новый проект Cargo под названием `greetings` и добавьте следующий ниже исходный код в файл `src/main.rs`.

Листинг 5.1 Исходный код получения имени пользователя и его приветствия

```
use std::io::stdin;

fn main() {
    let name = get_name();

    hello(&name);
    goodbye(&name);
}

fn get_name() -> String {
    let mut name = String::new();

    println!("Please enter your name");
    stdin().read_line(&mut name).unwrap(); ←
        name
}

fn goodbye(name: &str) {
    println!("Goodbye, {}", name);
}

fn hello(name: &str) {
    println!("Hello, {}", name);
}
```

Функция `read_line` читает строку текста из STDIN и копирует ее в буфер типа `String`.

Если ее выполнить, то можно будет увидеть, что получилась очень вежливая программа:

```
$ cargo run
Please enter your name
Thalia
Hello, Thalia

Goodbye, Thalia
```

Возможно, возникнет потребность организовать эти функции в два модуля – один для функций ввода, таких как `get_name`, и один для функций вывода, таких как `hello` и `goodbye`. Модули создаются в исходном коде Rust, используя ключевое слово `mod`, за которым следует имя модуля, а затем содержимое модуля в фигурных скобках (`{}`).

Давайте прямо сейчас создадим модули `input` и `output`.

Листинг 5.2 Программа приветствия пользователей с добавленными модулями

```
fn main() {
    let name = get_name();
    hello(&name);
    goodbye(&name);
}

mod input {
    use std::io::stdin;

    fn get_name() -> String {
        let mut name = String::new();

        println!("Please enter your name");
        std::io::stdin().read_line(&mut name).unwrap();

        name
    }
}

mod output {
    fn goodbye(name: &str) {
        println!("Goodbye, {}", name);
    }

    fn hello(name: &str) {
        println!("Hello, {}", name);
    }
}
```

Если попытаться выполнить ее сейчас, то возникнут три ошибки компилятора:

```
$ cargo run
error[E0425]: cannot find function `get_name` in this scope
--> src/main.rs:2:14
  |
2 |     let name = get_name();
  |     ^^^^^^^^^ not found in this scope
  |
help: consider importing this function
  |
1 | use input::get_name;
  |
... (same error for `hello` and `goodbye`)
```

К счастью, эти сообщения об ошибках сопровождаются подсказками о том, как их устраниТЬ. Поскольку все функции были помещены в модули `input` и `output`, они больше не находятся в том же пространстве имен, что и функция `main`. Эта проблема решается несколькими способами, один из которых описан в справке, которую предоставляет

компилятор. Для того чтобы импортировать функции `get_name`, `hello` и `goodbye` из их модулей, над функцией `main` добавляют инструкцию `use`.

Пока что давайте вставим инструкции `use`, которые указал компилятор. Для модуля `output` даже можно объединить две инструкции в одну.

Листинг 5.3 Программа приветствия пользователей с добавленными инструкциями `use`

```
use input::get_name;
use output::{goodbye, hello};

fn main() {
    let name = get_name();

    hello(&name);
    goodbye(&name);
}

...
```

Попробуем выполнить исходный код еще раз:

```
$ cargo run
error[E0603]: function `get_name` is private
  --> src/main.rs:1:19
   |
1 | use input::get_name;
   |          ^^^^^^^^^^ private function
   |
note: the function `get_name` is defined here
  --> src/main.rs:14:3
   |
14 | fn get_name() -> String {
   | ^^^^^^^^^^^^^^^^^^^^^^^^^^
... (same error for `hello` and `goodbye`)
```

Теперь компилятор способен разобраться в именах, но инструкции `use` вызывают ошибки, так как делается попытка импортировать приватные функции. Вспомните из главы 3, что все функции в Rust по умолчанию являются приватными и должны помечаться как публичные явным образом. Для этого перед определениями функций нужно добавлять ключевое слово `pub`. Давайте сейчас это сделаем.

Листинг 5.4 Программа приветствия пользователей с публичными функциями в модулях

```
...
mod input {
    use std::io::stdin;

    pub fn get_name() -> String {
```

```

let mut name = String::new();

println!("Please enter your name");
stdin().read_line(&mut name).unwrap();

name
}

}

mod output {
    pub fn goodbye(name: &str) {
        println!("Goodbye, {}", name);
    }

    pub fn hello(name: &str) {
        println!("Hello, {}", name);
    }
}

```

Теперь можно выполнить программу, и она будет работать так же, как и изначально:

```
$ cargo run
Please enter your name
Ругамус
Hello, Ругамус

Goodbye, Ругамус
```

5.1.1 Какая разница?

Выше удалось повторить функциональность изначальной программы, добавив гораздо больше синтаксиса. Ну и что? Зачем кому-то понадобилось бы утруждать себя добавлением ключевых слов `mod`, `use` и `pub` во весь свой исходный код, вместо того чтобы поместить их все в один большой модуль? Для многих людей думать о нескольких взаимосвязанных функциях в одном модуле проще, чем думать о всех функциях программы сразу. Если вы имеете дело с дефектом во взаимодействии программы с базой данных, то его будет легче отследить, если весь исходный код базы данных находится в одном месте, а не перемешан с исходным кодом HTTP, журналирования, хронометража или многопоточности исполнения в одном глобальном пространстве имен. На практике обычно принято сортировать соотнесенные элементы по группам и категориям; модули – это просто то, как это делается на языке Rust. На рис. 5.1 показан график модулей в программе приветствия пользователей.

Кроме того, можно создавать модули, которые хранятся в самостоятельных файлах. Давайте посмотрим, как это делается.

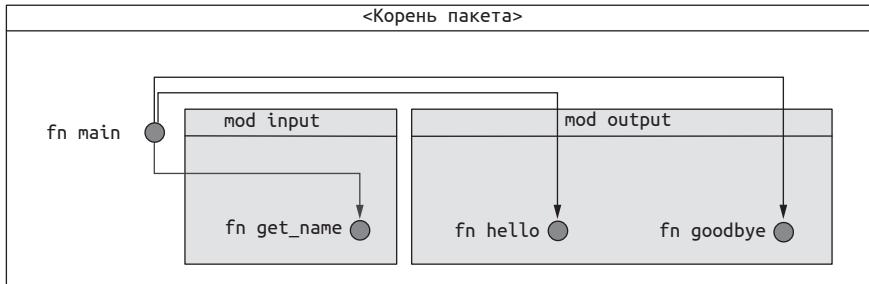


Рисунок 5.1 График программы приветствия пользователей

5.1.2 Несколько файлов

Прямо сейчас модули `input` и `output` находятся в одном и том же файле `main.rs`, как и остальная часть исходного кода. Если модули не очень маленькие, то обычно считается лучшим образом практики размещать модули внутри самостоятельных файлов. Для этого создается новый файл под названием `module.rs`, где имя `module` заменяется на имя создаваемого модуля. Для целей данного изложения давайте создадим `input.rs` и `output.rs`.

Листинг 5.5 Программа приветствия пользователей: main.rs

```
use input::get_name;
use output::{goodbye, hello};

mod input; <-- Инструкции mod изменены незначительным образом. Они
mod output; перемещены в начало файла, что является вопросом вы-
fn main() {   бора стиля, а в отношении содержимого фигурные скобки
    let name = get_name();                         убираются в пользу точки с запятой, что указывает на то,
                                                что для этого модуля используется не блок, а файл.

    hello(&name);
    goodbye(&name);
}
```

Листинг 5.6 Программа приветствия пользователей: input.rs

```
use std::io::stdin;

pub fn get_name() -> String {
    let mut name = String::new();

    println!("Please enter your name");
    stdin().read_line(&mut name).unwrap();

    name
}
```

Листинг 5.7 Программа приветствия пользователей: output.rs

```
pub fn goodbye(name: &str) {
    println!("Goodbye, {}", name);
}

pub fn hello(name: &str) {
    println!("Hello, {}", name);
}
```

После этих изменений программа по-прежнему функционирует, как положено:

```
$ cargo run
Please enter your name
world
Hello, world

Goodbye, world
```

ПРИМЕЧАНИЕ. Во многих языках программирования для выстраивания иерархии модулей неявно используется структура файловой системы. Компилятор Rust нуждается в том, чтобы в исходном коде указывалась инструкция `mod`, сообщающая ему о том, в какие файлы следует заглядывать. Для того чтобы сообщить компилятору Rust о файле `src/bananas.rs`, необходимо в корень пакета вставить `mod bananas`. Если бы возникла потребность поместить `bananas.rs` в модуль `forest`, то нужно было бы поместить его в `src/forest/bananas.rs`; файл `src/forest.rs` должен был бы содержать `mod bananas`, а `mod forest` должен был бы находиться в корне пакета.

Важно отметить, что с точки зрения компилятора между модулями, в которых используется блочный синтаксис (`mod my_mod { ... }`), и модулями, в которых используются обособленные файлы исходного кода (`mod my_mod;`), нет никакой разницы. Они оба обеспечивают одинаковую степень изоляции; единственное, что их отличает, – это стиль, который программист в них видит.

Одна из полезных стилистических причин размещения модулей в самостоятельных файлах заключается в том, что некоторые разработчики считают полезным иметь возможность переходить к определенным файлам с известным содержимым. Например, в большинстве текстовых редакторов проще открывать файл с именем `http.rs`, чем искать модуль с именем `http` в файле `lib.rs` длиной 10 000 строк.

Разбив исходный код на модули, теперь давайте посмотрим, как он изменится при добавлении новых функциональных компонентов. Представьте себе, что нужно обновить программу возможностью спрашивать пользователя о том, хорошо ли он провел день, и отвечать соответствующим образом. На высоком уровне можно создать элементы, которые будут выглядеть следующим образом:

```

enum DayKind {
    Good,
    Bad,
}

fn get_day_kind() -> DayKind {
    ...
}

fn print_day_kind_message(day_kind: DayKind) {
    ...
}

```

К чему относятся эти элементы при текущей настройке исходного кода? Функция `get_day_kind`, вероятно, относится к модулю `input`, поскольку она принимает входные данные от пользователя, а функция `print_day_kind_message` соответственно относится к модулю `output`, так как она отправляет сообщение пользователю. К чему же тогда относится перечисление `DayKind`? Оно напрямую не связано ни с вводом, ни с выводом, поэтому концептуально оно не относится ни к одному из них. Давайте создадим для него новый модуль и назовем его `day_kind`; он перейдет в `day_kind.rs`, и единственное, что в нем будет, – это новое перечисление. Кроме того, нужно добавить `mod day_kind;` в файл `main.rs`. Теперь эти файлы должны выглядеть следующим образом.

Листинг 5.8 Перечисление DayKind в файле main.rs

```

use input::get_name;
use output::{goodbye, hello};

mod day_kind;
mod input;
mod output;

fn main() {
    let name = get_name();

    hello(&name);
    goodbye(&name);
}

```

Листинг 5.9 Перечисление DayKind в файле day_kind.rs

```

pub enum DayKind { ←
    Good,
    Bad,
}

```

Перечисление `DayKind` теперь является публичным, вследствие чего к нему можно обращаться из других модулей в пакете.

Теперь давайте напишем функцию вывода, которая отвечает за печать сообщения пользователю о том, как прошел его день. Она будет написана в файле `output.rs`.

Листинг 5.10 Перечисление DayKind в файле output.rs

```
use day_kind::DayKind;

pub fn print_day_kind_message(day_kind: DayKind) {
    match day_kind {
        DayKind::Good => println!("I'm glad to hear you're having a good
day!"),
        DayKind::Bad => println!("I'm sorry to hear you're having a bad
day"),
    }
}
```

Сейчас давайте попробуем исполнить программу:

```
$ cargo run
error[E0432]: unresolved import `day_kind`
--> src/output.rs:1:5
  |
1 | use day_kind::DayKind;
  |     ^^^^^^^^ help: a similar path exists: `crate::day_kind`
```

Исходный код не компилируется. Компилятор предоставляет текстовую справку, которая поможет его скомпилировать, но перед этим давайте немного углубимся в механизм обработки путей языком Rust.

5.2 Пути

В языке Rust на все, что имеет имя (переменная, функция, структура, перечисление, тип и т. д.), можно ссылаться с помощью пути. Путь – это последовательность имен, которые называются отрезками пути, разделенными символами `::`, в совокупности ссылающихся на элемент или переменную (если путь содержит только один отрезок). В следующем ниже листинге показано несколько примеров.

Листинг 5.11 Примеры путей

```
fn main() {
    let value = true;

    // Все, что находится ниже, является путями
    value;           Путь к значению
                     локальной булевой
                     переменной.

    hello;          Путь к функции hello,
                    определенной сразу
                    под функцией main.

    std::io::stdin; Путь к функции stdin
                    в модуле стандартной
                    библиотеки.

    std::collections::hash_map::ValuesMut::<i32, String>::len; <-- Путь к функции len в итераторе ValuesMut для хеш-
    }                  словаря, содержащего ключи типа i32 и значения типа
                      String из модуля hash_map внутри модуля collections
                      стандартной библиотеки.

fn hello() { }
```

Как вы видите, пути могут быть очень маленькими или очень большими, но все они являются путями. Если попытаться собрать эту программу, то компилятор даже выдаст предупреждение о том, что все инструкции содержат только пути (то есть она ничего не делает):

```
$ cargo build
warning: path statement with no effect
--> src/main.rs:5:3
|
5 |     value;
|     ^^^^^^
|
|= note: `#[warn(path_statements)]` on by default

warning: path statement with no effect
--> src/main.rs:7:3
|
7 |     hello;
|     ^^^^^^

warning: path statement with no effect
--> src/main.rs:9:3
|
9 |     std::io::stdin;
|     ^^^^^^^^^^^^^^^

warning: path statement with no effect
--> src/main.rs:11:3
|
11 |     std::collections::hash_map::ValuesMut::<i32, String>::len;
|     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Появление предупреждений компилятора обусловлено тем, что пути сами по себе не слишком полезны. Путь к функции в строке кода сам по себе бесполезен; он полезен только тогда, когда эта функция вызывается фактически. Путь к структуре бесполезен (равно как и синтаксис недопустим); он полезен только при конструировании экземпляра этой структуры или вызове связанной с ней функции.

Пути содержат важный подвох, который сбивает с толку многих начинающих разработчиков на языке Rust, – тонкое различие между относительными и абсолютными путями.

5.2.1 Относительные и абсолютные пути

Относительные пути, такие как `hello` в листинге 5.11, касаются переменных или элементов в текущем пространстве имен, а абсолютные пути, такие как `std::io::stdin`, касаются переменных или элементов относительно корня пакета.

Пути в языке Rust полезно сравнивать с путями в файловой системе. Пути в языке Rust разделены на пакеты (которые всегда появляются в корне абсолютных путей) и модули (которые могут появляться,

а могут и не появляться в путях). Это похоже на то, как конструируют-
ся пути в операционных системах Windows. В относительных путях
используются имена только каталогов и файлов, чтобы указывать мес-
тотоположение относительно рабочего каталога, но абсолютные пути
имеют корень на определенном диске ввода-вывода, например `C:`. Раз-
личие между дисками и каталогами в Windows аналогично различию
между пакетами и модулями в Rust.

ПРИМЕЧАНИЕ. В Unix-подобных операционных системах все
пути, как правило, начинаются с символа `/`, обозначающего ко-
рень файловой системы, а файлы и папки растут оттуда вниз.
Система пространств имен в языке Rust не так проста, как там.

Когда нужно использовать абсолютный путь, чтобы ссылаться на
элементы в текущем каталоге, нужно использовать ключевое слово
`crate`, которое представляет собой особый отрезок пути, обозна-
чающий корень текущего каталога. Можно использовать еще один
особый отрезок пути, который называется `super`. Он используется
в относительных путях, для того чтобы ссылаться на пространство
имен, расположенное над текущим пространством имен. Давайте
рассмотрим небольшой пример, чтобы увидеть относительные и аб-
солютные пути в действии. Представьте себе, что вы пишете вы-
мышленный пакет Rust `libsнак`, в котором есть функции и типы для
приобретения и употребления вкусных закусок. В настоящее время
в `libsнак` есть файл `lib.rs`.

Листинг 5.12 Пакет libsнак

```
pub mod treats {
    pub mod shop {}

    pub enum Treat {
        Candy,
        IceCream,
    }

    pub struct ConsumedTreat {
        treat: Treat,
    }
}
```

Обратите внимание, что этот пример содержит модули, декориро-
ванные ключевым словом `pub`. Ключевое слово `pub` добавляется к мо-
дулям так же, как и к функциям, структурам или перечислениям. Для
модулей это означает то же самое, что и для других элементов. Доступ
к модулю без ключевого слова `pub` перед его определением возможен
только из того модуля, в котором он был объявлен. Если бы модуль
`shop` в листинге 5.12 не был `pub`, то к нему невозможно было бы полу-
чить доступ из корня пакета. Доступ к нему был бы возможен только
из модуля `treats`.

Представьте себе, что требуется добавить следующие ниже три функции в модули пакета `libsnack`, чтобы выполнять важнейшие операции по перекусыванию на скорую руку. Функция `buy` будет находиться в модуле `treats::shop`:

```
fn buy() -> Treat
```

Функция `eat` будет помещена в модуль `treats`:

```
fn eat(treat: Treat) -> ConsumedTreat
```

И наконец, в корне пакета будет представлена функция `regret`:

```
fn regret(treat: ConsumedTreat)
```

В сигнатурах всех этих функций используются типы из модуля `treats` пакета `libsnack`. Все пути к этим типам могут быть выражены с помощью относительных либо абсолютных путей. Функции будут написаны обоими способами, чтобы увидеть, как меняется исходный код при использовании каждого типа путей. Начнем с абсолютных путей.

Листинг 5.13 Методы жизненного цикла в пакете libsnack с использованием абсолютных путей

```
pub mod treats {
    pub mod shop {
        fn buy() -> crate::treats::Treat {
            crate::treats::Treat::IceCream
        }
    }

    pub enum Treat {
        Candy,
        IceCream,
    }

    pub struct ConsumedTreat {
        treat: Treat,
    }

    fn eat(treat: crate::treats::Treat) -> crate::treats::ConsumedTreat
    {
        crate::treats::ConsumedTreat { treat }
    }
}

fn regret(treat: crate::treats::ConsumedTreat) {
    println!("That was a mistake");
}
```

Как вы видите, этот исходный код очень быстро становится многословным. Особенно трудно читать сигнатуру функции `treats::eat`, потому что ей требуется два больших пути в одной строке. Давайте попробуем использовать только относительные пути.

Листинг 5.14 Методы жизненного цикла в пакете `libsнак` с использованием относительных путей

```
pub mod treats {
    pub mod shop {
        fn buy() -> super::Treat {
            super::Treat::IceCream
        }
    }

    pub enum Treat {
        Candy,
        IceCream,
    }

    pub struct ConsumedTreat {
        treat: Treat,
    }

    fn eat(treat: Treat) -> ConsumedTreat {
        ConsumedTreat { treat }
    }
}

fn regret(treat: treats::ConsumedTreat) {
    println!("That was a mistake");
}
```

Теперь этот исходный код читается немного легче. Функция `eat` больше не нуждается в какой-либо квалификации модуля, так как она определена в том же модуле, что и используемые ею типы `Treat` и `ConsumedTreat`. Недостатком относительных путей является то, что при перемещении функции, сигнатура которой имеет относительный тип, необходимо переписывать типы относительно нового местоположения. Например, если переместить функцию `regret` в модуль `shop`, то потребуется изменить сигнатуру на

```
fn regret(treat: super::ConsumedTreat)
```

Не проблема, когда имеется всего несколько функций и типов, но эти изменения будут накапливаться и вызывать разочарование. По этой причине часто бывает полезно сочетать использование абсолютных и относительных путей в исходном коде Rust с помощью инструкции `use`, о которой упоминалось ранее. Давайте посмотрим, как переписать этот пакет с использованием `use`.

Листинг 5.15 Использование как относительных, так и абсолютных путей

```
pub mod treats {
    pub mod shop {
        use crate::treats::Treat;

        fn buy() -> Treat {
            Treat::IceCream
        }
    }

    pub enum Treat {
        Candy,
        IceCream,
    }

    pub struct ConsumedTreat {
        treat: Treat,
    }

    fn eat(treat: Treat) -> ConsumedTreat {
        ConsumedTreat { treat }
    }
}

use crate::treats::ConsumedTreat;

fn regret(treat: ConsumedTreat) {
    println!("That was a mistake");
}
```

На рис. 5.2 показаны все относительные и абсолютные пути, которые используются в листинге 5.15.

Обратите внимание, что стрелки, обозначающие абсолютные пути, проходят до самого верха пакета. Это сделано намеренно и служит для напоминания о том, что абсолютные пути всегда начинаются в корне пакета и ведут от того места, где мы находимся, обратно вверх к корню.

Если пишутся инструкции `use`, которые опираются на абсолютные пути, то остальная часть исходного кода может опираться на относительные пути, и вообще не нужно беспокоиться об иерархии модулей. За счет этого внимание концентрируется на иерархии модулей в инструкциях `use`, упрощая перемещение и чтение остальной части исходного кода.

Теперь давайте вернемся к программе приветствия пользователей и скомпилируем ее. Вспомните, что написанный исходный код состоял из следующих четырех листингов, которые не компилировались.

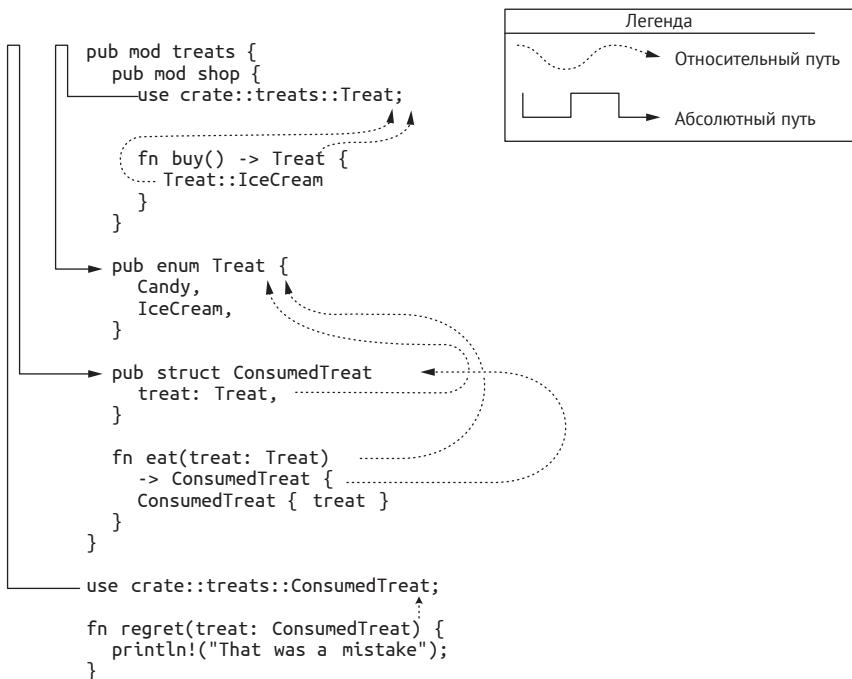


Рисунок 5.2 Относительные и абсолютные пути, используемые в листинге 5.15

Листинг 5.16 main.rs

```

use input::get_name;
use output::{goodbye, hello};

mod day_kind;
mod input;
mod output;

fn main() {
    let name = get_name();

    hello(&name);
    goodbye(&name);
}

```

Листинг 5.17 day_kind.rs

```

pub enum DayKind {
    Good,
    Bad,
}

```

Листинг 5.18 input.rs

```
use std::io::stdin;

pub fn get_name() -> String {
    let mut name = String::new();

    println!("Please enter your name");
    stdin().read_line(&mut name).unwrap();

    name
}
```

Листинг 5.19 output.rs

```
use day_kind::DayKind; ← Причиной ошибки компилятора является
                        неурегилированный импорт 'day_kind'.
pub fn print_day_kind_message(day_kind: DayKind) {
    match day_kind {
        DayKind::Good => println!("I'm glad to hear you're having a good
day!"),
        DayKind::Bad => println!("I'm sorry to hear you're having a bad
day"),
    }
}

pub fn goodbye(name: &str) {
    println!("Goodbye, {}", name);
}

pub fn hello(name: &str) {
    println!("Hello, {}", name);
}
```

Располагая необходимыми знаниями о путях, теперь удастся исправить ошибку. Имя `day_kind` в модуле `output` не существует, поэтому для того чтобы до него добраться, нельзя использовать относительный путь. Можно применить особый отрезок пути под названием `super`, который позволяет продвигаться вверх по иерархии модулей, аналогично синтаксису `..` в файловой системе. Однако использовать `super` обычно не рекомендуется, за исключением очень простых случаев. Для исправления ошибки следует применять абсолютный путь. Поскольку модуль `day_kind` находится непосредственно под корнем пакета, абсолютным путем к нему будет `crate::day_kind`. Это означает, что исходный код можно исправить, видоизменив инструкцию `use` на

```
use crate::day_kind::DayKind
```

Теперь исходный код должен компилироваться. Разобравшись с этим, теперь можно завершить обновление программы приветствия, разрешив ей спрашивать пользователя о том, как прошел его

день. Давайте напишем новую функцию в модуле `input.rs`, которая выполняет именно это.

Листинг 5.20 Опрашивание пользователя о том, как прошел его день

```
use crate::day_kind::DayKind;

pub fn how_was_day() -> DayKind {
    let mut day = String::new();

    println!("How was your day?");
    stdin().read_line(&mut day).unwrap();

    let day_trimmed = day.trim(); <--
```

`if` `day_trimmed == "good"` {
 DayKind::Good
} `else` {
 DayKind::Bad
}
}

Функция `read_line` генерирует строковый литерал, содержащий символ новой строки в конце. Вызов метода `.trim` удаляет начальные и концевые пробелы, что необходимо для сравнения этого строкового литерала с "good". Если бы метод `.trim` не был вызван, то пришлось бы писать `if day == "good\n"`.

Теперь, когда есть способ получать от пользователя ответ на вопрос о проведенном им дне и способ печатать сообщение, давайте совместим их в функции `main`.

Листинг 5.21 Вызов функций о проведенном дне из функции main

```
use input::{get_name, how_was_day};
use output::{goodbye, hello, print_day_kind_message};

mod day_kind;
mod input;
mod output;

fn main() {
    let name = get_name();

    hello(&name);

    let day_kind = how_was_day(); <--
```

`print_day_kind_message(day_kind);`

`goodbye(&name);`

}

Импорт типа `DayKind`, чтобы сохранить `DayKind` в переменной, не требуется. Rust нуждается в импорте структур и перечислений только в том случае, если они используются по имени. Если бы нужна была явная аннотация типа, например `let day_kind: DayKind`, то ее нужно было бы импортировать.

И теперь можно попробовать исполнить программу как для хороших, так и для плохих дней:

```
$ cargo run
Please enter your name
Rose
Hello, Rose
```

```
How was your day?  
good  
I'm glad to hear you're having a good day!  
Goodbye, Rose
```

```
$ cargo run  
Please enter your name  
Jack  
Hello, Jack
```

```
How was your day?  
bad  
I'm sorry to hear you're having a bad day  
Goodbye, Jack
```

Итак, теперь можно спросить пользователя о том, как его зовут и как прошел его день, и ответить соответствующим образом. Осталось решить две небольшие проблемы:

- после текста "Hello, {name}" идет символ новой строки, потому что метод `.trim()` не вызывается в строковом литерале `name`. Можно создать единую функцию для извлечения строки текста из `stdin` и обрезки пробелов;
- кажется излишним везде ссылаться `crate::day_kind::DayKind`, так как имя типа совпадает с именем модуля. Можно создать псевдоним, который упростит его использование.

Давайте начнем с первой проблемы. Учитывая то, что вы увидели в других функциях, которые читают данные из `stdin` в модуле `input`, можно было бы придумать что-то похожее на вот это:

```
fn read_line() -> String {  
    let mut line = String::new();  
  
    stdin().read_line(&mut line).unwrap();  
  
    line.trim()  
}
```

Но приведенный выше исходный код не компилируется, и компилятор Rust спешит объяснить причину:

```
$ cargo build  
error[E0308]: mismatched types  
--> src/input.rs:9:3  
|  
4 | fn read_line() -> String {  
| |----- expected `String` because of return type  
|  
9 |     line.trim()  
| ^^^^^^^^^^  
| |  
| |----- expected struct `String`, found `&str`  
| help: try using a conversion method: `line.trim().to_string()`
```

Метод `String::trim` не возвращает еще одно значение типа `String` с собственным пространством памяти; вместо этого он возвращает строковый срез `&str`, который ссылается на ту же опорную память, что и у изначального значения типа `String`. В большинстве случаев это хорошо, потому что означает, что не нужно перераспределять строковые литералы в памяти, если требуется удалить только пробелы. В данном случае нужно перераспределить. Это делается, следуя инструкции компилятора путем добавления метода `.to_string()` в конец строки кода, чтобы перераспределить `&str` в `String`.

Теперь нужно переписать функции `get_name` и `how_was_day`, чтобы использовать новую вспомогательную функцию `read_line`.

Листинг 5.22 Модуль `input` программы приветствия пользователей с добавленной вспомогательной функцией `read_line`

```
use crate::day_kind::DayKind;
use std::io::stdin;

fn read_line() -> String { ←
    let mut line = String::new();

    stdin().read_line(&mut line).unwrap();

    line.trim().to_string()
}

pub fn get_name() -> String {
    println!("Please enter your name");
    read_line()
}

pub fn how_was_day() -> DayKind {
    println!("How was your day?");
    let day = read_line();

    if day == "good" {
        DayKind::Good
    } else {
        DayKind::Bad
    }
}
```

Эта функция не помечена как `pub`.
Она бесполезна вне контекста модуля `input`, поэтому ее не нужно экспорттировать в другие модули пакета.

Теперь исходный код выполняется без каких-либо разрывов в выходных данных после имен:

```
$ cargo run
Please enter your name
Lonnie
Hello, Lonnie
How was your day?
good
I'm glad to hear you're having a good day!
Goodbye, Lonnie
```

Устранив разрывы и централизовав доступ к `stdin`, теперь давайте создадим псевдоним для типа `DayKind`, чтобы упростить его импорт.

5.2.2 Псевдонимы путей

Для того чтобы создать псевдоним для `DayKind`, надо скомбинировать два ключевых слова, которые ранее использовались много раз, – `pub use`. Их комбинация называется *реэкспортом* и действует как псевдоним для импортируемого элемента. Давайте посмотрим, как это работает на практике; добавьте следующую ниже строку в начало файла `main.rs`:

```
pub use crate::day_kind::DayKind;
```

Этот исходный код импортирует `DayKind` из модуля `day_kind` и создает новое публичное имя `DayKind`, которое находится в корне пакета. Затем его можно использовать в модулях ввода и вывода:

```
use crate::DayKind;           ← Новый способ написания инструкции import.  
use crate::day_kind::DayKind; ← Старый способ написания инструкции import.
```

Обе инструкции `use` относятся к одному и тому же элементу, но одна из них короче и основана на инструкции `pub use`, которая была добавлена в файл `main.rs` ранее.

Полное содержимое пакета приветствия пользователей представлено в следующих ниже четырех листингах.

Листинг 5.23 Завершенное приложение приветства пользователей: main.rs

```
use input::{get_name, how_was_day};  
use output::{goodbye, hello, print_day_kind_message};  
  
pub use day_kind::DayKind;  
  
mod day_kind;  
mod input;  
mod output;  
  
fn main() {  
    let name = get_name();  
  
    hello(&name);  
  
    let day_kind = how_was_day();  
    print_day_kind_message(day_kind);  
  
    goodbye(&name);  
}
```

Листинг 5.24 Завершенное приложение приветствия пользователей: input.rs

```
use crate::DayKind;
use std::io::stdin;

fn read_line() -> String {
    let mut line = String::new();

    stdin().read_line(&mut line).unwrap();

    line.trim().to_string()
}

pub fn get_name() -> String {
    println!("Please enter your name");
    read_line()
}

pub fn how_was_day() -> DayKind {
    println!("How was your day?");
    let day = read_line();

    if day == "good" {
        DayKind::Good
    } else {
        DayKind::Bad
    }
}
```

Листинг 5.25 Завершенное приложение приветствия пользователей: output.rs

```
use crate::DayKind;

pub fn print_day_kind_message(day_kind: DayKind) {
    match day_kind {
        DayKind::Good => println!("I'm glad to hear you're having a good day!"),
        DayKind::Bad => println!("I'm sorry to hear you're having a bad day"),
    }
}

pub fn goodbye(name: &str) {
    println!("Goodbye, {}", name);
}

pub fn hello(name: &str) {
    println!("Hello, {}", name);
}
```

Листинг 5.26 Завершенное приложение приветствия пользователей: day_kind.rs

```
pub enum DayKind {
    Good,
    Bad,
}
```

Инструкции `pub use` часто добавляются в исходный код Rust, чтобы скрывать иерархию модулей от публичного API. Это позволяет создавать глубоко вложенные и специфичные модули внутри пакета, не требуя от конечных пользователей заботы о них. Представьте себе, что вы используете пакет Rust под названием `forest`, который имеет следующий ниже библиотечный файл `lib.rs`:

```
pub mod the {
    pub mod secret {
        pub mod entrance {
            pub mod to {
                pub mod the {
                    pub mod forest {
                        pub fn enter() { }
                    }
                }
            }
        }
    }
}

pub use the::secret::entrance::to::the::forest::enter;
```

К функции `enter` можно было бы построить очень большой путь самим либо вызвать `forest::enter`. Что бы вы предпочли сделать? Будучи специалистом по техническому сопровождению библиотек, хотите ли вы придерживаться этого очень длинного пути как составной части публичного API? Если вы измените какую-либо часть этого пути, то у пользователей, применяющих длинную версию пути, возникнут ошибки компиляции.

Осталось обсудить еще несколько вопросов, касающихся путей и модулей. Давайте рассмотрим значительно упрощенную версию пакета `forest`. Этот пакет содержит большое количество модулей, представляющих разнообразные области леса, каждый из которых содержит функцию `enter`, используемую для прохода в эту область леса. В реализациях всех этих функций `enter` используется коллективная функция `forest::enter_area`.

Листинг 5.27 Пакет `forest`

```
pub mod forest {
    pub fn enter_area(area: &str) {
        match area {
```

```

        "tree cover" => println!("It's getting darker..."),
        "witches coven" => println!("It's getting spookier..."),
        "walking path" => println!("It's getting easier to walk..."),
        x => panic!("Unexpected area: {}", x),
    }
}
}

pub mod tree_cover {
    pub fn enter() {
        crate::forest::enter_area("tree cover");
    }
}

pub mod walking_path {
    pub fn enter() {
        crate::forest::enter_area("walking path");
    }
}

pub mod witches_coven {
    pub fn enter() {
        crate::forest::enter_area("witches coven");
    }
}
}

```

Пользователи пакета `forest` должны иметь возможность вызывать функции `tree_cover::enter`, `walking_path::enter` и `witches_coven::enter`. Они не должны вызывать обобщенную функцию `forest::enter_area`, так как она предназначена только для работы со строковыми литералами, которые поступают из других функций в этом пакете. Текущий пакет `forest` не защищает пользователей от неправильного использования этого API. Пакет `forest` и его функция `enter_area` выставлены наружу в публичное пространство и могут использоваться пользователями пакета напрямую. Эти элементы не должны выставляться в публичное пространство, их необходимо скрыть. Давайте удалим ключевое слово `pub` из модуля `forest` и функции `enter_area`.

Листинг 5.28 Модуль forest с удаленным ключевым словом pub

```

mod forest {
    fn enter_area(area: &str) {
        match area {
            "tree cover" => println!("It's getting darker..."),
            "witches coven" => println!("It's getting spookier..."),
            "walking path" => println!("It's getting easier to
walk..."),
            x => panic!("Unexpected area: {}", x),
        }
    }
}

...

```

Если сейчас попытаться скомпилировать этот исходный код, то возникнет небольшая трудность:

```
$ cargo build
error[E0603]: function `enter_area` is private
  --> src/lib.rs:14:20
   |
14 |     crate::forest::enter_area("tree cover");
   |             ^^^^^^^^^^ private function
   |
note: the function `enter_area` is defined here
  --> src/lib.rs:2:3
   |
2 | fn enter_area(area: &str) {
   | ^^^^^^^^^^^^^^^^^^^^^^^^^^
... (same error on lines 20 and 26)
```

Компилятор жалуется, потому что функция `enter_area` была сделана приватной, что является отличительным признаком уровня модуля, а не уровня пакета. Теперь функцию `enter_area` можно вызывать только из еще одной функции внутри модуля `forest`. Нежелательно добавлять `pub` в функцию `enter_area`, так как нежелательно, чтобы она была доступна за пределами пакета, но также нежелательно, чтобы она была скрыта от других модулей *внутри* пакета. Здесь можно выполнить оба требования, используя другую разновидность модификатора видимости – `pub(crate)`.

Как следует из синтаксиса, `pub(crate)` означает, что элемент виден всем другим модулям внутри пакета, но не виден ни из одного другого пакета. Это полезно при написании служебных функций, которые используются во всем пакете и которые нежелательно выставлять наружу в публичное пространство. Она точно описывает функцию `enter_area` в модуле `forest`. Давайте добавим эту аннотацию.

Листинг 5.29 Модуль с функцией, видимой внутри пакета

```
mod forest {
    pub(crate) fn enter_area(area: &str) {
        match area {
            "tree cover" => println!("It's getting darker..."),
            "witches coven" => println!("It's getting spookier..."),
            "walking path" => println!("It's getting easier to
walk..."),
            x => panic!("Unexpected area: {}", x),
        }
    }
}

...
```

Теперь пакет компилируется без проблем:

```
$ cargo build
Compiling forest
Finished dev [unoptimized + debuginfo] target(s) in 0.13s
```

Но минуточку! Почему он компилируется? Модуль `forest` не помечен как `pub(crate)`. Почему его можно использовать из других модулей? Для того чтобы ответить на этот вопрос, нужно взглянуть на правила восходящей видимости, которые относятся к модулям.

5.3. Восходящая видимость

Исходный код внутри модуля наследует правила видимости от модуля, расположенного над ним. Возможно, понять эту концепцию будет сложновато, поэтому давайте рассмотрим краткий пример.

Листинг 5.30. Восходящая видимость без ключевого слова pub

```
fn function() {}

mod nested {
    fn function() {
        crate::function();
    }
}

mod very_nested {
    fn function() {
        crate::function();
        crate::nested::function();
        crate::nested::very_nested::function();
    }
}

mod very_very_nested {
    fn function() {
        crate::function();
        crate::nested::function();
        crate::nested::very_nested::function();
    }
}
```

Обратите внимание, что ни одна функция или модуль не помечена как `pub`. Все функции являются приватными, но это работает, потому что функция пытается вызывать только те функции, которые находятся выше в дереве модулей, чем они сами. Можно сделать так, чтобы исходный код не компилировался, изменив его так, чтобы функции вызывались вниз по дереву модулей.

Листинг 5.31 Нисходящая видимость без ключевого слова pub (не работает)

```
fn function() {
    nested::function();
}

mod nested {
```

```
fn function() {
    very_nested::function();
}

mod very_nested {
    fn function() {
        very_very_nested::function();
    }
}

mod very_very_nested {
    fn function() {}
}
}
```

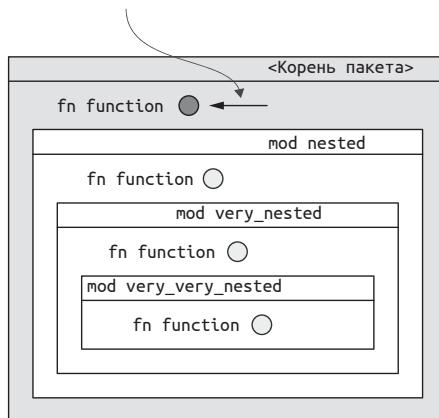
Теперь каждая строка, которая пытается вызвать вниз по дереву модулей, приводит к ошибке компиляции:

```
$ cargo build
error[E0603]: function `function` is private
--> src/lib.rs:2:11
|
2 |     nested::function();
|         ^^^^^^^^^ private function
|
error[E0603]: function `function` is private
--> src/lib.rs:7:18
|
7 |     very_nested::function();
|         ^^^^^^^^^ private function
|
error[E0603]: function `function` is private
--> src/lib.rs:12:25
|
12 |         very_very_nested::function();
|             ^^^^^^^^^ private function
|
```

На рис. 5.3 показаны функции в каждой точке дерева модулей, которые разрешено вызывать.

Таким образом, приведенный в листинге 5.29 исходный код работает вследствие используемых в языке Rust неявных правил видимости для членов родительского модуля. Вот окончательный исходный код пакета `forest`.

Поскольку все функции и модули являются приватными, вызывать какие-либо другие функции из функции в корне пакета нельзя.



Эта стрелка на визуализации показывает функции, которые могут вызываться из `crate::nested::function`. Отсюда можно вызывать только `crate::function`.

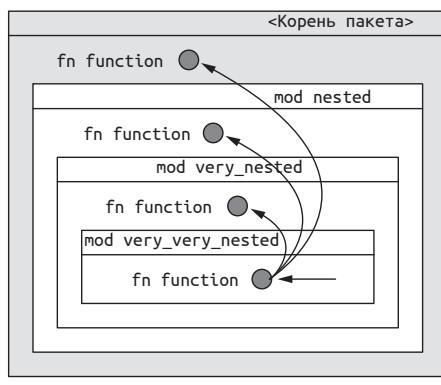
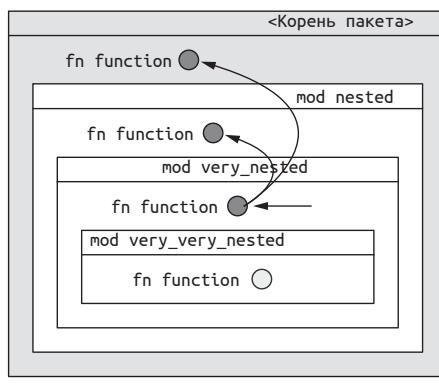
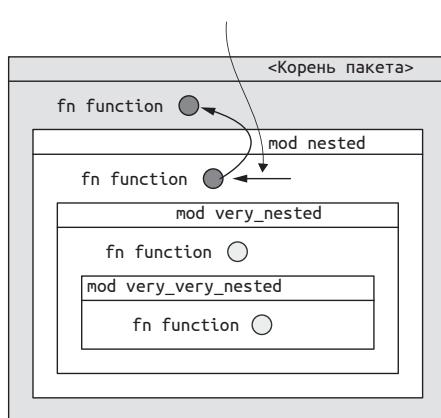


Рисунок 5.3 Визуализация правила видимости родителей: модули могут использовать приватные элементы из родительских модулей

Листинг 5.32 Окончательный исходный код пакета forest

```
mod forest {
    pub(crate) fn enter_area(area: &str) {
        match area {
            "tree cover" => println!("It's getting darker..."),
            "witches coven" => println!("It's getting spookier..."),
            "walking path" => println!("It's getting easier to
walk..."),
            x => panic!("Unexpected area: {}", x),
        }
    }
}
```

```
pub mod tree_cover {  
    pub fn enter() {  
        crate::forest::enter_area("tree cover");  
    }  
}  
  
pub mod walking_path {  
    pub fn enter() {  
        crate::forest::enter_area("walking path");  
    }  
}  
  
pub mod witches_coven {  
    pub fn enter() {  
        crate::forest::enter_area("witches coven");  
    }  
}
```

Теперь вы гораздо лучше понимаете систему модулей в языке Rust, что очень пригодится при создании более крупных программ и библиотек. Возможность легко подразделять исходный код и скрывать тот исходный код, который не должен быть частью публичного интерфейса, имеет решающее значение для разработки программного обеспечения, которое легко понять и сопровождать. В следующей главе будет рассмотрен вопрос о том, как ускорять исходный код на языке Python, используя язык Rust и пакет [pyO3](#).

Краткий итог

- Использование ключевого слова `mod` позволяет делить исходный код на логические модули под конкретные цели.
- Написание `mod your_mod_name { contents; }` позволяет хранить модули в одном файле.
- Написание `mod your_mod_name;` позволяет писать содержимое модуля в файле `your_mod_name.rs`.
- Ключевое слово `pub` необходимо для того, чтобы делать элементы публичными, если их планируется использовать между модулями.
- Модули могут быть вложены настолько глубоко, насколько потребуется.
- Для доступа к элементам внутри модулей используются относительные и абсолютные пути.
- Относительные пути вычисляются относительно текущего модуля.
- Абсолютные пути начинаются с имени пакета.
- Ключевое слово `crate` указывает на корень текущего пакета.
- Комбинация ключевых слов `pub use` позволяет создавать псевдонимы для элементов.
- Модули наследуют видимость от своих родительских модулей.
- Модификатор видимости `pub(crate)` используется для того, чтобы помечать элементы как публичные внутри пакета, но приватные для других пакетов.



Интеграция с динамическими языками

Эта глава охватывает следующие ниже темы:

- написание исходного кода на языке Rust, который можно легко вызывать из Python;
- вызов исходного кода Python из Rust;
- измерение и тестирование производительности исходного кода на языке Rust с помощью библиотеки Criterion.

До сих пор много времени уделялось основам языка Rust и межъязыковому интерфейсу к внешним функциям С (C FFI). В этой главе более непосредственно будет рассмотрен вопрос о том, как интегрировать исходный код на языке Rust в языки динамического программирования и получать от этого огромные преимущества в производительности.

6.1 Обработка данных в Python

Представьте себе, что вы работаете над программой на языке Python, которая агрегирует данные в формате JSON, разделенные символом новой строки. Вот файл входных данных; назовем его `data.jsonl`:

```
{
  "name": "Stokes Baker", "value": 954832 }
{
  "name": "Joseph Solomon", "value": 279836 }
{
  "name": "Gonzalez Koch", "value": 140431 }
{
  "name": "Parrish Waters", "value": 490411 }
{
  "name": "Sharlene Nunez", "value": 889667 }
{
  "name": "Meadows David", "value": 892040 }
{
  "name": "Whitley Mendoza", "value": 965462 }
{
  "name": "Santiago Hood", "value": 280041 }
{
  "name": "Carver Caldwell", "value": 632926 }
{
  "name": "Tara Patterson", "value": 678175 }
```

Программа вычисляет общую сумму всех записей `"value"` и сумму длин всех строковых литералов `"name"`. В обычном исходном коде на языке Python эта процедура относительно проста. Давайте сохраним ее в файле с именем `main.py`.

Листинг 6.1 Программа на языке Python для агрегирования строк в формате JSON

```
import sys
import json

s=0

for line in sys.stdin:
    value = json.loads(line)
    s += value['value']
    s += len(value['name'])

print(s)
```

Теперь выполним ее и посмотрим, что получится:

```
$ python main.py < data.jsonl
6203958
```

Исходный код агрегации работает, но вы слышали жалобы на то, что он недостаточно отвечает потребностям пользователей. Люди возлагают очень большие надежды на производительность этого функционального компонента. Следовательно, вы решили попробовать перенести часть функциональности по синтаксико-структурному разбору данных JSON на язык Rust, оставив ввод-вывод на языке Python, поскольку это часть более крупного приложения на Python. Давайте рассмотрим план выполнения этого перемещения.

6.2 Планирование перемещения на язык Rust

Поскольку переписывается функциональность агрегации данных, хранящихся в формате JSON, будет сделано несколько вещей:

- реализована функциональность агрегации в версии на чистом Rust;
- использована библиотека PyO3 для обертывания исходного кода Rust в формат, который может вызываться из Python;

- создан инструментарий измерения и тестирования производительности для сравнения изначального исходного кода на чистом Python с чистым Rust и с Rust, интегрированным в Python.

Давайте начнем с написания функциональности на языке Rust. Сначала нужно определить, какую часть исходного кода требуется переписать. Желательно оставить часть кода ввода-вывода на Python, поскольку предполагается, что исходный код агрегации данных JSON является частью более крупной программы на Python, такой как HTTP-сервер. Исходный код на Python также отвечает за суммирование каждого вызова исходного кода на Rust. Python'овский исходный код выглядит примерно так, как показано в следующем ниже списке.

Листинг 6.2 Исходный код на языке Python

```
import sys
import rust_json

s=0

for line in sys.stdin:
    s += rust_json.sum(line)

print(s)
```

Функция на Rust должна выполнять то, что было удалено из исходного кода на Python:

- 1 принимать на входе строковый литерал;
- 2 выполнять разбор этого строкового литерала как объект JSON, содержащий строковое свойство "name" и числовое свойство "value";
- 3 возвращать сумму свойства "value" и длину свойства "name".

Реализацию задачи можно набросать в псевдокоде Rust.

Листинг 6.3 Псевдокод Rust для суммирования данных JSON

```
pub fn sum(line: &str) -> i32 {
    let data = parse_as_json(line);
    data.value + data.name.len()
}
```

Как будет выглядеть на языке Rust этот код разбора данных JSON, пока непонятно, но он будет рассмотрен в следующем далее разделе.

Работа с исходным кодом на языке Rust почти закончена, но нужно сделать небольшой крюк, чтобы посмотреть, как выполнять разбор данных JSON на языке Rust.

6.3 Разбор данных JSON

Многие форматы данных в Rust легко поддаются разбору в структуры данных Rust с помощью Serde. Serde – это «фреймворк для эффективной и обобщенной сериализации и десериализации структур данных Rust (<https://serde.rs>)». Название Serde происходит от первых частей

слов serialize (сериализовать) и deserialize (десериализовать). Фреймворк Serde действует как обобщенный фреймворк, которого не заботит какой-либо конкретный формат данных, а другие пакеты, такие как `serde_json`, служат мостом между обобщенной моделью данных Serde и форматом данных JSON. В экосистеме фреймворка Serde имеется огромное количество пакетов для самых разных форматов. На официальном веб-сайте перечислено более 20 разных форматов данных, из которых с помощью фреймворка Serde можно сериализовывать и/или десериализовывать в типы данных языка Rust. На рис. 6.1 показано то, как различные части указанной экосистемы сочетаются друг с другом.

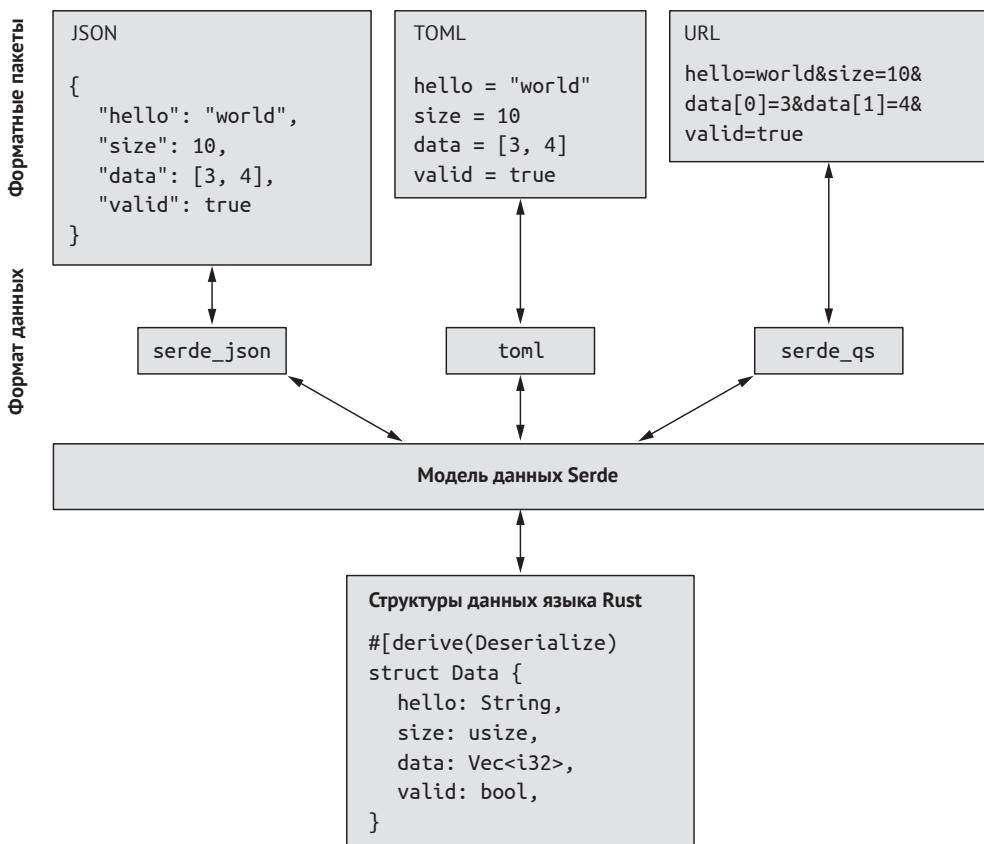


Рисунок 6.1 Экосистема фреймворка Serde

В основе фреймворка Serde лежат две общие черты. Общая черта `Serialize` используется для приема типа данных Rust и его переложения в некий формат данных. И наоборот, общая черта `Deserialize` используется для разбора формата данных в тип данных Rust. Исходный код с реализацией этих общих черт можно написать вручную, но также можно использовать компилятор Rust, который выполнит эту работу за вас. Давайте посмотрим, как это сделать. Вспомните, что стоит задача выполнить разбор объектов JSON, которые выглядят следующим образом:

```
{ "name": "Rachelle Ferguson", "value": 948129 }
```

Поле `name` содержит строковый литерал, а поле `value` – число. Если нужно было бы создать структуру Rust для хранения этих данных, то она могла бы выглядеть следующим образом:

```
struct Data {  
    name: String,  
    value: i32,  
}
```

Давайте приступим к разработке исходного кода разбора в эту структуру. Создайте новый проект Rust с именем `rust_json`:

```
$ cargo new rust_json
```

Перед тем как перейти к исходному коду, в файл `Cargo.toml` нужно добавить несколько зависимостей. Также нужно использовать какой-то новый синтаксис TOML, который раньше не встречался. Добавьте следующие ниже строки в секцию `[dependencies]` файла `Cargo.toml`:

```
[dependencies]  
serde_json = "1.0" ← Несмотря на то что serde_json зависит от serde,  
                     его можно указать первым. Cargo не следит за по-  
                     рядком расположения зависимостей.  
serde = { version = "1.0", features = ["derive"] }
```

Строка `serde_json` выглядит достаточно знакомой, но строка зависимостей для `serde` – немного странная. Как и в случае с форматом JSON, TOML может содержать объекты с произвольными ключами и значениями. Cargo принимает зависимости либо в виде имени в соотнесении с литералом версии, либо в виде имени в соотнесении с конфигурационным объектом, который имеет больше опций. Для получения полной информации о ключах, которые можно указывать, посетите раздел зависимостей в книге по Cargo (<https://mng.bz/N1qX>).

В данном случае указывается номер версии фреймворка `serde` и набор функциональных компонентов. *Функциональные компоненты* (англ. `features`) – это механизм, который Rust использует для условной компиляции. В пакетах можно задавать любое количество функциональных компонентов, которые могут активировать разные пути исполнения кода, включать дополнительные зависимости либо действовать функциональные компоненты в своих собственных зависимостях. В данном конкретном случае подлежит активации функциональный компонент `derive`, который содержит исходный код, позволяющий компилятору Rust генерировать исходный код синтаксико-структурного разбора за нас. Разработчик не только избавлен от необходимости вводить много текста, но и генерирует исходный код разбора, специфичный для любого предоставляемого типа данных.

Разобравшись с зависимостями, теперь давайте перейдем к исходному коду. Откройте файл `main.rs` и добавьте следующий ниже исходный код.

Листинг 6.4 Первая попытка написать исходный код разбора данных JSON

```
struct Data {
    name: String,
    value: i32,
}

fn main() {
    let input = "{ \"name\": \"Sharpe Oliver\", \"value\": 134087 }";

    let parsed = serde_json::from_str(input).unwrap();

    println!("{:?}", parsed);
}
```

Программа должна попытаться выполнить разбор предоставленного строкового литерала JSON и распечатать результирующий тип данных Rust. Давайте попробуем ее выполнить:

```
$ cargo run
error[E0282]: type annotations needed
--> src/main.rs:9:7
 |
9 | let parsed = serde_json::from_str(input).unwrap();
|     ^^^^^^ consider giving `parsed` a type
```

Пока что возникает ошибка ввиду того, что компилятор недостаточно умен, чтобы заключить, что из функции `serde_json::from_str` ожидается экземпляр типа `Data`. Указанная функция имеет обобщенный возвращаемый тип, аналогичный функции `parse`, о которой вы узнали в главе 3. Как и в случае с функцией `parse`, компилятору нужна подсказка о том, какой тип он должен вернуть. Добавьте явную аннотацию типа к переменной `parsed`:

```
let parsed: Data = serde_json::from_str(input).unwrap();
```

Давайте попробуем выполнить программу еще раз:

```
$ cargo run
error[E0277]: the trait bound `Data: serde::de::Deserialize<'_>` is not
              satisfied
--> src/main.rs:9:22
 |
9 |     let parsed: Data = serde_json::from_str(input).unwrap();
|     ^^^^^^^^^^^^^^^^^^^^^^ the trait
|             `serde::de::Deserialize<'_>` is not
|             implemented for `Data`
|
::: serde_json-1.0.68/src/de.rs:2587:8
|
2587 |     T: de::Deserialize<'a>,
```

```

|----- required by this bound in
|                               `serde_json::from_str`  

error[E0277]: `Data` doesn't implement `Debug`
--> src/main.rs:11:20
|
11 |     println!("{}:", parsed);
|           ^^^^^ `Data` cannot be formatted using `{:?}`  

|

```

Теперь получено два разных сообщения об ошибках. Сразу распознается ошибка, возникающая, когда в `Data` не реализована общая черта `Debug`. Если требуется распечатывать значения Rust с помощью форматировщика `{:?}", то необходимо обеспечить реализацию общей черты Debug. Другая ошибка исходит из serde_json: в Data не реализована общая черта Deserialize. Как и в случае с общей чертой Debug, если требуется выполнять десериализацию в структуру, то необходимо реализовать общую черту Deserialize. Благодаря функциональному компоненту derive, который был включен в зависимость serde, обе ошибки можно устранить с помощью одной строки.`

Листинг 6.5 Рабочий исходный код разбора данных JSON

```

#[derive(Debug, serde::Deserialize)] ←
struct Data {                         здесь обратите внимание
    name: String,                      на новую строку derive.
    value: i32,
}

fn main() {
    let input = "{ \"name\": \"Sharpe Oliver\", \"value\": 134087 }";

    let parsed: Data = serde_json::from_str(input).unwrap();

    println!("{}:", parsed);
}

```

Давайте прямо сейчас попробуем выполнить исходный код:

```
$ cargo run
Data { name: "Sharpe Oliver", value: 134087 }
```

Работает! Для того чтобы выполнить разбор из любого поддерживаемого фреймворком `serde` формата данных в большинство простых типов данных, достаточно добавить атрибут `#[derive(serde::Deserialize)]`. Обратите внимание, что определение структуры не содержит исходного кода, специфичного для JSON. Если бы были добавлены правильные зависимости, то в структуру `Data` можно было бы так же легко выполнить разбор из YAML, TOML, MessagePack или даже переменных среды. Подобного рода простые типы данных обычно используются авторами библиотек, для того

чтобы реализовывать десериализацию и/или сериализацию, а потом уже пользователи библиотек могут сериализовывать и/или десериализовывать эти типы в любые форматы, которые они хотят.

Во фреймворке Serde есть много более сложных опций конфигурации, которые служат для переименования полей, предоставления дефолтных значений или даже для организации вложенного поведения. Все они хорошо задокументированы по адресу <https://serde.rs>, но здесь они рассматриваться не будут.

Фреймворк Serde также предоставляет возможность проверки типов. Давайте попробуем изменить тип поля `name` на `i32`.

Листинг 6.6 Исходный код разбора данных JSON с ошибкой типа во время исполнения

```
#[derive(Debug, serde::Deserialize)]
struct Data {
    name: i32,           ← Ожидаемый тип поля
    value: i32,           | name теперь i32.
}
fn main() {
    let input = "{ \"name\": \"Sharpe Oliver\", \"value\": 134087 }"; ← Предоставляется
    let parsed: Data = serde_json::from_str(input).unwrap();             | строковое значение для name.

    println!("{}: {}", parsed);
}
```

Теперь давайте выполним исходный код, чтобы посмотреть, что произойдет:

```
$ cargo run
thread 'main' panicked at called `Result::unwrap()` on an `Err` value:
Error("invalid type: string \"Sharpe Oliver\", expected i32", line: 1,
      column: 19)
```

Поскольку на объекте `Result`, возвращаемом из функции `serde_json::from_str`, используется метод `unwrap`, программа поднимает панику, когда функция возвращает ошибку. Но вы видите, что эта ошибка включает в себя информацию о строке и столбце, а также произошедшую точную ошибку типа. Эти детали представляют работу, которую разработчики не выполняют сами при генерировании сообщений об ошибках и валидации; при использовании фреймворка `serde` эта рабочая, по сути, выполняется бесплатно.

Разобравшись в том, как выполнять разбор данных JSON в простые структуры на языке Rust, давайте воссоздадим остальную Python'овскую функциональность. Напомним, что нужно просуммировать значение свойства `value` и длину свойства `name`. Давайте создадим функцию, которая выполняет разбор данных JSON и возвращает значение математического выражения.

Листинг 6.7 Программа на языке Rust, имитирующая функциональность на языке Python

```
#[derive(Debug, serde::Deserialize)]
struct Data {
    name: String,
    value: i32,
}

fn main() {
    let result =
        sum("{\"name\": \"Rachelle Ferguson\", \"value\": 948129 }");

    println!("{}", result);
}

fn sum(input: &str) -> i32 {
    let parsed: Data = serde_json::from_str(input).unwrap();

    parsed.name.len() as i32 + parsed.value
}
```

`String::len()` возвращает тип `usize`, который должен быть приведен к типу `i32` вручную.

Этот исходный код можно выполнить прямо сейчас и проверить возвращаемое им значение:

```
$ cargo run
948146
```

Давайте прогоним вот этот строковый литерал JSON через версию на Python, чтобы удостовериться в результатах:

```
$ echo '{ "name": "Rachelle Ferguson", "value": 948129 }' | python main.py
948146
```

Результаты совпадают! Теперь, когда есть исходный код на языке Rust, который выполняет ту же функциональность, что и небольшой фрагмент исходного кода на Python, осталось написать небольшой склеивающий исходный код, который позволит вызывать функцию Rust из Python.

6.4. Написание модуля расширения Python на языке Rust

В этом разделе будет создан модуль расширения Python. *Модули* в языке Python используются, подобно языку Rust, в качестве организационной единицы для функций, классов и других элементов верхнего уровня. *Модуль расширения* – это модуль, который не пишется на Python, а компилируется с использованием Python’овских библиотек C/C++. Вследствие этого они значительно быстрее обычных модулей Python, но имеют публичные API, которые работают так же, как и обычные модули Python. Язык Rust можно использовать для определения классов, функций, глобальных переменных и других элементов языка Python. Однако в данном случае здесь будут рассмотрены только функции. Давайте начнем.

Прежде всего нужно обновить файл `Cargo.toml`, чтобы включить новую зависимость. Предполагается использовать пакет `pyo3`. Данный пакет предоставляет высокоуровневые привязки языка Rust к интерпретатору Python. Указанные привязки можно использовать как для создания модулей расширения, так и для выполнения произвольного исходного кода на языке Python из Rust. В этой главе будет рассмотрено и то, и другое, но сначала давайте обследуем написание модуля расширения. Откройте файл `Cargo.toml` и обновите его, чтобы он выглядел следующим образом:

```
[package]
name = "rust_json"
version = "0.1.0"
edition = "2018"

[lib]
crate-type = ["cdylib"]

[dependencies]
serde_json = "1.0"
serde = { version = "1.0", features = ["derive"] }
pyo3 = { version = "0.14", features = ["extension-module"] }
```

Поскольку пакет PyO3 обладает самыми разными возможностями, он не включает API модуля расширения по умолчанию. Его необходимо активировать, включив в список используемых функциональных компонентов.

Далее нужно преобразовать исполняемый пакет в библиотечный. Исполняемый пакет содержит файл `main.rs` и может быть скомпилирован в автономный исполняемый файл. Для сравнения, библиотечный пакет содержит файл `lib.rs` и не может исполняться сам по себе; он должен быть включен в какой-либо другой исполняемый пакет. Вспомните, что это различие проводилось ранее, при передаче опции `--lib` команде `cargo new`. В данном случае единственное, что команда `cargo new` делает по-другому, – это создает файл `lib.rs` вместо `main.rs`. Следовательно, миграция довольно проста. Необходимо переименовать файл `main.rs` в `lib.rs` и удалить функцию `main`:

Листинг 6.8 rust_json в качестве библиотеки (lib.rs)

```
#[derive(Debug, serde::Deserialize)]
struct Data {
    name: String,
    value: i32,
}

fn sum(input: &str) -> i32 {
    let parsed: Data = serde_json::from_str(input).unwrap();
    parsed.name.len() as i32 + parsed.value
}
```

Разобравшись с этим, теперь давайте напишем склеивающий исходный код Python! Перво-наперво нужно создать модуль, который интерпретатор Python может успешно импортировать. Затем в этот модуль можно будет добавить функцию `sum`. Давайте создадим базовый модуль, обновив сейчас `lib.rs`.

Листинг 6.9. Пустой модуль расширения

```
use pyo3::prelude::*;

#[derive(Debug, serde::Deserialize)]
struct Data {
    name: String,
    value: i32,
}

fn sum(input: &str) -> i32 {
    let parsed: Data = serde_json::from_str(input).unwrap();

    parsed.name.len() as i32 + parsed.value
}

#[pymodule]
fn rust_json(_py: Python, m: &PyModule) -> PyResult<()> {
    Ok(())
}
```

Здесь появилось несколько новых интересных вещей. Давайте начнем с указания инструкции `use` в первой строке. Обратите внимание, что она заканчивается символом `*`, который называется *подстановочным знаком* и указывает на то, что из модуля `prelude` надо импортировать все имена. `prelude` – это специальный модуль, который (по общепринятым правилам) включает в себя многочисленные типы, необходимые для пользователей конкретного пакета. Обычно пакет создает модули `prelude`, которые реэкспортируют широко используемые типы, чтобы пользователям не нужно было называть их все по именам по отдельности. При разработке одной из этих прелюдий важно обеспечивать, чтобы реэкспорты не конфликтовали с другими глобальными именами. Например, обратите внимание, что все элементы, которые импортируются из пакета PyO3, начинаются с префикса `py`.

Далее, давайте посмотрим на объявление функции `rust_json`. Прежде всего у нее есть атрибут `#[pymodule]`. Подобно атрибуту `#[no_mangle]`, этот атрибут выполняет особую функцию во время компиляции. В отличие от `#[no_mangle]`, он не отключает искажение имени в Rust, а исполняет код во время компиляции, чтобы создать модуль расширения Python с именем `rust_json`. Важно, чтобы функция называлась `rust_json` (так же, как и имя пакета), иначе возникнут проблемы с урегулированием имен при попытке импортировать модуль в Python.

Функция `rust_json` также содержит два неиспользуемых параметра, `Python` и `&PyModule`. Хотя они и не используются, тем не менее оба являются обязательными. Если попытаться удалить любой из них, то атрибут

`#[pymodule]` отклонит функцию. `Python` – это маркерный тип, который указывает на приобретение глобальной блокировки (GIL), а `PyModule` представляет только что созданный модуль Python. Позже в `PyModule` будет добавлена функция `sum`. Указанная функция возвращает тип-обертку `PyResult` вокруг типа `Result` языка Rust, где вариантом с ошибкой является совместимый с Python тип `PyError`.

Разобравшись со структурой пустого модуля, теперь давайте попробуем импортировать его из Python:

```
$ python
>>> import rust_json
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'rust_json'
```

Редко, когда дела идут по плану, согласитесь? Прежде чем можно будет импортировать `rust_json` в Python, нужно скомпилировать модуль расширения в ключе, который был бы понятен интерпретатору Python. Разработчики пакета PyO3 создали инструмент `maturin`, чтобы упростить этот процесс. С его помощью можно настраивать среды разработки для расширений Python, основанных на Rust, или создавать готовые к распространению пакеты. Он устанавливается с помощью менеджера пакетов Python `pip`:

```
$ pip install maturin
```

У инструмента `maturin` есть подкоманда `develop`, которая компилирует исходный код Rust и устанавливает результирующий модуль Python для немедленного использования. Однако есть одно предостережение: ее следует выполнять из виртуальной среды Python. Тема виртуальных сред не будет затрагиваться подробно, но следует знать, что они используются для изоляции зависимостей в проектах Python, чтобы пользователи не могли случайно перезаписывать глобально установленную (возможно, стабильную) версию своего пакета, пока она еще находится в стадии разработки. Давайте теперь создадим и активируем виртуальную среду для целей разработки:

```
$ virtualenv rust-json
$ source rust-json/bin/activate
(rust-json) $
```

Точное имя, которое дается этой виртуальной среде, не играет роли, но обратите внимание, что имя (`rust-json`) теперь появляется перед командной строкой. В будущих листингах этот префикс будет указывать на то, что команда должна выполняться из указанной виртуальной среды. Если открыть новую оболочку или покинуть эту среду, то в нее можно будет войти повторно, снова выполнив команду `source rust-json/bin/activate`. Для выхода выполняется команда `deactivate`.

Настроив виртуальную среду, теперь можно выполнить сборку модуля, установить его и импортировать! Давайте попробуем:

```
(rust-json) $ maturin develop
  Found pyo3 bindings
  Found CPython 3.8 at python
    ... lots of cargo output
  Finished dev [unoptimized + debuginfo] target(s) in 7.49s

(rust-json) $ python
>>> import rust_json
>>> print(rust_json)
<module 'rust_json' from 'rust_json/__init__.py'>
```

Получилось! Наконец-то можно импортировать модуль Python, написанный на Rust, и он не выдает ошибку! Теперь, когда есть пустой модуль, давайте добавим в него функцию `sum`. Это можно сделать, внеся небольшие правки в файл `lib.rs`.

Листинг 6.10 Работающий модуль расширения `rust_json`

```
use pyo3::prelude::*;

#[derive(Debug, serde::Deserialize)]
struct Data {
    name: String,
    value: i32,
}

#[pyfunction]           | Добавлен вот этот
fn sum(input: &str) -> i32 { | атрибут pyfunction.
    let parsed: Data = serde_json::from_str(input).unwrap();

    parsed.name.len() as i32 + parsed.value
}

#[pymodule]
fn rust_json(_py: Python, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(sum, m)?); | Добавлен вот этот вызов
                                                | метода add_function.
    Ok(())
}
```

Были добавлены две новые вещи: в функцию `sum` добавлена атрибутная макрокоманда `pyfunction`, и метод `add_function` теперь вызывается на созданном модуле `PyModule`. Подобно атрибуту `#[pymodule]`, который используется для объявления модуля Python, атрибут `#[pyfunction]` требуется для того, чтобы оберывать функцию Rust в формат, понятный интерпретатору Python.

В строке `add_function` есть несколько интересных функций; немного странная макрокоманда `wrap_pyfunction!` необходима для того, чтобы оберывать функцию `sum` дополнительным слоем совместимости с Python. Добавив функцию `sum` в модуль, теперь давайте попробуем вызвать ее из Python:

```
(rust-json) $ maturin develop
(rust-json) $ python
>>> import rust_json
>>> rust_json.sum('{"name": "Rachelle Ferguson", "value": 948129 }')
948146
```

Получилось! Вы заново реализовали небольшой фрагмент исходного кода на языке Rust и вызывали его из Python. Давайте попробуем интегрировать его в изначальную Python'овскую программу.

Листинг 6.11 Программа на Python с использованием модуля rust_json

```
import sys
import json
import rust_json

s=0

for line in sys.stdin:
    s += rust_json.sum(line)

print(s)
```

И попробуем выполнить ее, вспомнив, что исходный код на Python выдает 6203958:

```
(rust-json) $ python main.py < data.jsonl
6203958
```

Тот же самый результат! Таким образом, на языке Rust была успешно продублирована изначальная функциональность исходного кода на Python. По идее, все должно работать быстрее, но в настоящее время нет способа это подтвердить, который заслуживал бы внимание. Для того чтобы узнать настоящий эффект от проделанной работы, нужно измерить и протестировать производительность.

6.5 Измерение и тестирование производительности в Rust

Тема измерения и тестирования производительности (бенчмаркинг) чревата недоразумениями и путаницей. При ненадлежащем построении тесты на производительность могут выдавать вводящие в заблуждение результаты, которые дают одному экспериментальному пути несправедливое преимущество перед другим. Тесты нередко проводятся в условиях сценариев наилучшего развития событий, чтобы тестировать теоретические пределы производительности системы, при этом реальные результаты никогда не приближаются к тем, которые были получены во время тестирования.

В целях сведения этого риска к минимуму будет применен инструментарий измерения и тестирования производительности под назва-

нием Criterion, который был разработан с нуля с целью обеспечивать простоту в использовании и надежные и правильные результаты. Библиотека Criterion – это пакет Rust, который позволяет измерять и тестировать исходный код на производительность. Criterion можно использовать для замера производительности как исходного кода на Rust, так и исходного кода на Python, используя библиотеку [PyO3](#), чтобы выполнять Python из исходного кода на Rust. Этот процесс чуть более сложный. На рис. 6.2 показано, как все это сочетается.

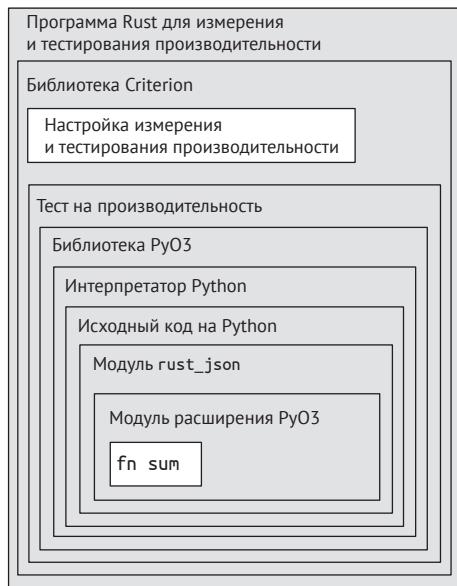


Рисунок 6.2 Анатомия программы для измерения и тестирования производительности

Работа начинается с создания нового пакета, в котором будет храниться исходный код измерения и тестирования производительности. Ввиду ограничений на привязку, которые возникают из-за того, что главный пакет является модулем расширения [PyO3](#), это должен быть обособленный пакет. Если бы это был обычный пакет Rust, то можно было бы оставить исходный код теста в главном пакете. Давайте создадим этот пакет как сестринский каталог пакета [rust_json](#):

```
$ cargo new --lib rust_json
$ ls
main.py
json-sum-benchmark
rust_json
```

Этот новый пакет зависит от пакетов Criterion и PyO3. На этот раз в PyO3 должен быть активирован не [extension-module](#), а другой функциональный компонент. Необходимо добавить функциональный компонент [auto-initialize](#), который упрощает выполнение исходного кода на Python из Rust.

Обычно зависимости добавляют в секцию `dependencies`, но в данном случае они будут размещены в другом месте – в секции `dev-dependencies` файла `Cargo.toml` для зависимостей, которые требуются только при выполнении примеров, модульных и интеграционных тестов и тестов на производительность. При добавлении пакетов, которые полезны только во время теста, таких как Criterion, их необходимо включать в эту секцию, с тем чтобы они не компилировались или компоновались с какой-либо окончательной библиотекой или исполняемым файлом, создаваемым пакетом.

Кроме того, менеджеру пакетов Cargo нужно сообщить о вновь создаваемом файле теста на производительность. Давайте дадим новому файлу имя `py-vs-rust.rs`. Cargo должен знать имя тестового файла, и нужно отключить встроенный в Rust инструментарий измерения и тестирования, который применяется по умолчанию. Встроенный инструментарий работает нестабильно, и его невозможно использовать со стандартным компилятором. Criterion – это более полнофункциональный пакет, поэтому, пропуская встроенный инструментарий, ничего не теряется.

Давайте прямо сейчас добавим эти модули в секцию `dev-dependencies` и новый файл теста на производительность.

Листинг 6.12 Файл Cargo.toml с criterion и pyo3

```
[package]
name = "json-sum-benchmark"
version = "0.1.0"
edition = "2018"

[[bench]]
name = "py-vs-rust"
harness = false

[dependencies]
[dev-dependencies]
criterion = "0.3.5"
pyo3 = { version = "0.14", features = ["auto-initialize"] }
```

Annotations for the Cargo.toml file:

- A bracket under the [dev-dependencies] section points to the text: "Два набора квадратных скобок обязательны; это часть синтаксиса TOML, указывающая на возможность использования нескольких тестов на производительность."
- A bracket under the harness = false line points to the text: "Сообщает Cargo о том, что нужно игнорировать встроенный инструментарий измерения и тестирования, который заменяется на библиотеку Criterion."
- A bracket under the pyo3 dependency line points to the text: "Только базовое имя файла без расширения .rs"

Разобравшись с зависимостями, теперь можно создать файл инструментария измерения и тестирования. Начнем с тестирования чего-то гораздо более простого, чем исходный код на Python: производительности операций сложения с использованием значений типа `u8` и `u128`. Откройте `benchs/py-vs-rust.rs` и добавьте исходный код из следующего ниже листинга.

Листинг 6.13 Базовый пример теста на производительность в benchmarks/py-vs-rust.rs

```
use criterion::black_box, criterion_group, criterion_main, Criterion;

criterion_main!(python_vs_rust);
criterion_group!(python_vs_rust, bench_fn);
```

```
fn bench_fn(c: &mut Criterion) {
    c.bench_function("u8", |b| {
        b.iter(|| {
            black_box(3u8 + 4);
        });
    });

    c.bench_function("u128", |b| {
        b.iter(|| {
            black_box(3u128 + 4);
        });
    });
}
```

Эта программа тестирования производительности в рамках пакета Criterion – одна из самых простых, которые можно написать. Здесь много частей, но все они строятся на том, что вы видели ранее. Давайте рассмотрим их по отдельности.

Первая строка с инструкцией `use` привносит несколько элементов из пакета Criterion, но `use` как инструкция не нова, поэтому идем дальше. Далее идет макрокоманда `criterion_main!`. Поскольку встроенный инструментарий измерения и тестирования производительности был отключен, нужно предоставить свой собственный. Необходимо предоставить функцию `main`, которая будет вызываться при запуске программы. Пакет Criterion предоставляет макрокоманду `criterion_main!` для конструирования этой функции `main`, и она принимает на входе несколько подлежащих выполнению групп Criterion. Указанные группы создаются с помощью макрокоманды `criterion_group!`, и каждая из них содержит несколько подлежащих выполнению функций. Группы представляют собой коллекции функций измерения и тестирования производительности, которые выполняются с одинаковой конфигурацией. В данном случае это дефолтная конфигурация, так как никаких переопределений не указывается.

После вызовов макрокоманд идет функция `bench_fn`:

```
fn bench_fn(c: &mut Criterion) {
    ...
}
```

Имя этой функции не играет роли, но важно, чтобы оно совпадало с именем, указанным при вызове макрокоманды `criterion_group!`. Эта функция должна принимать на входе `&mut Criterion`, то есть структуру менеджера измерения и тестирования производительности. При этом вызывается функция `.bench_function`, которая принимает имя теста (в данном случае `u8`) и замыкание:

```
c.bench_function("u8", |b| {
    b.iter(|| {
        black_box(3u8 + 4);
    });
});
```

Указанное замыкание принимает исполнителя тестов `&mut Bencher` в качестве аргумента, и на нем можно вызывать замыкание `.iter`. Здесь происходит фактический запуск, итерации и измерение. Все, что находится внутри замыкания `.iter`, будет исполняться много раз и измеряться на производительность. В рамках этого замыкания вычисляется результат арифметического выражения `3 + 4`, и он передается в предоставляемую пакетом Criterion функцию `black_box`, которая обеспечивает, чтобы в процессе оптимизации компилятор не удалил вычисление, обнаруживаемое им как неиспользуемое. Для примера с `u128` функции `.bench_function` и `.other` вызываются еще один раз, и он работает точно так же:

```
c.bench_function("u128", |b| {
    b.iter(|| {
        black_box(3u128 + 4);
    });
});
```

ПРИМЕЧАНИЕ. Важно помнить о передаче в функцию `black_box` окончательных результатов тестов, чтобы в процессе оптимизации компилятор не удалил весь тест!

На рис. 6.3 в наглядном виде описано то, что происходит в файле теста на производительность.

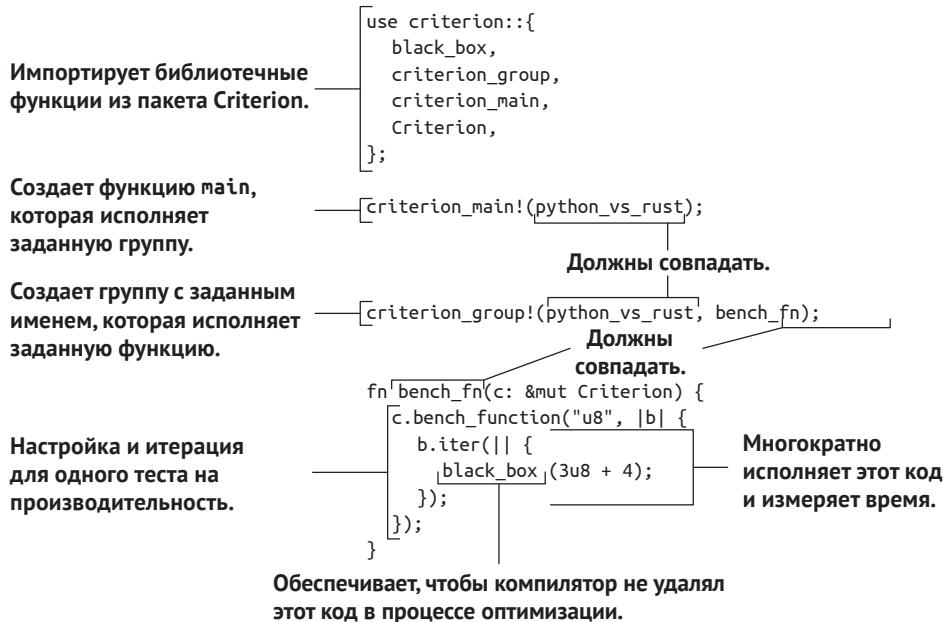


Рисунок 6.3 Анатомия файла теста на производительность

Немного разобравшись с тем, что происходит в файле теста на производительность, теперь давайте исполним тест и посмотрим, какие

результаты он выдает. Его можно исполнить с помощью команды `cargo bench`. Вы должны получить результат, который выглядит примерно так:

```
$ cd json-sum-benchmark
$ cargo bench
    Compiling json-sum-benchmark v0.1.0
    Finished bench [optimized] target(s) in 2.09s
    Running unittests

running 1 test
test tests::it_works ... ignored

test result: ok. 0 passed; 0 failed; 1 ignored; 0 measured; 0 filtered
out;

    Running unittests
Benchmarking u8: Warming up for 3.0000 s
Benchmarking u8: Collecting 100 samples in estimated 5.0000 s (20B
iters)
u8                      time: [257.13 ps 261.71 ps 266.79 ps]

Benchmarking u128: Warming up for 3.0000 s
Benchmarking u128: Collecting 100 samples in estimated 5.0000 s (10B
iters)
u128                    time: [502.27 ps 510.24 ps 521.03 ps]
```

По завершении компиляции сначала прогоняются все модульные тесты и строка `ignored` для каждого из них. Модульный тест `it_works` пишется по умолчанию менеджером Cargo при выполнении команды `cargo new --lib`. Тесты на производительность считаются подмножеством модульных тестов, и поэтому встроенный инструментарий модульного тестирования позволяет пользователям писать тесты на производительность наряду с модульными тестами, именно поэтому они представлены в этом выводе.

Далее тест исполняется на основе полученных данных столько раз, сколько возможно в течение 3 секунд, чтобы прогреть процессор и кеш-память и получить чистые результаты измерений. Затем инструментарий пытается исполнить тест столько раз, сколько возможно в течение 5 секунд, и измеряет время выполнения этих итераций. По оценкам инструментария, он сможет выполнить 20 млрд итераций для версии `u8` и 10 млрд итераций для версии `u128`.

Наконец, по каждому тесту выдается строка, показывающая оценочное время выполнения одной итерации теста в пределах доверительного интервала. Доверительный интервал можно конфигурировать, но по умолчанию он равен 95 %. Первое и последнее числа – это нижняя и верхняя границы интервала, а среднее число – это наилучшая догадка инструментария Criterion о времени, затрачиваемом на каждый интервал. Использование доверительного интервала – отличный способ сократить количество данных с 20 млрд итераций теста до трех чисел. На рис. 6.4 показаны выходные данные по каждому тесту на производительность.

В дополнение к своей простоте эта программа является отличным примером использования пакета Criterion, поскольку она подчеркивает точность библиотеки. Мы зафиксировали разницу в два раза на уровне 0.1 наносекунды – разница составляет 250 триллионных долей секунды. Пакет Criterion очень точен и требует минимальных затрат времени. Он позволяет засекать время и измерять практически все, что вам нужно.

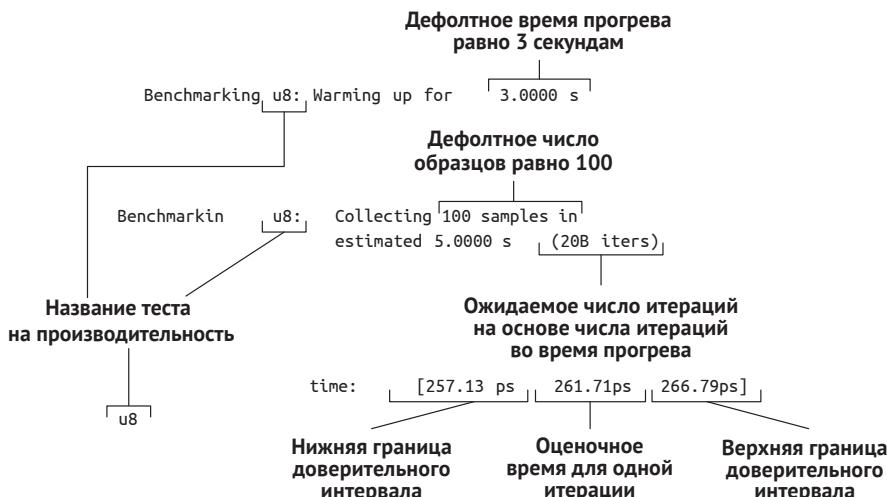


Рисунок 6.4 Анатомия выходных данных инструментария Criterion в командной оболочке

Теперь давайте попробуем применить инструментарий Criterion к рассматриваемому варианту использования. Вспомните, что вы пытаетесь сравнить встроенный в Python модуль `json` с конкретно-прикладным методом `rust_json.sum`, который был написан на языке Rust и выставлен наружу через модуль расширения PyO3.

Для того чтобы протестировать исходный код на Python из Rust, нужно написать исходный код, который использует другую часть API `pyo3`. Она уже применялась для создания кода на Rust, который можно вызывать из Python, но PyO3 также можно использовать для выполнения кода на Python из Rust.

Давайте теперь напишем функцию под названием `bench_py`, которая позволит это сделать. Указанной функции требуется несколько параметров: исполнитель тестов `Bencher` из пакета Criterion, чтобы он мог исполнять тест на производительность, входной строковый литерал, который используется для разбора, и исходный код на Python, который будет выполняться в teste. Вот как эта функция будет выглядеть:

```
use criterion::Bencher;
use pyo3::prelude::*;
use pyo3::types::PyDict;

fn bench_py(b: &mut Bencher, code: &str, input: &str) {
    Python::with_gil(|py| {
        let locals = PyDict::new(py);
```

```

locals.set_item("json", py.import("json").unwгар()).унвгар();
locals
.set_item("rust_json", py.import("rust_json").унвгар())
.унвгар();

locals.set_item("INPUT", input).унвгар();

b.iter(|| black_box(py.run(code, None, Some(&locals)).унвгар()));
});
}

```

В этой функции многое чего происходит. Давайте разложим все по по-лочкам. Функция начинается с вызова функции `Python::with_gil`. Интерпретатор Python нуждается в том, чтобы большинство операций каждого процесса выполнялись из одного потока исполнения, задействуя структуру данных глобальной блокировки. Ключевые структуры данных Python предусматривают, что пользователи будут удерживать глобальную блокировку, и они не являются потокобезопасными. Эти требования не имеют большого значения для обычного исходного кода на Python (за исключением проблем с производительностью, к которым они приводят), но они очень важны при использовании Python'овского С-интерфейса. В пакете PyO3 применяется правило, согласно которому глобальная блокировка всегда удерживается, когда это требуется, и ее приобретают, используя указанную выше функцию `with_gil`. В качестве единственного параметра она принимает функцию, которой самой передается абстрактная ссылка на Python'овскую глобальную блокировку¹. Эта абстрактная ссылка необходима для взаимодействия со многочисленными типами PyO3.

После приобретения глобальной блокировки создается новый словарь `PyDict` для хранения локальных переменных, которые будут внедрены в пример исходного кода. `PyDict` – это принятый в PyO3 эквивалент для создания Python'овского словаря `dict`. Обратите внимание, что для этого действия необходимо использовать абстрактную ссылку на глобальную блокировку, которая была приобретена ранее.

Следующие несколько строк помещают элементы в только что созданный словарь `locals`. Первые две импортируют библиотеки: библиотеку `json`, которая используется в исходном коде теста на производительность на чистом Python, а затем библиотеку `rust_json` для тестирования модуля расширения pyO3. Метод `import` на абстрактной ссылке на глобальную блокировку используется для импорта библиотеки Python и возвращает экземпляр модуля. Функция `set_item`, которая использу-

¹ Глобальная блокировка со стороны интерпретатора (Global Interpreter Lock,abbr. GIL) – это мьютексный механизм, используемый в Python для защиты доступа к объектам Python, предотвращающий одновременное выполнение байт-кода несколькими собственными потоками. Это означает, что глобальная блокировка практически ограничивает способность Python выполнять потоки параллельно на нескольких ядрах процессора, что влияет на истинный параллелизм. Однако потоки могут исполняться конкурентно, то есть они могут находиться в состоянии ожидания, пока другие исполняются. Абстрактная ссылка на глобальную блокировку (или дескриптор глобальной блокировки) позволяет потоку безопасно приобретать и снимать глобальную блокировку. – Прим. перев.

ется в словаре `PyDict`, является обобщенной и может передавать любые типы ключей и значений, которые могут конвертироваться в объекты Python. Последняя строка `set_item` используется для передачи входного строкового литерала из исходного кода на языке Rust в исходный код на языке Python в виде переменной под названием `INPUT`.

Заключительная часть функции выполняет фактический тест на производительность. Вспомните из предыдущего примера, что замыкание `b.iter` использует функцию, которая многократно выполняется инструментарием Criterion и измеряется на предмет производительности. Обратите внимание на отсутствие кода инициализации как составной части этой итерации, чтобы сэкономить время выполнения теста и устраниТЬ возможные источники шума. Внутри этой функции снова используется функция `black_box`; этим обеспечивается, чтобы компилятор не удалял какие-либо вычисления в процессе оптимизации. Вызываемая тут функция `py.run` принимает строковый литерал с подлежащим исполнению исходным кодом на Python и два значения `Option<&PyDict>` для хранения глобальных и локальных переменных. Входные данные хранятся как локальные переменные. На рис. 6.5 показано, как все описанные части работают вместе.

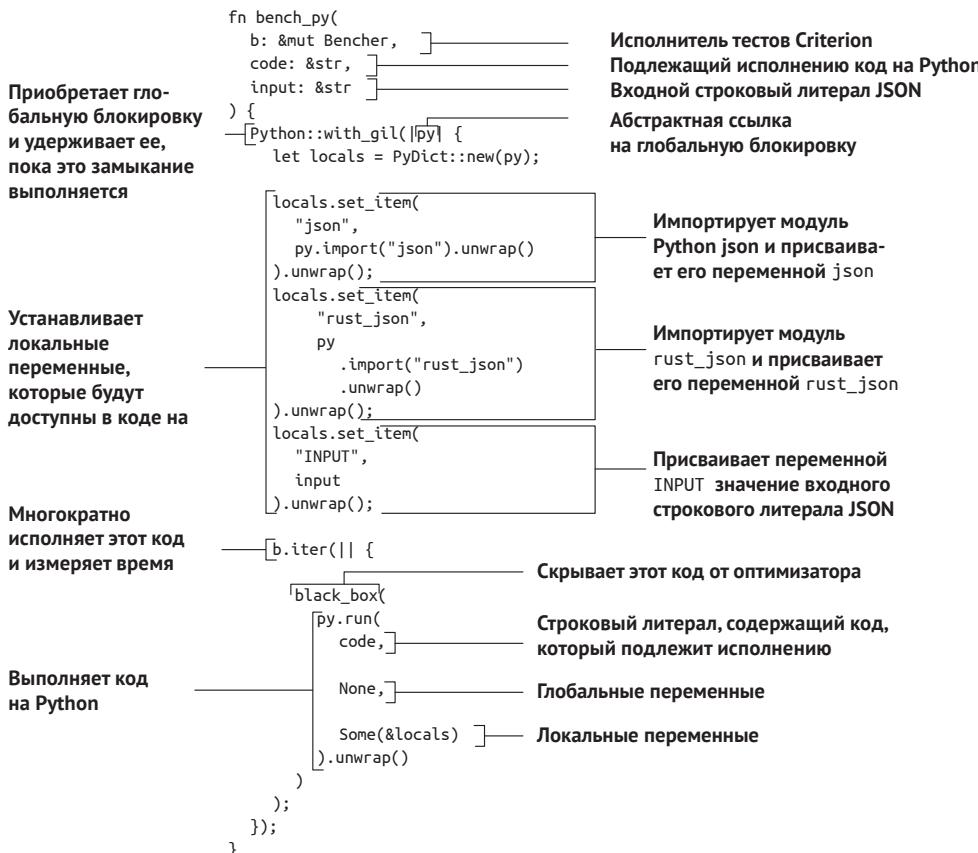


Рисунок 6.5 Диаграмма функции `bench_fn`

Давайте применим эту функцию, чтобы сравнить производительность двух версий исходного кода.

Листинг 6.14 Тестирование производительности чистого Python и модуля расширения Rust

```
use criterion::{
    black_box, criterion_group, criterion_main, Bencher, Criterion,
};

use pyo3::prelude::*;
use pyo3::types::PyDict;

criterion_main!(python_vs_rust);
criterion_group!(python_vs_rust, bench_fn);

fn bench_py(b: &mut Bencher, code: &str, input: &str) {
    Python::with_gil(|py| {
        locals = PyDict::new(py);

        locals.set_item("json", py.import("json").unwrap()).unwrap();
        locals
            .set_item("rust_json", py.import("rust_json").unwrap())
            .unwrap();
        locals.set_item("INPUT", input).unwrap();

        b.iter(|| black_box(py.run(code, None, Some(&locals)).unwrap()));
    });
}

fn bench_fn(c: &mut Criterion) {
    let input = r#"{"name": "lily", "value": 42}"#;

    c.bench_function("pure python", |b| {
        bench_py(
            b,
            "
value = json.loads(INPUT)
s = value['value'] + len(value['name'])
",
            input,
        );
    });

    c.bench_function("rust extension library", |b| {
        bench_py(b, "s = rust_json.sum(INPUT)", input);
    });
}
```

Теперь попробуем выполнить тест на производительность, чтобы убедиться, что все происходит внутри созданной ранее виртуальной среды:

```
(rust-json) $ cd json-sum-benchmark
(rust-json) $ cargo bench
Benchmarking pure python: Collecting 100 samples in estimated
                           5.1074 s (202k iterations)
pure python              time: [25.415 us 25.623 us 25.842 us]
Benchmarking rust extension library: Collecting 100 samples in estimated
                                      5.0931 s (232k iterations)
rust extension library   time: [21.746 us 21.987 us 22.314 us]
```

Минуточку! Версия на Rust едва ли быстрее, чем версия на чистом Python. Вся эта огромная работа была проделана только для того, чтобы увеличить скорость на 10 % по сравнению с базовым Python? Дело в том, что здесь упускается из виду одна важная деталь, которая есть в Rust и которой нет в Python: *оптимизирующий компилятор*.

Давайте немного с ним разберемся.

6.6 Оптимизированные сборки

Возможно, вы помните следующую строку из конца всех команд `cargo build`:

```
Finished dev [unoptimized + debuginfo] target(s) in 2s
```

Эта строка указывает на то, что Cargo компилирует код без активированных оптимизаций. Выполнение оптимизаций во время компиляции увеличивает время компиляции, поэтому по умолчанию они не активированы. Если вы выполняете исходный код на компьютере для разработок в целях тестирования, то, как правило, это может сходить с рук, как это удавалось в данной книге до сих пор. Если вы хотите распространять свой код или выполнять его где-либо в производственной среде, то следует использовать оптимизированные сборки. Сообщать менеджеру пакетов Cargo о необходимости создавать оптимизированные сборки довольно просто; достаточно добавлять флаг `--release` к любой из двух используемых команд: `cargo build` или `cargo run`.

В данном конкретном случае создается модуль расширения `pyo3`, и для этого используется команда `maturin develop`. Данная команда является небольшой оболочкой вокруг команды `cargo build` и принимает многие параметры и флаги, что и Cargo. Она принимает флаг `--release`, поэтому давайте перекомпилируем модуль расширения с этим флагом, чтобы создать оптимизированный двоичный файл:

```
$ (rust-json) cd rust_json
$ (rust-json) maturin develop --release
    Found pyo3 bindings
    Found CPython 3.8 at python
    Compiling pyo3-build-config v0.14.5
    Compiling pyo3-macros-backend v0.14.5
    Compiling pyo3 v0.14.5
    Compiling pyo3-macros v0.14.5
```

```
Compiling rust-json v0.1.0
  Finished release [optimized] target(s) in 7.91s
```

Обратите внимание, что в последней строке теперь указано, что Cargo сгенерировал оптимизированную сборку [**optimized**] в режиме релиза.

Теперь давайте выполним тесты на производительность повторно, чтобы увидеть, как это повлияет на результаты:

```
$ (rust-json) cd json-sum-benchmark
$ (rust-json) cargo bench
  Compiling pyo3-build-config v0.14.5
  Compiling pyo3-macros-backend v0.14.5
  Compiling pyo3 v0.14.5
  Compiling pyo3-macros v0.14.5
  Compiling json-sum-benchmark v0.1.0
    Finished bench [optimized] target(s) in 9.21s
      Running unitests
```

Cargo компилирует
тесты на произ-
водительность
в режиме релиза
по умолчанию.

```
Benchmarking pure python: Collecting 100 samples in estimated
                           5.1069 s (202k iterations)
pure python           time: [25.019 us 25.188 us 25.377 us]
```

```
Benchmarking rust extension library: Collecting 100 samples in estimated
                                         5.0306 s (454k iterations)
rust extension library   time: [10.843 us 10.918 us 10.996 us]
```

Здесь можно наблюдать несколько интересных результатов. Удалось удвоить производительность исходного кода на Rust, просто переключившись на релизную сборку. Версия на Rust теперь работает более чем в два раза быстрее, чем чистый исходный код на Python. Этот пример является изолированным, и во многих случаях замена Python на Rust может приводить к еще более значительному повышению производительности. Вам нужно будет замерить свой собственный исходный код, чтобы определить, какую выгоду вы получите от внедрения языка Rust.

Подводя итоги, вы прошли через поступательный процесс добавления языка Rust в существующее приложение на Python. Вот эти шаги:

- 1 выявление изолированного исходного кода, который можно извлечь;
- 2 написание исходного кода на Rust, который обеспечивает ожидаемое поведение;
- 3 обертывание исходного кода Rust в специфичные для языка привязки;
- 4 компиляция модуля расширения с помощью опции **--release**;
- 5 импорт нового модуля в язык, отличный от языка Rust;
- 6 измерение и тестирование производительности старого и нового путей исходного кода, чтобы получить подтверждение о повышении производительности.

В данной главе был рассмотрен конкретный пример интеграции с языком Python, но аналогичные шаги можно предпринимать и со многими другими динамическими языками. Аналогичные модули доступны для других языков точно так же, как `pyo3`, который используется для интеграции языка Python с языком Rust. Пакет Rutie интегрируется с Ruby, Neon предназначен для Node.js, j4rs и JNI работают с Java, а `flutter_rust_bridge` может использоваться для интеграции с приложениями Flutter.

Краткий итог

- `serde` – это де-факто стандартная экосистема для сериализации и десериализации в языке Rust.
- Атрибут `#[derive(serde::Deserialize)]` позволяет легко выполнять разбор структур из самых разных форматов данных.
- Функциональный компонент `derive` из `serde` должен быть активирован, чтобы использовать макрокоманду `derive`.
- Функция `serde_json::from_str` используется для выполнения разбора в структуру данных Rust из строкового литерала в формате JSON.
- `pyo3` – это пакет Rust, который используется для взаимодействия с интерпретатором Python.
- Активация функционального компонента `extension-module` пакета PyO3 позволяет легко выставлять функции Rust в Python.
- `maturin` – это инструмент командной оболочки, который упрощает разработку модулей Python в Rust.
- Команда `maturin develop` компилирует и устанавливает основанный на Rust Python'овский модуль в виртуальной среде.
- Функциональный компонент `auto-initialize` пакета PyO3 должен быть активирован при выполнении Python'овского исходного кода из Rust.
- Секция `dev-dependencies` в `Cargo.toml` содержит зависимости, используемые для модульных тестов, интеграционных тестов и тестов на производительность.
- Criterion – это пакет Rust для измерения и тестирования производительности исходного кода.
- Секции `bench` в `Cargo.toml` содержат информацию о файлах тестов на производительность.
- Для каждой секции `bench` требуется поле `name` и значение `harness = false`.
- Внутри функции группы измерения и тестирования используются функции `.bench_function` и `.iter`, чтобы выполнять исходный код, который требуется замерить.
- Функция `criterion::black_box` используется для того, чтобы компилятор не удалял исходный код в процессе оптимизации.
- Функция `Python::with_gil` приобретает глобальную блокировку с помощью пакета PyO3.

- **PyDict** – это принятый в пакете PyO3 эквивалент словарных объектов языка Python.
- Функция `.run` используется для выполнения строковых литералов Python'овского исходного кода из Rust.
- Передача опции `--release` многим командам менеджера пакетов Cargo приводит к применению компилятором оптимизации, в результате которой происходит многоократное повышение производительности.



Тестирование интеграций с Rust

Эта глава охватывает следующие ниже темы:

- написание автоматизированных тестов на языке Rust;
- тестирование исходного кода Rust из динамического языка;
- реиспользование существующих тестов с помощью метода динамического латания;
- тестирование нового исходного кода относительно старого с использованием рандомизированных входных данных.

При поставке результатов крупных переработок исходного кода важно удостовериться в надлежащем поведении исходного кода. Приятие некоторой формы автоматизированного тестирования, как правило, считается образцом наилучшей практики в отрасли. В этой главе будут созданы автоматизированные тесты для исходного кода суммирования данных JSON, который был написан в предыдущей главе. Давайте начнем с добавления нескольких модульных тестов в исходный код на языке Rust.

7.1 Написание тестов на языке Rust

В сам язык Rust встроена минимальная система тестирования. Возможно, вы помните краткое упоминание об этом в главе 3. А в главе 2 вы узнали, что при инициализации нового приложения Rust автоматически создается программа «Hello world!». При создании пустой библиотеки происходит аналогичное – предлагается каркас автоматизированного тестирования. Давайте создадим пустой библиотечный пакет под названием `testing`, чтобы поиграть с несколькими тестами, после чего можно будет приступить к применению полученных знаний к библиотеке JSON:

```
$ cargo new --lib testing
```

Теперь откройте файл `testing/src/lib.rs` и посмотрите на готовый тестовый исходный код, который генерируется менеджером пакетов Cargo.

Листинг 7.1 Содержимое только что инициализированной библиотеки Rust

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let result = 2 + 2;
        assert_eq!(result, 4);
    }
}
```

Давайте разберем все части этого файла по полочкам, чтобы понять, насколько они полезны и как соединяются вместе, дабы создать комплект тестов. Начнем с первых двух строк файла, которые содержат ранее не встречавшийся синтаксис:

```
#[cfg(test)]
mod tests {
```

Вторая строка похожа на встроенные модули, которые встречались ранее, но первая строка – это нечто новое. Здесь создается новый модуль под названием `tests`, который будет содержать все функции тестирования библиотеки. Первая строка – это атрибутная макрокоманда под названием `cfg`, которая позволяет сообщать компилятору о необходимости компилировать или пропускать определенные части исходного кода при работе в определенных обстоятельствах. Например, можно было бы создать версии функции для конкретных операционных систем и использовать `cfg`, чтобы осуществлять контроль за тем, какую версию компилировать в зависимости от целевой операционной системы. Разработчики могут создавать конкретно-прикладные флаги условной компиляции, которые предоставляют пользователям возможность включать в компиляцию или исключать из нее целые функциональные компоненты.

Эти флаги можно прикреплять к любому элементу – к функции, структуре, общей черте, блоку или, в данном случае, к модулю. Поскольку `cfg(test)` находится на уровне модуля, все, что находится внутри модуля `tests`, будет компилироваться только тогда, когда компилятор компилирует тесты. Вследствие этого в сборки исполняемого файла или библиотеки тесты не включаются. За счет этого уменьшается размер двоичного файла и ограничивается количество строк кода, которые должны проверяться компилятором.

ПРИМЕЧАНИЕ. Рекомендация размещать тесты внутри модуля с атрибутной макроМандой `#cfg(test)` не является строго обязательной к исполнению, но считается образцом лучшей практики.

Размещение всех тестов внутри модуля позволяет легко исключать исходный код тестирования из производственных сборок без необходимости прикреплять атрибутную макроМанду `#cfg(test)` ко всем тестовым функциям. За счет этого снижается риск непреднамеренного использования тестового значения или функции и сокращается размер двоичных файлов.

Далее взглянем на функцию внутри модуля `it_works`:

```
#[test]
fn it_works() {
```

Как и во многих других языках, отдельной единицей тестирования в Rust (минимальной частью, которая может отказать или пройти успешно) является функция. В отличие от некоторых других языков, имена тестовых функций в Rust не играют роли. Они полезны только для общения с разработчиком. Вместо этого атрибутная макроМанда `#test` сигнализирует компилятору о том, какие функции содержат тесты. В данном случае тест `it_works` удостоверяется, что `2 + 2` равно `4`. Давайте заглянем внутрь функции, чтобы понять, как это делается:

```
let result = 2 + 2;
assert_eq!(result, 4);
```

МакроМанда `assert_eq!` сравнивает два переданных ей значения на предмет равенства. Если они не равны, то это неравенство вызывает панику в исполняющем тест потоке исполнения. Тестовый фреймворк улавливает панику, и тест помечается как «отказавший» с сообщением, содержащим отладочное (`Debug`) представление обоих значений, чтобы помочь в отладке теста. МакроМанда `assert_eq!` не предназначена только для тестирования; ее можно использовать в любом исходном коде Rust, но из-за особенностей большинства автоматизированных тестов она появляется в них довольно регулярно.

Можно писать тесты, в которых макроМанда `assert_eq!` вообще отсутствует. Так, макроМанда `assert!` ровно так же удостоверяется в том, что любое переданное в нее булево значение является `true`, и поднимает панику, если это не так. Кроме того, можно писать тесты, которые удостоверяются только в том, что функции не возвращают

ошибок, и это достигается с помощью методов `.unwrap()` или `.expect()` без применения макрокоманд `assert!/assert_eq!`. На рис. 7.1 показаны наиболее важные части тестового модуля.

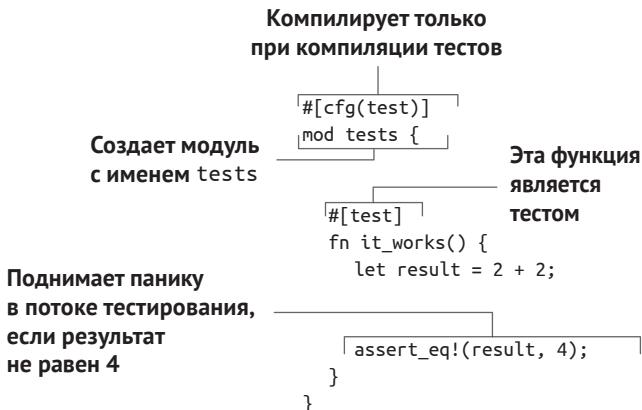


Рисунок 7.1 Диаграмма тестового модуля

Разобравшись в том, как части теста сочетаются друг с другом, теперь давайте посмотрим, как выглядит прогон теста:

```

$ cargo test
Compiling testing v0.1.0
Finished test [unoptimized + debuginfo] target(s) in 0.31s
Running unittests

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

test result: ok. 1 passed; 0 failed;

Doc-tests testing

running 0 tests
    
```

Наиболее важной частью этого результата является строка с именем теста рядом с `ok`, что указывает на успешный прогон теста. Давайте также посмотрим, что будет, когда в список добавляется безуспешный тест. Добавьте тест `it_does_not_work` в файл `lib.rs`.

Листинг 7.2 Тест, который завершается отказом

```

#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let result = 2 + 2;
        assert_eq!(result, 4);
    }
}
    
```

```

#[test]
fn it_does_not_work() {
    let result = 2 + 2;
    assert_eq!(result, 5);    ← Констатация, что  $2 + 2 = 5$ ,
}                                            всегда дает сбой.
}

```

Давайте выполним его:

```

$ cargo test
Compiling testing v0.1.0
Finished test [unoptimized + debuginfo] target(s) in 0.33s
Running unit tests

running 2 tests
test tests::it_works ... ok
test tests::it_does_not_work ... FAILED

failures:

---- tests::it_does_not_work stdout ----
thread 'tests::it_does_not_work' panicked at 'assertion failed:
  left: `4`,
  right: `5`, testing/src/lib.rs:12:5
note: run with `RUST_BACKTRACE=1` environment variable to display
backtrace

failures:
  tests::it_does_not_work

test result: FAILED. 1 passed; 1 failed;

error: test failed, to rerun pass '--lib'

```

Этот результат содержит много информации. Тест `it_works` по-прежнему проходит, но тест `it_does_not_work` выделен как откazавший. После списка тестов можно увидеть захваченный `stdout` из откazавшего теста, который показывает два значения, переданных в макрокоманду `assert_eq!`. Из этих значений можно понять, где была допущена ошибка. Также имеется имя файла и номер строки откazавшей макрокоманды `assert_eq!`. Вспомните из главы 2, что примечание о `RUST_BACKTRACE` является обобщенным и печатается всякий раз, когда поток исполнения поднимает панику.

Встроенный в Rust фреймворк тестирования по умолчанию захватывает потоки данных `stdout` и `stderr` и эмитирует их на консоль. Они хранятся в памяти и эмитируются только при отказе теста. Следовательно, во время прогона теста можно распечатывать столько сообщений журнала регистрации событий, сколько потребуется, и выходные данные останутся чистыми. Давайте посмотрим, как это работает, добавив немного выходных данных в тесты.

Листинг 7.3 Запись в stdout и stderr из тестов

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        eprintln!("it_works stderr");
        println!("it_works stdout");
        let result = 2 + 2;
        assert_eq!(result, 4);
    }

    #[test]
    fn it_does_not_work() {
        eprintln!("it_does_not_work stderr");
        println!("it_does_not_work stdout");
        let result = 2 + 2;
        assert_eq!(result, 5);
    }
}
```

И давайте посмотрим, как выглядит их вывод на консоль:

```
$ cargo test
Compiling testing v0.1.0
Finished test [unoptimized + debuginfo] target(s) in 0.31s
Running unittests

running 2 tests
test tests::it_works ... ok
test tests::it_does_not_work ... FAILED

failures:

---- tests::it_does_not_work stdout ----
it_does_not_work stderr
it_does_not_work stdout
thread 'tests::it_does_not_work' panicked at 'assertion failed:
  left: `4`,
  right: `5`, testing/src/lib.rs:16:5
note: run with `RUST_BACKTRACE=1` environment variable to display
backtrace

failures:
  tests::it_does_not_work
```

Обратите внимание, что из результата теста выходят потоки данных `stdout` и `stderr`, объединенные под знаменем `stdout`, но из теста `it_works` не выходит ни одного сообщения. Иногда бывает полезно получать полные выходные потоки данных из всех тестов, отключая захват. Это делается путем передачи флага `--nocapture` в тестовый двоичный файл. Важно отметить, что этот флаг передается в тестовый двоичный файл, а не в Cargo. Это делается, используя дополнительную опцию `--no-run`:

тельные `--`, чтобы отделить аргументы для Cargo от аргументов для тестового двоичного файла. Давайте сейчас это сделаем:

```
$ cargo test -- --nocapture
    Finished test [unoptimized + debuginfo] target(s) in 0.03s
        Running unittests

running 2 tests
it_does_not_work stderr
it_does_not_work stdout
thread 'tests::it_does_not_work' panicked at 'assertion failed:
  left: `4`,
  right: `5`, testing/src/lib.rs:16it_works stderr
:5it_works stdout

note: run with `RUST_BACKTRACE=1` environment variable to display
backtrace
test tests::it_works ... ok
test tests::it_does_not_work ... FAILED
failures:

failures:
    tests::it_does_not_work

test result: FAILED. 1 passed; 1 failed;

error: test failed, to rerun pass '--lib'
```

←
←

**Обратите внимание на stderr
теста it_works в конце этой строки.**

**Обратите внимание на
stdout теста it_works
в конце этой строки.**

Возможно, это немного сложно увидеть, но обратите внимание, что теперь результат теста `it_works` выходит вместе с тестом `it_does_not_work`. Однако выходные потоки данных смешаны воедино, так как по умолчанию Rust выполняет тесты параллельно. Это можно немного исправить, выполнив тесты только из одного потока исполнения, что управляется с помощью аргумента `--test-threads`:

```
$ cargo test -- --nocapture --test-threads=1
    Finished test [unoptimized + debuginfo] target(s) in 0.03s
        Running unittests

running 2 tests
test tests::it_does_not_work ... it_does_not_work stderr
it_does_not_work stdout
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `4`,
  right: `5`, chapter-07/listing_03_stdout/src/lib.rs:16:5
note: run with `RUST_BACKTRACE=1` environment variable to display
backtrace
FAILED
test tests::it_works ... it_works stderr
it_works stdout
ok
```

```

failures:

failures:
    tests::it_does_not_work

test result: FAILED. 1 passed; 1 failed;

```

Теперь результаты расположены независимо друг от друга, но последовательное исполнение тестов оказывается на времени исполнения. Обычно при прогоне тестов не требуется распечатывать все результаты и нежелательно выполнять все тесты последовательно, как показано выше. Пока что давайте удалим выходной исходный код и сбойный тест. Теперь исходный код должен выглядеть как стартовый исходный код библиотечного пакета:

```

#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let result = 2 + 2;
        assert_eq!(result, 4);
    }
}

```

При написании пакетов Rust, которые будут использоваться другими пользователями, также рекомендуется документировать свои функции. К сожалению, документация и примеры часто устаревают. На помощь приходит встроенная в Rust система, которая поддерживает выполнение примеров исходного кода в документации через систему тестирования. Давайте рассмотрим краткий пример, чтобы понять, как это работает.

7.1.1 Документарные тесты

Представьте себе, что вы пишете небольшую функцию `add`, которая принимает два числа и складывает их. Вы хотите максимально упростить применение исходного кода разработчику, использующему вашу библиотеку, поэтому вы пишете несколько комментариев. Давайте добавим эту функцию в файл `lib.rs` за пределами модуля `tests`.

Листинг 7.4 Функция add

```

// Сложить два числа типа i32 и вернуть результат сложения
pub fn add(x: i32, y: i32) -> i32 {
    x+y
}

```

Сейчас при просмотре исходного кода этот комментарий выглядит достаточно разумно, но в Rust встроена мощная система документирования, к которой можно получить доступ, слегка изменив комментарий. Вместо стандартного комментария с двумя символами косой черты при использовании трех косых черт создается *документарный комментарий*,

или сокращенно doc-комментарий. Такие комментарии связаны с элементами, которые будут подхватываться встроенной в Rust системой документирования. Давайте прямо сейчас создадим один из них.

Листинг 7.5 Документарный комментарий для функции add

```
/// Сложить два числа типа i32 и вернуть результат сложения
pub fn add(x: i32, y: i32) -> i32 {
    x+y
}
```

С точки зрения исходного кода разница невелика, но давайте посмотрим, что можно с этим сделать. Сгенерируем документацию для библиотеки и посмотрим на результат:

```
$ cargo doc --open
```

Приведенная выше команда генерирует документацию для всех публичных элементов в пакете и открывает веб-браузер для доступа к этой документации. Кликните по функции `add`, чтобы увидеть ее сигнатуру и только что написанный документарный комментарий, как показано на рис. 7.2.

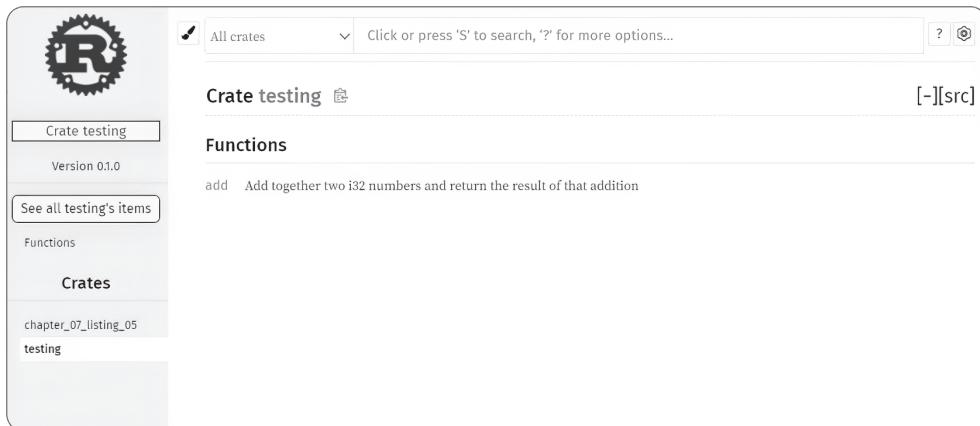


Рисунок 7.2. Скриншот документации для функции `add`

В дополнение к самой документации в документарные комментарии можно добавлять примеры, которые проверяются при прогоне тестов. Давайте сейчас добавим несколько из них. Для полноты картины будут добавлены успешный, сбойный и некомпилируемый тесты.

Листинг 7.6 Документарные тесты

```
/// Сложить два числа типа i32 и вернуть результат сложения
/// ``
/// assert_eq!(testing::add(2, 2), 4);
```

```

/// ``
/// ``
/// ``
/// use testing::add;
/// assert_eq!(add(2, 2), 5);
/// ``
/// ``
/// ``
/// use testing::add;
/// assert_eq!(add("hello", 2), 5);
/// ``
pub fn add(x: i32, y: i32) -> i32 {
    x+y
}

```

Обратите внимание на приведенные выше блоки кода – они написаны в формате упрощенной разметки Markdown. Документарные комментарии поддерживают синтаксис упрощенной разметки с целью создания списков, ссылок, выделений жирным шрифтом, курсивом и многоного другого. Также важно отметить, что каждый документарный комментарий компилируется в виде обособленного пакета. Вследствие этого он имеет доступ только к публичному API пакета, и необходимо либо импортировать элементы из своего пакета, либо использовать полный путь; для пользователей пакета эти элементы должны быть примерами публичного API.

Обратите внимание, что второй документарный тест не проходит. Он содержит констатацию, что $2 + 2 = 5$, что является бессмыслицей. Третий тест даже не будет скомпилирован, так как он пытается передать строковый срез "hello" туда, где требуется тип i32. Давайте посмотрим, как встроенная в Rust система тестирования показывает этот сбой в документации:

```

$ cargo test
Compiling testing v0.1.0
Finished test [unoptimized + debuginfo] target(s) in 0.30s
Running unittests

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

test result: ok. 1 passed; 0 failed;

Doc-tests chapter-07-listing-06

running 3 tests
test src/lib.rs - add (line 11) ... FAILED
test src/lib.rs - add (line 2) ... ok
test src/lib.rs - add (line 6) ... FAILED

failures:

---- src/lib.rs - add (line 11) stdout ----

```

```

error[E0308]: mismatched types
--> src/lib.rs:13:16
|
5 |     assert_eq!(add("hello", 2), 4);
|           ^^^^^^ expected `i32`, found `&str`
|
error: aborting due to previous error

For more information about this error, try `rustc --explain E0308`.
Couldn't compile the test.
---- src/lib.rs - add (line 6) stdout ----
Test executable failed (exit code 101).

stderr:
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `4`,
  right: `5`', src/lib.rs:5:1
note: run with `RUST_BACKTRACE=1` environment variable to display
a backtrace

failures:
src/lib.rs - add (line 11)
src/lib.rs - add (line 6)

test result: FAILED. 1 passed; 2 failed;

```

В этом коде нет обособленной команды документарного тестирования; при выполнении команды `cargo test` выполняются все типы тестов. Из теста `it_works` приходит `ok`, и он немедленно приступает к прогону документарных тестов.

Некомпилируемый документарный тест не блокирует компиляцию всего теста. О нем сообщается только как о части отдельного документарного теста, который завершился отказом.

Обратите внимание на то, как выглядят эти отказы. Оба они указывают на отказ в строке 5, но это не соответствует строке файла, в которой появляются ошибки. Это связано с тем, что документарные тесты обернуты в неявную функцию `main`, и номера строк, поступающие из этих панических сообщений, ненадежны. Вместо этого следует смотреть на номер строки теста. Сообщения `src/lib.rs - add (line 6)` и `src/lib.rs - add (line 11)` указывают на блоки кода, в которых начинаются сбойные документарные тесты. Теперь можно обновить пример, чтобы он содержал правильный код.

Листинг 7.7 Прохождение документарных тестов

```

/// Сложить два числа типа i32 и вернуть результат сложения
/// ``
/// assert_eq!(testing::add(2, 2), 4);
/// ``

```

```

/// ...
/// ``
/// use testing::add;
/// assert_eq!(add(3, 2), 5);
/// ``
pub fn add(x: i32, y: i32) -> i32 {
    x+y
}

```

Прогон тестов теперь показывает, что они проходят, как положено:

```

$ cargo test
Compiling testing v0.1.0
Finished test [unoptimized + debuginfo] target(s) in 0.41s
Running unit tests

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

test result: ok. 1 passed; 0 failed;

Doc-tests testing

running 2 tests
test src/lib.rs - add (line 2) ... ok
test src/lib.rs - add (line 6) ... ok

test result: ok. 2 passed; 0 failed;

```

Давайте также обновим документацию, чтобы увидеть, как примеры будут выглядеть для пользователей пакета. Это показано на рис. 7.3:

```
$ cargo doc --open
```



Рисунок 7.3. Скриншот документации для функции add с документарным тестом

Поняв, как писать тесты в целом, теперь давайте добавим несколько тестов для созданного в главе 6 пакета `rust_json`.

7.1.2 Добавление тестов в существующий исходный код

Откройте файл `lib.rs` из пакета `rust_json`. Он должен выглядеть следующим образом.

Листинг 7.8 `rust_json/src/lib.rs` из главы 6

```
use pyo3::prelude::*;

#[derive(Debug, serde::Deserialize)]
struct Data {
    name: String,
    value: i32,
}

#[pyfunction]
fn sum(input: &str) -> i32 {
    let parsed: Data = serde_json::from_str(input).unwrap();

    parsed.name.len() as i32 + parsed.value
}

#[pymodule]
fn rust_json(_py: Python, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(sum, m)?);

    Ok(())
}
```

Теперь давайте создадим модуль `test` и напишем базовый тест.

Листинг 7.9 Базовый тест для `rust_json::sum`

```
...
#[cfg(test)]
mod tests {
    use crate::sum;

    #[test]
    fn test_stokes_baker() {
        assert_eq!(
            sum("{ \"name\": \"Stokes Baker\", \"value\": 954832 }"),
            954844
        );
    }
}
```

Давайте выполним тест, чтобы убедиться, что он работает:

```
$ cargo test
    Compiling rust_json
    Finished test [unoptimized + debuginfo] target(s) in 7.56s
    Running unitests

running 1 test
test tests::test_stokes_baker ... ok

test result: ok. 1 passed; 0 failed;
```

Приведенный выше тест удостоверяется в надлежащем поведении исходного кода при небольшом объеме входных данных, но есть возможность кое-что улучшить. Прежде всего немного раздражают все эти экранирующие элементы в строковом литерале, позволяющие заключать в двойные кавычки. К счастью, в Rust есть способ обойти эту назойливость с помощью *нераразобранного строкового литерала*.

Нераразобранные строковые литералы

Нераразобранные строковые литералы (англ. raw string) – это строковые литералы, которые не выполняют разбор экранирующих последовательностей и могут открываться/закрываться чем-то иным, кроме двойной кавычки. Обычный строковый литерал превращается в нераразобранный строковый литерал путем размещения буквы `r` непосредственно перед открывающей кавычкой. Этот символ отключает экранирующие последовательности внутри строкового литерала. Давайте попробуем проделать это со строковым литералом JSON в новом тесте:

```
sum(r"{"name": "Stokes Baker", "value": 954832 }),
```

Если сейчас попытаться выполнить тест, то он не будет скомпилирован! К тому же ошибка довольно длинная, и в ней трудно разобраться:

```
$ cargo test
    Compiling rust_json v0.1.0
error: unknown start of token: \
--> src/lib.rs:30:21
  |
30 |     sum(r"{"name": "Stokes Baker", "value": 954832 }),           ^
  |
error: suffixes on a string literal are invalid
--> src/lib.rs:30:11
  |
30 |     sum(r"{"name": "Stokes Baker", "value": 954832 }),           ^
  |           ^^^^^^^^^^ invalid suffix `name`  

  
error: expected one of `)` , `,` , `.` , `?` , or an operator,
      found `": " "Stokes Baker", "value": 954832 }```
--> src/lib.rs:30:22
  |
30 |     sum(r"{"name": "Stokes Baker", "value": 954832 }),
```

```

operator
|           -^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
|           expected one of `)` , `.` , `?` , or an
|           help: missing `,` , `.` , `?` , or an

error[E0061]: this function takes 1 argument but 2 arguments were
supplied
--> src/lib.rs:30:7
|
30 |     sum(r#"{"name": "Stokes Baker", "value": 954832 }"),  

|     ^^^ -----  

|     supplied 2 arguments  

|     |  

|     expected 1 argument
|

```

Данная ошибка возникает из-за того, что конвертация строкового литерала в неразобранный строковый литерал отключает экранирующие последовательности, которые позволяют использовать двойные кавычки. Когда компилятор видит первую двойную кавычку перед буквой `n` в `name`, он теперь трактует ее как конец строки. На рис. 7.4 показано, как теперь компилятор выполняет разбор этого кода.

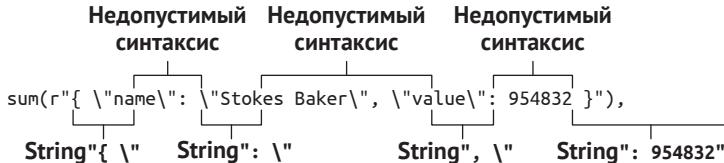


Рисунок 7.4 Разбор неразобранного строкового литерала

В настоящее время этот код хуже прежнего, который может компилироваться и исполняться. Ошибки можно исправить с помощью существующего в Rust умного дополнения для неразобранных строковых литералов. В начале и конце строкового литерала можно использовать разделитель, отличный от двойной кавычки. Кроме того, можно подбивать двойные кавычки любым количеством решеточных знаков (или «октоторпов»; это «знак числа», «знак фунта», «знак хеша», `#`). Предпринимая эти шаги, разблокируется способность писать строковые литералы, содержащие двойные кавычки, без их экранирования. Исходный код выглядит следующим образом:

```
sum(r#"{"name": "Stokes Baker", "value": 954832 }"#),
```

Этот метод упрощает чтение строковых литералов в формате JSON. Тут был применен только один решеточный знак, но если бы нужно было написать литерал `"#"` внутри него, то в начало и конец строкового литерала можно было бы добавить столько решеточных знаков, сколько потребуется, чтобы обозначить его начало и конец, например:

```
println!("{}", r###"hello"#world"##how are you today?"###);
```

Приведенная выше строка кода печатает строковый литерал

```
hello"#world"##how are you today?
```

Этот код работает, потому что требуется указать двойную кавычку и три решеточных знака, чтобы закончить строковый литерал, а внутренние элементы содержат только один или два решеточных знака. В полном исходном коде новый неразобранный строковый литерал выглядит следующим образом.

Листинг 7.10 Неразобранный строковый литерал, используемый в тесте JSON

```
use pyo3::prelude::*;

#[derive(Debug, serde::Deserialize)]
struct Data {
    name: String,
    value: i32,
}

#[pyfunction]
fn sum(input: &str) -> i32 {
    let parsed: Data = serde_json::from_str(input).unwrap();

    parsed.name.len() as i32 + parsed.value
}

#[pymodule]
fn rust_json(_py: Python, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(sum, m)?);

    Ok(())
}

#[cfg(test)]
mod tests {
    use crate::sum;

    #[test]
    fn test_stokes_baker() {
        assert_eq!(
            sum(r#"{"name": "Stokes Baker", "value": 954832 }"#),
            954844
        );
    }
}
```

Строка, в которую
было внесено из-
менение.

И давайте проверим, что тест по-прежнему проходит:

```
$ cargo test
```

```
Finished test [unoptimized + debuginfo] target(s) in 8.33s
Running unitests
```

```
running 1 test
test tests::test_stokes_baker ... ok

test result: ok. 1 passed; 0 failed;
```

Прежде чем перейти к тестированию исходного кода Rust из языка Python, давайте добавим еще несколько тестовых случаев на будущее.

Листинг 7.11 Дополнительные тестовые случаи для исходного кода на языке Rust

```
...

#[cfg(test)]
mod tests {
    use crate::sum;

    #[test]
    fn test_stokes_baker() {
        assert_eq!(
            sum(r#"{"name": "Stokes Baker", "value": 954832 }"#),
            954844
        );
    }

    #[test]
    fn test_william_cavendish() {
        assert_eq!(
            sum(r#"{"name": "William Cavendish", "value": -4011 }"#),
            -3994
        );
    }

    #[test]
    fn test_ada_lovelace() {
        assert_eq!(
            sum(r#"{"name": "Ada Lovelace", "value": 18151210 }"#),
            18151222
        );
    }
}
```

И они все должны проходить:

```
$ cargo test
Finished test [unoptimized + debuginfo] target(s) in 7.15s
Running unitests

running 3 tests
test tests::test_ada_lovelace ... ok
```

```
test tests::test_stokes_baker ... ok
test tests::test_william_cavendish ... ok

test result: ok. 3 passed; 0 failed;
```

Отлично! Теперь, когда на языке Rust написано несколько базовых тестов, давайте посмотрим, как новый исходный код на Rust будет использовать существующие тесты, написанные относительно изначальной Python'овской реализации.

7.2 Тестирование исходного кода Rust с использованием языка Python

В этом разделе будет затронута тема обновления существующих Python'овских тестов, чтобы охватить ими новый исходный код на языке Rust в дополнение к существующему исходному коду на языке Python. Существующие тесты написаны на языке Python с использованием фреймворка `pytest`. Python'овский фреймворк `pytest` был разработан для того, чтобы упрощать написание небольших, удобочитаемых тестов.

ПРИМЕЧАНИЕ. В этом разделе требуется управление виртуальными средами Python, и предполагается, что вы используете настройки виртуальной среды, основанные на инструкциях из главы 6. Если у вас нет таких настроек, то исходный код в этом разделе работать не будет.

Сначала давайте установим фреймворк `pytest` в виртуальной среде `rust-json`:

```
(rust-json) $ pip install pytest
...
Successfully installed
    attrs-21.4.0
    configparser-1.1.1
    packaging-21.3
    pluggy-1.0.0
    py-1.11.0
    pyparsing-3.0.7
    pytest-7.0.1
    tomli-2.0.1
```

Ради напоминания в следующем ниже листинге показан изначальный исходный код на языке Python.

Листинг 7.12 Программа на языке Python, которая подлежит тестированию

```
import sys
import json
```

```
s=0

for line in sys.stdin:
    value = json.loads(line)
    s += value['value']
    s += len(value['name'])

print(s)
```

Для того чтобы сделать этот исходный код более пригодным для тестирования, он будет преобразован в функцию с заданными входными и выходными данными, а не просто что-то, что оперирует на `stdin/stdout`. Теперь программа будет выглядеть так, как показано в следующем ниже листинге.

Листинг 7.13 Программа на языке Python после обновления под использование функции

```
import sys
import json

def sum(lines_iter):
    s=0

    for line in lines_iter:
        value = json.loads(line)
        s += value['value']
        s += len(value['name'])

    return s

if __name__ == '__main__':
    print(sum(sys.stdin))
```

Эта конструкция языка Python аналогична функции `main` в других языках.

Представьте себе, что файл `pytest` уже настроен и внутри него есть один-единственный готовый к прогону тест. Указанный тест выполняется с использованием 10 строк данных с известными свойствами и известным суммарным значением. Этот тестовый файл называется `main_test.py`.

Листинг 7.14 Тестовый файл main_test.py

```
import main

def test_10_lines():
    lines = [
        '{ "name": "Stokes Baker", "value": 954832 }',
        '{ "name": "Joseph Solomon", "value": 279836 }',
        '{ "name": "Gonzalez Koch", "value": 140431 }',
        '{ "name": "Parrish Waters", "value": 490411 }',
        '{ "name": "Sharlene Nunez", "value": 889667 },
```

```
'{ "name": "Meadows David", "value": 892040 }',
'{ "name": "Whitley Mendoza", "value": 965462 }',
'{ "name": "Santiago Hood", "value": 280041 }',
'{ "name": "Carver Caldwell", "value": 632926 }',
'{ "name": "Tara Patterson", "value": 678175 }',
]
assert main.sum(lines) == 6203958
```

Фреймворк `pytest` будет обнаруживать любую функцию, начинаяющуюся с `test_`, и автоматически ее выполнять. В данном случае он будет трактовать `test_10_lines` как тест и выполнит его при вызове `pytest`. Давайте сейчас это сделаем, чтобы убедиться, что он работает, как положено, после чего начнем вносить изменения:

```
(rust-json) $ pytest -v
=====
platform linux -- Python 3.8.10, pytest-7.0.1, pluggy-1.0.0
cachedir: .pytest_cache
collected 1 item

main_test.py::test_10_lines PASSED [100%]

=====
```

На практике рекомендуется, чтобы один раз тест заканчивался отказом, поэтому давайте внесем в исходный код поправку и выполним тест повторно. Для этого функция `sum` будет обновлена прибавлением 1 в возвращаемое значение, что должно привести к отказу теста.

Листинг 7.15 Версия файла main.py, которая не проходит тест

```
import sys
import json

def sum(lines_iter):
    s=0

    for line in lines_iter:
        value = json.loads(line)
        s += value['value']
        s += len(value['name'])

    return s + 1
```

Обратите внимание на дополнительные + 1.

```
if __name__ == '__main__':
    print(sum(sys.stdin))
```

Теперь если выполнить тест повторно, то он завершится отказом и включит сообщение об ошибке:

```
(rust-json) $ pytest -v
=====
platform linux -- Python 3.8.10, pytest-7.0.1, pluggy-1.0.0
cachedir: .pytest_cache
collected 1 item

main_test.py::test_10_lines FAILED
[100%]

=====
FAILURES =====
test_10_lines
=====
def test_10_lines():
    lines = [
        '{ "name": "Stokes Baker", "value": 954832 }',
        '{ "name": "Joseph Solomon", "value": 279836 }',
        '{ "name": "Gonzalez Koch", "value": 140431 }',
        '{ "name": "Parrish Waters", "value": 490411 }',
        '{ "name": "Sharlene Nunez", "value": 889667 }',
        '{ "name": "Meadows David", "value": 892040 }',
        '{ "name": "Whitley Mendoza", "value": 965462 }',
        '{ "name": "Santiago Hood", "value": 280041 }',
        '{ "name": "Carver Caldwell", "value": 632926 }',
        '{ "name": "Tara Patterson", "value": 678175 }',
    ]

>     assert main.sum(lines) == 6203958
E     assert 6203959 == 6203958
E         +6203959
E         -6203958

main_test.py:17: AssertionError
=====
short test summary info =====
FAILED main_test.py::test_10_lines - assert 6203959 == 6203958
===== 1 failed in 0.01s =====
```

Теперь уберите `+ 1` из конца инструкции `return` и выполните тест повторно, чтобы удостовериться, что функциональность была восстановлена. Далее давайте обновим программу на Python под использование Rust'овской библиотеки суммирования данных JSON.

Листинг 7.16 Программа на Python, переписанная под использование библиотеки Rust

```
import sys
import rust_json

def sum(lines_iter):
    s=0

    for line in lines_iter:
        s += rust_json.sum(line)
```

```

    return s

if __name__ == '__main__':
    print(sum(sys.stdin))

```

После внесения этого изменения тест должен по-прежнему проходить успешно:

```

(rust-json) $ pytest -v
=====
test session starts =====
platform linux -- Python 3.8.10, pytest-7.0.1, pluggy-1.0.0
cachedir: .pytest_cache
collected 1 item

main_test.py::test_10_lines PASSED
[100%]

=====
1 passed in 0.01s =====

```

В более крупном приложении, надо надеяться, большее количество тестов будет задействовать больше путей исполнения кода Rust. Обновление тестов под использование нового пути исполнения кода – это, конечно, хорошо, но было бы неплохо иметь возможность тестировать версию Rust относительно изначальной версии Python более непосредственно, чтобы определять их отличие друг от друга, и если оно есть, то чем они отличаются. Для этого можно создать тест, который выполняет две версии на рандомизированных входных данных и сравнивает выходные данные.

Прежде чем добавить рандомизацию, давайте напишем служебную функцию, которая позволит выполнять функцию `sum`, поддерживающую изначальным кодом на Python либо новой функцией на Rust. Это будет сделано с использованием *динамического латания*.

7.2.1 Динамическое латание

Динамическое латание¹ – это процесс динамического пересмотра элементов в программах, который обычно используется при написании модульных тестов, чтобы переключать глубокие зависимости от одной версии к другой или заменять реальные ресурсы ввода-вывода поддельными. Давайте посмотрим, как написать функцию, в которой динамическое латание задействуется для того, чтобы вызывать две разные версии исходного кода суммирования.

Далее будет добавлен тест и вспомогательная функция, которая сравнивает две версии. Также нужно будет предоставить изначальную

¹ Под динамическим латанием (англ. monkey patching) понимается практика динамического изменения или расширения поведения библиотек или модулей во время выполнения, часто путем добавления новых методов или изменения существующих. Этот метод обычно используется в динамических языках программирования для изменения поведения кода без изменения изначального исходного кода. – Прим. перев.

Python'овскую реализацию функции, чтобы использовать ее для переопределения Rust'овской версии.

Листинг 7.17 Тест, сравнивающий результаты работы версий на языках Rust и Python

```
from pytest import MonkeyPatch

def test_compare_py_rust():
    compare_py_and_rust(
        ['{"name": "Stokes Baker", "value": 95483 }']
    )

def python_sum(line):
    import json

    value = json.loads(line)
    return value['value'] + len(value['name'])

def compare_py_and_rust(input):
    rust_result = main.sum(input)

    with MonkeyPatch.context() as m:
        m setattr(main.rust_json, 'sum', python_sum)
        py_result = main.sum(input)

    assert rust_result == py_result
```

Мы не собираемся слишком долго останавливаться на точном синтаксисе Python, который здесь требуется, но давайте все же немного разложим происходящее на части:

```
from pytest import MonkeyPatch
```

Сначала нужно импортировать класс `MonkeyPatch` из `pytest`. Этот класс позволяет позже переопределить функцию `rust_json.sum`:

```
def test_compare_py_rust():
    compare_py_and_rust(
        ['{"name": "Stokes Baker", "value": 95483 }']
    )
```

Новый тест выполняет вспомогательную функцию сравнения с одним известным входным элементом данных. В будущем этот тест будет обновлен под передачу рандомизированных входных данных:

```
def python_sum(line):
    import json

    value = json.loads(line)
    return value['value'] + len(value['name'])
```

Далее перересматривается изначальная Python'овская реализация функциональности, чтобы использовать ее в качестве базовой ли-

нии, относительно которой будет сравниваться новый исходный код на языке Rust. В данном случае функциональность была перемещена в сам тестовый файл. Это делать не обязательно, но все же было сделано ввиду того, что изначальная Python'овская реализация больше не используется в главной программе:

```
def compare_py_and_rust(input):
    rust_result = main.sum(input)

    with MonkeyPatch.context() as m:
        m setattr(main.rust_json, 'sum', python_sum)
        py_result = main.sum(input)

    assert rust_result == py_result
```

Наконец, идет сама функция сравнения. Эта функция выполняет функцию `sum`, используя функцию `rust_json.sum` и функцию `python_sum`, а затем сравнивает результаты. В ней используется метод `MonkeyPatch.context`, чтобы создать небольшую область в коде, где функция `main.rust_json.sum` переопределяется функцией `python_sum`. Давайте выполним этот тест, чтобы убедиться, что он работает, как положено:

```
$ pytest -v
=====
 test session starts =====
platform linux -- Python 3.8.10, pytest-7.0.1, pluggy-1.0.0
cachedir: .pytest_cache
collected 2 items

main_test.py::test_10_lines PASSED [50%]
main_test.py::test_compare_py_rust PASSED [100%]
===== 2 passed in 0.01s =====
```

Давайте также вкратце повторим ошибку в исходном коде, чтобы удостовериться, что констатация не срабатывает в случае, когда результаты из Python не совпадают с результатами из Rust. На этот раз будет добавлена ошибка в исходный код на языке Rust. Изменим значение, возвращаемое из функции `sum` файла `lib.rs`.

Листинг 7.18 Библиотека Rust с добавленным дефектом

```
use pyo3::prelude::*;

#[derive(Debug, serde::Deserialize)]
struct Data {
    name: String,
    value: i32,
}

#[pyfunction]
```

```

fn sum(input: &str) -> i32 {
    let parsed: Data = serde_json::from_str(input).unwrap();

    parsed.name.len() as i32 + parsed.value + 10  ◀
}

#[pymodule]
fn rust_json(_py: Python, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(sum, m)?);
}

Ok(())
}

```

Обратите внимание на дополнительные + 10.

Теперь пересоберем исходный код Rust и выполним повторный прогон Python'овских тестов:

```

$ cd rust_json
$ cargo build
$ cd ..
$ pytest -v -k test_compare_py_rust
=====
test session starts =====
platform linux -- Python 3.8.10, pytest-7.0.1, pluggy-1.0.0
cachedir: .pytest_cache
collected 2 items / 1 deselected / 1 selected

main_test.py::test_compare_py_rust FAILED
[100%]
=====
FAILURES =====
----- test_compare_py_rust -----

```

...

```

>     assert rust_result == py_result
E     assert 954854 == 954844  ◀
E         +954854
E         -954844

```

Для краткости результат усечен.

Обратите внимание на разницу между значениями.

```

main_test.py:38: AssertionError
=====
short test summary info =====
FAILED main_test.py::test_compare_py_rust - assert 954854 == 954844
===== 1 failed, 1 deselected in 0.02s =====

```

После выполнения тест завершается безуспешно из-за дополнительных + 10, которые были добавлены в исходный код на Rust. Обратите внимание, что результат из Rust, переменная `rust_result`, теперь на 10 больше, чем результат из Python, сохраненный в переменной `py_result`.

Вернем исходный код на Rust назад в рабочее состояние и выполним тесты повторно, чтобы убедиться, что все работает:

```

$ cd rust_json
$ cargo build

```

```
$ cd ..
$ pytest -v
=====
platform linux -- Python 3.8.10, pytest-7.0.1, pluggy-1.0.0
cachedir: .pytest_cache
collected 2 items

main_test.py::test_10_lines PASSED [50%]
main_test.py::test_compare_py_rust PASSED [100%]
=====
2 passed in 0.01s =====
```

Разобравшись в самом принципе работы динамического латания, теперь добавим в тест рандомизацию, чтобы убедиться, что он работает с неизвестными входными данными. При этом снова будет написана вспомогательная функция, чтобы пропускать один тестовый случай через исходный код, а затем вызывать ее из тестовой функции-исполнителя.

Эта тестовая функция Python исполняет функцию `randomized_test_case` 100 раз. Каждый раз генерируется от 100 до 500 строк в формате JSON, каждая из которых содержит значение `name` длиной от 100 до 200 символов в нижнем регистре ASCII и число `value`, представляющее собой случайное целое число в диапазоне от 0 до 10 000.

Листинг 7.19 Рандомизированный тест, сравнивающий результаты из Python и Rust

```
import json
import string
import random

...

def test_random_inputs(monkeypatch):
    for _ in range(100):
        randomized_test_case(monkeypatch)

def randomized_test_case(monkeypatch):
    number_of_lines = random.randint(100, 500)

    lines = []
    for _ in range(number_of_lines):
        number_of_chars = random.randint(100, 200)

        lines.append(json.dumps({
            'name': ''.join(random.choices(
                string.ascii_lowercase,
                k=number_of_chars,
            )),
            'value': random.randint(0, 10_000),
        }))
```

```
compare_py_and_rust(monkeypatch, lines)
```

```
...
```

После того как будет сконструирован список строк в формате JSON, в ранее определенную функцию сравнения будет подан список данных.

Тестовая функция имеет высокую степень рандомности и может обнаруживать в библиотеке закоулки, которые не были раскрыты в тестах, написанных от руки. Приведенный выше подход является довольно грубым способом рандомизированного тестирования. Для «тестирования свойств» разрабатываются специализированные библиотеки, которые способны более разумно конструировать входные значения, чтобы проходить пути исполнения кода. В данном контексте достаточно этой тестовой функции. Количество тестовых случаев можно легко контролировать, увеличивая количество итераций в функции `test_random_inputs`, что также увеличивает время прогона теста. При увеличении данного числа исполнителю тестов придется выполнять больше работы, и в результате можно легко сделать тест, на выполнение которого будет уходить несколько часов.

Здесь интересно то, что уже существует реализация на Python, относительно которой можно тестировать исходный код на языке Rust. При этом можно непрерывно генерировать случайные входные данные и подавать их как в исходный код на Python, так и в исходный код на Rust, чтобы проверять, что обе библиотеки эмитируют одинаковые результаты.

В этой главе содержится много информации о тестировании и документировании. Применяя эти сведения и навыки работы с ними, можно получать более прочную уверенность в результатах переработки исходного кода при их развертывании в производственных системах.

Краткий итог

- По общепринятому правилу, тесты Rust должны размещаться в модуле `tests` рядом с тестируемым исходным кодом.
- Добавление атрибутной макрокоманды `#[cfg(test)]` к элементу приводит к тому, что этот элемент будет компилироваться только во время компиляции тестов.
- Исходный код на языке Rust можно протестировать, написав для него функции с атрибутной макрокомандой `#[test]`.
- Макрокоманда `assert_eq!` позволяет поднимать панику на teste, если два значения не равны.
- Команда `cargo test` компилирует, обнаруживает и выполняет все тестовые функции.
- Добавление документарных комментариев (`///`) перед элементом приводит к добавлению информации в автогенерируемую документацию.

- Команда `cargo doc` компонует документацию для пакета.
- Команда `cargo doc -open` компонует документацию для пакета и открывает ее в дефолтном веб-браузере.
- Добавление блока кода (`````) внутри документарного комментария позволяет написать пример внутри документации, который также будет скомпилирован и исполнен в качестве теста.
- Неразобранные строковые литералы позволяют пропускать экранирующие символы, которые в противном случае пришлось бы экранировать.
- Неразобранные строки имеют префикс `r` и должны содержать одинаковое количество решеточных символов (#) в начале и в конце (это число может быть равно нулю).
- Динамическое латание используется во многих динамических языках для внедрения зависимостей там, где в противном случае это было бы затруднительно. Оно применяется для тестирования исходного кода с разными версиями одной и той же функции.



Асинхронный Python с языком Rust

Эта глава охватывает следующие ниже темы:

- написание вычислительно интенсивных приложений на языке Python;
- повышение производительности приложений за счет использования потоков исполнения;
- экстернализация модуля на языке Rust, чтобы повышать асинхронную производительность за счет возможностей языка Rust по масштабированию.

Язык Python идеально подходит для прототипирования. Этот титул он заслужил благодаря своей простоте и гибкости. В основу данного языка, который был разработан Гвидо ван Россумом в 1980-х годах и выпущен в 1991 году, изначально была положена цель построения более совершенного языка программирования (преемника языка АВС). Сначала ван Россум разработал интерпретатор языка и среду исполнения; затем постепенно разработал первые версии языка. С этого момента люди начали осознавать мощь языка, который было легко читать и на котором было легко писать. Вынуждая разработчиков использовать отступы для блоков кода, Python автоматически придает исходному коду приложения некоторую структуру. Поскольку этот язык является интерпретируемым, разработчики имеют возможность быстро убеждаться в рабо-

способности своих приложений без необходимости компилировать исходный код, что в крупных проектах может занимать много времени. По мере своего развития Python становился все более популярным и остается таким и по сей день. Гибкость и простота языка снизили барьер для тех, кто работает в академических и исследовательских кругах, и поэтому многие изыскательские проекты и производственные системы выполняются на Python. Это привело к появлению бесчисленного числа библиотек математического и имитационного моделирования, которые используются при разработке моделей машинного обучения и глубокой переработки данных (т. н. добычи данных).

Поскольку данный язык является интерпретируемым, положенный в основу интерпретатор может быть написан на самых разных языках, что приводит к таким проектам, как Jython, который позволяет получать доступ к библиотекам языка Java, и IronPython, который поддерживает платформу .NET. Однако, возможно, вы больше знакомы с CPython или Python, написанным на языке C, который для многих устанавливается по умолчанию. В ситуации, когда интерпретатор написан на C, Python может читать и использовать библиотеки языков C и C++. Одна только эта способность предоставляет языку Python массу библиотек и некоторые преимущества в производительности.

Тем не менее существуют компромиссы. Python – это простой язык, что обеспечивает быструю разработку, но, как вы увидите, и ущерб быстродействию. Python также лишен некоторой гибкости, так как не обеспечивает типобезопасность «прямо из коробки». Будучи интерпретируемым языком, Python должен опираться на интерпретатор, чтобы справляться со всеми этими частями. Возможно, вы задаетесь вопросом, а какое место занимает язык Rust в этом мире интерпретаторов языка Python. Для ответа на этот вопрос будет задействован стандартный дистрибутив Python под названием Cython, в котором, как уже упоминалось, используются библиотеки языка C, а следовательно, он может взаимодействовать с языком Rust. И снова будет использована библиотека PyO3 из главы 6, но вместо обработки исходного кода на языке Python языком Rust теперь язык Python будет обрабатывать исходный код на языке Rust.

В этой главе будет рассмотрено написание вычислительно интенсивной функции на Python. Затем будут найдены способы масштабирования приложения, чтобы вызывать эту функцию несколько раз и измерять улучшения по мере постепенного продвижения к языку Rust. Мы увидим, что язык Rust снова обеспечивает защищенность, необходимую для быстроты – даже на языке Python.

8.1 Генерирование множества Мандельброта на языке Python

Бенуа Мандельброт известен в математике своими исследованиями в области фрактальной геометрии и был одним из первых, кто использовал компьютерную визуализацию в своих исследованиях. Фрактальная геометрия – это увлекательный раздел математики, предметом

изучения которого является рекурсивная природа функций и создаваемых ими структур, как показано на рис. 8.1. При увеличении масштаба фрактала обнаруживается, что он никогда не заканчивается, а продолжает генерировать формы и узоры. Это свойство бывает очень полезно при выполнении определенных вычислений, например при расчете неправильных форм, таких как береговые линии. Однако зависимость фрактальной геометрии от рекурсивных определений и комплексных чисел придает ей вычислительную сложность и, следовательно, высокую вычислительную затратность.

С помощью языка Python и его удивительных математических библиотек приложение, генерирующее набор Мандельброта, создается относительно просто. Что интересно касательно многих стержневых модулей и библиотек Python, так это то, что по соображениям производительности они иногда написаны на языках С и С++. Поэтому вычислительно интенсивные модули, такие как Pillow, которые используются в математических расчетах, написаны на С и С++. Другими словами, производительность используемого кода не обязательно ограничивается языком Python, а зависит от того, как работает Python.

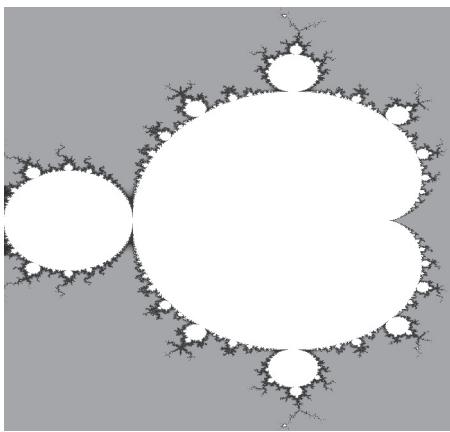


Рисунок 8.1. При увеличении масштаба множества Мандельброта будут содержать повторяющиеся узоры, так как такие множества являются рекурсивными по своей природе

Python – это интерпретируемый язык, и далее в данной главе вы увидите, что из этого следует. Раз он интерпретируемый, для него требуется интерпретатор, который использует Python'овский байт-код и исполняет приложение. Компилируемые языки, такие как C и Rust, компилируют исходный код на основе операционной системы, в которой он, как предполагается, будет работать. Этот скомпилированный код создает особый тип кода, именуемый *объектным кодом*, который понятен операционной системе и процессору. Байт-код похож на объектный код в том смысле, что он интерпретируется опорной системой. Однако, в отличие от объектного кода, который зависит от процессора, на котором выполняется код, байт-код интерпретируется опорной виртуальной средой.

дой исполнения. Такие языки, как Python и Java, предоставляют интерпретатор или среду исполнения, то есть приложение, которое работает в заданной системной конфигурации, а байт-код остается неизменным.

Таким образом, в конечном счете Python – это язык, который может интерпретироваться другими языками, имеющими реализацию среди исполнения Python, что наделяет язык Python возможностью совместимо оперировать с этими языками. Наиболее распространенной реализацией Python является дистрибутив CPython, в котором исходный код Python выполняется посредством интерпретатора, написанного на языке С, и, следовательно, может использовать модули и библиотеки языка С. Поскольку язык Rust тоже имеет операционную совместимость с языком С, вы в конечном итоге увидите, как эти пути сходятся. Тем временем, прежде чем внедрять язык Rust, будет задействована существующая взаимосвязь между языками Python и С.

Давайте посмотрим, как из множества Мандельброта создать изображение, подобное показанному на рис. 8.1, на языке Python, используя существующие библиотеки, а затем переработать приложение, чтобы повысить его производительность. Для начала давайте создадим новый каталог и виртуальную среду, чтобы изолировать Python'овский проект. Для этого создайте новый каталог и перейдите в него. Затем создайте виртуальную среду и установите пакет Pillow.

Листинг 8.1 Консоль: инициализация проекта

```
python -m venv venv
./venv/bin/activate
pip install Pillow
```

После того как все это будет сделано, можно приступить к созданию алгоритма и отрисовке изображения из множества Мандельброта. Откройте новый файл с именем `main.py` и добавьте следующее.

Листинг 8.2 main.py: использование алгоритма Мандельброта

```
from PIL import Image
def mandelbrot_func(size: int, path: str, range_x0: float, range_y0: float,
                     range_x1: float, range_y1: float):
    image = Image.new(mode='RGB', size=(size, size))
    size_f = float(size)

    x_range = abs(range_x1 - range_x0)
    x_offset = x_range / 2.0

    y_range = abs(range_y1 - range_y0)
    y_offset = y_range / 2.0

    for px in range(size):
```

`from PIL import Image` Импортирует математическую библиотеку Pillow.

`def mandelbrot_func(size: int, path: str, range_x0: float, range_y0: float,` Создает новое изображение.

`range_x1: float, range_y1: float):`

`image = Image.new(mode='RGB', size=(size, size))` Создает границы, которые множество Мандельброта будет использовать для вычисления.

`size_f = float(size)` Выполняет итерации по каждому пикселу изображения.

`x_range = abs(range_x1 - range_x0)`

`x_offset = x_range / 2.0`

`y_range = abs(range_y1 - range_y0)`

`y_offset = y_range / 2.0`

`for px in range(size):`

```

for py in range(size):
    x0 = float(px) / size_f * x_range - x_offset
    y0 = float(py) / size_f * y_range - y_offset

    c = complex(x0, y0)
    i=0
    z = complex(0, 0)

    while i < 255:
        z = (z * z) + c
        if float(z.real) > 4.0:
            break
        i += 1

    image.putpixel((px, py), (i, i, i))
image.save(path)

```

Создает комплексное число, которое будет содержаться внутри множества.

Помещает пиксели в изображение на основе вычисления.

Сохраняет изображение по указанному пути.

mandelbrot_func(1000, "single.png", -5.0, -2.12, -2.5, 1.12)

Вы заметите, что выполнение исходного кода занимает некоторое время, так как для вычисления значения каждого отдельного пикселя на изображении размером 1000×1000 пикселов в нем используются комплексные числа. Каждое вычисление характеризуется интенсивностью и обходится дорого, и все результаты записываются в одно изображение. Теперь представьте себе, что требуется создать службу, которая генерирует несколько экземпляров этих изображений. Как бы вы ее сконструировали?

8.2 Масштабирование

Решения о переработке исходного кода обусловлены самыми разными причинами, но все они направлены на то, чтобы каким-то образом улучшить исходный код. Системы быстро эволюционируют, и вскоре становится очевидным, что исходный код работает не так хорошо, как хотелось бы. Места в исходном коде, где происходят замедления, называются *бутылочными горлышками*, названными так из-за того, что горлышко бутылки ограничивает общий поток из бутылки, являясь *узким местом*. Точно так же самая медленная функция нередко определяет пропускную способность, которую можно получить из всей системы в целом. Как только система достигает предела производительности, у обладателей системы есть два варианта: переписать исходный код и масштабировать.

Эти два варианта часто идут рука об руку, поскольку системы можно масштабировать двумя способами: горизонтально и вертикально. Горизонтальное масштабирование означает добавление дополнительных экземпляров к уже работающей службе. Это эквивалентно развертыванию дополнительных серверов или какой-либо другой машины либо процесса. Суть горизонтального масштабирования состоит в том, чтобы дублировать или клонировать существующую систему в том виде, в каком она существует на данный момент, без изменения конфигурации самой машины. Вертикальное масштабирование оз-

начает добавление большего количества ресурсов к существующему экземпляру машины. Это эквивалентно добавлению более быстрого центрального процессора с дополнительными ядрами или дополнительной памятью. Дело в том, что сама система обладает большим количеством ресурсов и, теоретически, может делать больше, если приложение сможет воспользоваться этим преимуществом.

Однако в некоторых случаях ни вертикальное, ни горизонтальное масштабирование невозможно без изменения исходного кода. При горизонтальном масштабировании потребуется изменить исходный код, чтобы он работал как распределенная система, в которой сервер должен работать как группа, а не как отдельный экземпляр. Для того чтобы понять, как это работает, можно запустить несколько экземпляров своих программ на Python одновременно. В Unix-подобной системе вы бы набрали следующее.

Листинг 8.3 Консоль: одновременное выполнение нескольких процессов

```
python main.py & python main.py
```

Этот код просит операционную систему запустить одно и то же приложение дважды и выполнять одновременно, но в разных процессах. Если бы вы посмотрели на выходные данные, то увидели бы только один результат. Хотя система масштабирована горизонтально, она не обрабатывает выходные данные как уникальные значения. Поэтому при сдвоенном запуске приложения на выходе будет получено изображение `single.png`, и процессы будут перезаписывать друг друга. Хотя сама математическая функция является *идемпотентной*, структура службы не была запрограммирована на запись в уникальный выходной канал. Идемпотентные системы отличают свои задания и выходные данные уникальным образом. Это может делаться инициирующей системой, придавая сущности некую уникальную идентификацию. В нашем примере не было уникального имени файла, поэтому произошел конфликт, и файлы были перезаписаны. Если бы нужно было изменить выходные данные на идемпотентные, то можно было бы добавить уникальный ИД или временную метку, чтобы гарантировать, что они не будут перезаписаны. При исполнении этих функций в Unix-подобной системе одной командой `&` операционная система будет исполнять эти скрипты одновременно. Выполнение нескольких версий одного и того же исходного кода без изменения ресурсов называется *горизонтальным масштабированием*.

Горизонтальное масштабирование происходит естественным образом по мере роста системы, поскольку оно увеличивает ее объем, как показано на рис. 8.2. Но когда приложение выполняется на нескольких серверах, требуется определенный уровень координации как на уровне маршрутизации, так и на уровне системы. Прежде всего для координации необходимо, чтобы работал внешний механизм, который распределяет задания между работающими службами. В веб-приложениях это обычно делается с помощью балансировщика нагрузки. Работа балансировщика нагрузки в точности соответствует

его названию: распределять нагрузку от поступающих в систему вызовов между различными работающими процессами. Он выполняет это с помощью простой или сложной логики, возможно, чередуя службы и проверяя их текущую нагрузку. При этом нежелательно, чтобы два процессора подхватывали и выполняли одно и то же задание и выполняли избыточную работу. Это влечет за собой проверку существующих записей или маркировку задания как находящегося в процессе.



Рисунок 8.2 Горизонтальное масштабирование означает добавление большего количества физического оборудования

Поскольку данное изложение носит скорее архитектурный характер, оно не совсем укладывается в то, что мы пытаемся сделать в этой книге. Однако важно отметить, что добавление дополнительных серверов – это одно из эффективных решений, которое осуществляется за счет балансировки нагрузки и очередей, но не обязательно в полной мере действует преимущества оборудования, на котором оно работает. То есть у этого конкретного решения компромиссом являются не временные затраты, а денежные. Напротив, вертикальное масштабирование помогает повышать производительность без дополнительных денежных затрат (что радует руководителей компаний).

Вертикальное масштабирование означает добавление большего количества ресурсов к существующей системе и поиск способа подразделять работу таким образом, чтобы ее можно было обрабатывать небольшими порциями обособленными процессами, используя увеличенную мощность, а не несколько машин, чтобы выполнять отдельные задания. Проблема возникает в том, как подразделять работу. Вертикальное масштабирование – это все равно, что увеличивать объем оперативной памяти компьютера или увеличивать количество ядер центрального процессора, как показано на рис. 8.3. Его суть – совершенствовать существующую систему и ускорять обработку данных. Однако если при-



Рисунок 8.3 Вертикальное масштабирование означает добавление дополнительных ресурсов к существующему оборудованию

ложение не разработано надлежащим образом, то оно превращается в пластырь. Для того чтобы воспользоваться преимуществами вертикального масштабирования, приложение должно быть способным правильно пользоваться преимуществами новых ресурсов. Это больше становится проблемой программного обеспечения, чем то, что мы видели при горизонтальном масштабировании. Однако решение проблемы аналогично в той части, что работа должна быть распределена по всему процессу. Вместо балансировщика нагрузки или какой-либо внешней системы, управляющей распределяемыми заданиями, подразделять работу должны приложение или операционная система. Теперь встает проблема с координацией, возможно, внутри системы. Это сложная проблема, и не все языки справляются с ней хорошо.

К сожалению, язык Python не очень хорошо справляется с одновременным выполнением нескольких процессов. Компромиссы встречаются в каждом языке, и это один из тех, который используется в языке Python, чтобы поддерживать простоту языка. Давайте рассмотрим, как переработать исходный код под вертикальное масштабирование. Для этого внесем в него правки, чтобы в течение одного процесса создавалось несколько изображений одновременно. Достаточно будет просто добавить цикл.

Листинг 8.4 main.py: простой цикл

```
def main():
    for i in range(0,8):
        mandelbrot(1000, f"{i}.png", -5.0, -2.12, -2.5, 1.12)

if __name__ == "__main__":
    main()
```

Если выполнить этот исходный код локально, то будет получено время, равное примерно 46 секундам. Это не очень хорошо, но послужит основой для улучшения. На данный момент проблема заключается в том, что весь этот процесс выполняется синхронно, то есть прежде чем каждый процесс сможет начать работу, он ожидает завершения других. Это называется *блокированием*, то есть процесс не может продолжить работу до тех пор, пока не будет доступен предоставленный ресурс. Для того чтобы улучшить службу, желательно, чтобы процессы были независимы друг от друга, не дожидаясь завершения. Это называется *асинхронной обработкой*, при которой работа может выполняться без ожидания ответа. Давайте посмотрим на принцип ее работы.

8.3 Библиотека *asyncio*

Конвертация существующей службы в службу, выполняющую асинхронные задания, довольно проста, но, как обсуждалось ранее, требуется инструмент, который служил бы для управления этим процессом.

Асинхронные задания полезны тем, что мы сообщаем системе о том, что все в порядке, если придется подождать результата, а значит, процессору безразлично, какой результат будет возвращен первым. Это дает опорной системе (в данном случае интерпретатору Python) свободу возвращать результаты, как только они у нее появятся. Отпадает необходимость в ожидании задания 1 перед заданием 2. Кроме того, система больше не является детерминированной, так как неизвестно, какой процесс вернет результат первым.

Для того чтобы Python мог использовать процессы с разветвлением по выходу и по входу, перед определением метода добавляется ключевое слово `async`. Этим компилятору сообщается о том, что он может продолжать обработку, и ответ в конечном итоге придет, когда функция вернет результат. Затем нужно проделать то же самое с функцией `main`, но добавить метод для выполнения нескольких заданий на нескольких потоках исполнения. В конечном итоге нужно, чтобы все процессы завершились до завершения работы приложения. Здесь будет использоваться команда, которая собирает результаты в один всеобъемлющий результат. Наконец, нужно иметь что-то, чем можно управлять и выполнять задания в таком ключе. Давайте взглянем на пример в следующем ниже листинге.

Листинг 8.5 `main.py`: использование `async` и `asyncio`

```
from PIL import Image
import asyncio
```

`Импортирует библио-`
`теку asyncio.`

```
async def mandelbrot_func(
    size: int,
    path: str,
    range_x0: float,
    range_y0: float,
    range_x1: float,
    range_y1: float):
```

`Изменяет функцию`
`на асинхронную.`

```
    ...
async def main():
    await asyncio.gather(*[
        mandelbrot_func(1000, f"{i}.png", -5.0, -2.12, -2.5, 1.12)
        for i in range(0, 8)
    ])
asyncio.run(main())
```

`Делает функцию`
`main асинхронной.`

`Раскручивает несколько эк-`
`земпляров и ожидает возвра-`
`та до тех пор, пока все они не бу-`
`дут завершены.`

`Запускает асинхронное`
`выполнение.`

Когда эта команда запускается локально, общее время, необходимое для ее исполнения на компьютере одного из авторов, составило около 42 секунд. Это лучше по сравнению с первоначальными 46 секундами, но ненамного. Теперь, как узнать, что данный пример выполняется асинхронно? Если посмотреть на выходные данные, то, вероятно, можно увидеть, что задания выполняются по порядку, что

немного разочаровывает, но давайте посмотрим, можно ли его поменять принудительно. Для этого добавим несколько строк в функцию Мандельброта, чтобы попробовать.

Листинг 8.6 main.py: добавление состояния сна

```
from PIL import Image
from random import randint
import asyncio

async def mandelbrot_func(
    size: int,
    path: str,
    range_x0: float,
    range_y0: float,
    range_x1: float,
    range_y1: float):
    s = randint(1,5)           ← Создает случайное число для тестирования.
    print(f"{path} sleeping for {s} seconds")
    await asyncio.sleep(s)     ← Приостанавливает поток Python и позволяет другим процессам исполняться.
    ...

    ...
```

При локальном выполнении примера видна следующая закономерность:

```
0.png sleeping for 3 seconds
1.png sleeping for 1 seconds
2.png sleeping for 4 seconds
3.png sleeping for 2 seconds
4.png sleeping for 1 seconds
5.png sleeping for 4 seconds
6.png sleeping for 1 seconds
7.png sleeping for 5 seconds
1.png created
4.png created
6.png created
3.png created
0.png created           ← 0.png – это первое созданное изображение, и ему нужно быть в состоянии сна 3 секунды.
2.png created
5.png created
7.png created           ← Однако изображение 0.png не создается ровно в момент, когда оно готово (прошло более 3 секунд).
```

Все эти задания были поставлены в рабочую очередь, но они исполнялись в зависимости от их доступности, когда они не были заблокированы состоянием сна. Вот почему вы видите результаты, возвращаемые в полуслучайном порядке. Первое изображение (0.png), которое планируется создать, будет находиться в состоянии сна в течение 3 секунд. Если посмотреть на времена сна различных «завершенных» изображений, которые появляются до создания 0.png (1.png, 4.png, 6.png и 3.png), и сложить их времена сна, то можно заметить, что оно составляет более 3 секунд. Каким-то образом, только потому, что изо-

бражение 0.png было запланировано раньше всех остальных, Python не собирается околачиваться и ждать до тех пор, пока оно не будет готово. Вместо этого он хватает следующее изображение, которое не находится в состоянии сна и готово к обработке. В конечном итоге он возвращается к 0.png, но только после выполнения некоторых дополнительных заданий. На самом деле ему нужно подождать окончания времени сна и еще 2 секунды, чтобы завершилось изображение 3.png. Чего точно не видно, так это того, что Python по-прежнему разрешает исполнение только одного потока за раз. По мере изменений в языке Python библиотека `asyncio` в конечном счете будет работать быстрее, но для этого необходимо удалить весьма особое значение, которое содержится в интерпретаторе Python. Прежде чем перейти к этой теме, давайте сначала разберемся в том, как работает многопоточность исполнения в операционной системе.

8.4 Многопоточность исполнения

Первые компьютеры работали очень процедурно. Ввод данных состоял из бабин с магнитной лентой или перфокарт, которые компьютер обрабатывал и выводил результаты на экран, бумагу или обратно на ленту. Системы разделения времени были изобретены для того, чтобы отойти от этой традиции и позволить нескольким людям пользоваться системой одновременно. Это означало, что очень мощный компьютер мог использоваться несколькими людьми и множеством приложений одновременно, что резко снизило затраты в расчете на человека и в расчете на машину и открыло вычислительный мир к тому, что мы переживаем сегодня. Системы разделения времени сделали это за счет того, что позволили многочисленным процессам выглядеть как работающие одновременно, а затем, в более широком смысле, позволили приложениям разбиваться на более мелкие задания, именуемые *потоками исполнения*.

Есть два разных типа потоков исполнения, с которыми вы познакомитесь позже в этой главе, но, по сути, *поток исполнения* – это небольшой пакет информации о процессе. Поток включает в себя память и собственно подлежащие исполнению инструкции. Как только процессор завершил исполнение задания, над которым он работал, он хватает еще одно задание. После завершения работы с потоком процессор приостанавливает работу с ним по таймеру либо по сигналу из потока о том, что ему необходимо дождаться доступа к ресурсу (сети, файлу и т. д.). После приостановки потока процессор подхватывает еще один поток и начинает работу. При наличии всего одного процессора он все равно исполняет только одно задание за раз, однако с учетом потоков и скорости процессора все выглядит так, что компьютер делает много заданий одновременно.

Ситуация, когда приложение не может продолжить работу из-за того, что ему требуется ресурс, описывается как *блокировка* приложения. После блокировки приложение не может продолжать работу,

поэтому операционная система использует эту возможность для выполнения еще одного задания (см. рис. 8.4). Этот метод эффективен для системы в целом, но нередко является источником узких мест, которые обсуждались ранее.

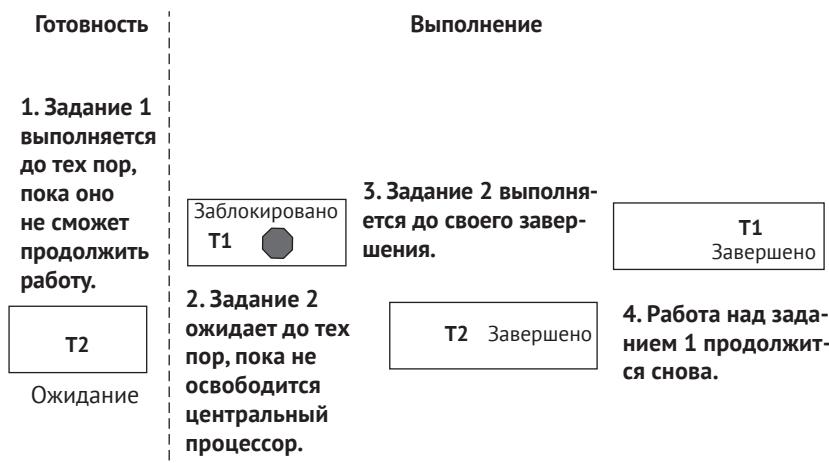


Рисунок 8.4 Центральный процессор будет пытаться обрабатывать задания, которые не были заблокированы

По мере эволюции компьютеров мы стали замечать появление дополнительных процессоров. Внезапно появились двухъядерные, четырехъядерные и даже восьмиядерные процессоры! Но означало ли это, что приложения стали быстрее? Да, но только в том случае, если они были написаны с учетом преимуществ этих ядер путем деления их работы на несколько заданий. Структура осталась прежней: на одном ядре может исполняться только один поток за раз. Раньше приложения можно было писать для запуска дополнительных потоков, чтобы распределять их работу. Например, одна часть приложения могла читать данные, тогда как другая – обрабатывать эти данные. Такое деление работы в приложении называется *конкурентностью*. Конкурентность нередко путают с параллелизмом, но это не одно и то же.

Параллельные системы означают, что система может исполнять несколько потоков одновременно. Если на машине исполняется четыре потока и есть четыре ядра, то работа действительно ведется параллельно. Однако во многих системах не так много процессоров, как потоков, поэтому приложения будут исполняться конкурентно. Следовательно, конкурентность – это наличие нескольких исполнений в одном приложении, но не обязательно в одно и то же время. Например, почтовое приложение может показывать сообщение, конкурентно получая новые сообщения с сервера.

Вот еще один пример: допустим, у одного из авторов книги есть машина для уплотнения консервных банок. Банки загружаются с одного конца, а уплотненные банки выходят с другого. Скорость уплотнения

банок можно измерить и получить представление об эффективности машины. После загрузки банок в приемный бункер скорость уплотнения была измерена, и получилась одна банка в секунду. Пометив банки пронумерованными наклейками, можно увидеть порядок уплотнения банок. Банки выходят в произвольном порядке.

На следующий день поступает обновленная версия машины, которая обещает более быстрое уплотнение. Теперь после измерения получается скорость уплотнения две банки в секунду. Решив нарушить условие гарантии, мы открываем обе машины. Внутри первой машины мы видим один молоток, который уплотняет банки с помощью воронки, а во второй машине – два молотка для уплотнения банок. Это решение выглядит довольно очевидным. Банки в этом примере представляют собой конкурентные процессы, которые ожидают своего «уплотнения», но машина может уплотнять только столько банок, сколько у нее есть молотков, как показано на рис. 8.5.

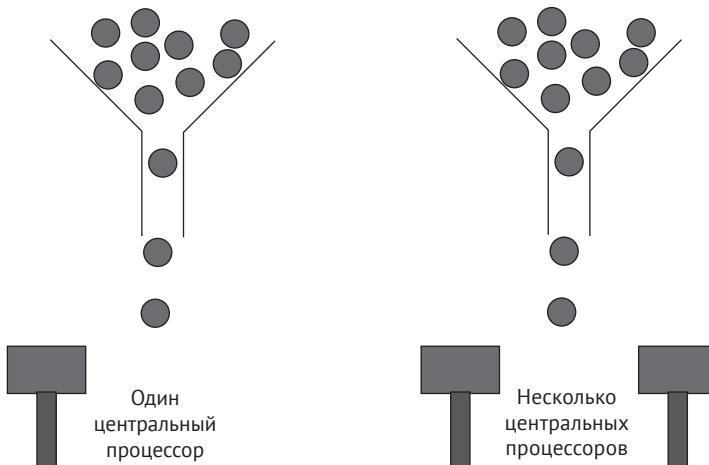


Рисунок 8.5 В данном случае известно количество машин-уплотнителей банок и нет гарантии порядка следования. Добавление большего количества машин-уплотнителей увеличивает пропускную способность

Указанный тип конкурентности на уровне операционной системы управляется потоками POSIX, или p-потоками (англ. pthreads). Эти потоки управляются операционной системой. В качестве альтернативы потоки могут создаваться языком или средой исполнения. Они называются *зелеными потоками*. По мере того как главное приложение выполняется в одном потоке, оно поддерживает свой собственный набор внутренних потоков, которыми оно управляет. Данный подход распространен в интерпретируемых языках и виртуальных машинах.

Поскольку Python является интерпретируемым языком, при использовании пакета `asyncio` он управляет всеми своими потоками. Python должен знать, когда часть службы невозможно выполнять из-за того, что она ожидает ресурс. В примере программы мы обращались к математическим библиотекам и инструментам отрисовки графиков. Оба этих за-

дания нередко блокируются из-за нехватки ресурсов или вызываются вне главного исходного кода на Python, что делает их отличными кандидатами для асинхронной системы и многопоточных вычислений. Для этого понадобится не просто конкурентность, а параллелизм. Давайте конвертируем исходный код, чтобы использовать потоки Python, и посмотрим, получится ли добиться каких-либо улучшений по сравнению с асинхронным кодом. Для начала нужно удалить состояния сна из предыдущего раздела и добавить следующий ниже исходный код.

Листинг 8.7 main.py: добавление потоков исполнения

```
from concurrent.futures import ThreadPoolExecutor
import asyncio

def mandelbrot_func:
    ...

executor = ThreadPoolExecutor(max_workers=4)

async def mandelbrot(size: int, path: str,
                      range_x0: float, range_y0: float,
                      range_x1: float, range_y1: float):
    return executor.submit(mandelbrot_func,
                           size,
                           path,
                           range_x0, range_y0,
                           range_x1, range_y1)

async def main():
    await asyncio.gather(*[
        mandelbrot(1000, f"purg{i}.png", -5.0, -2.12, -2.5, 1.12)
        for i in range(0,8)
    ])
```

Теперь каждое вычисление ставится в отдельный поток операционной системы, созданный интерпретатором Python. При выполнении программы улучшения по сравнению с асинхронной реализацией практически не заметны. Почему? Если она выполняется в трех дополнительных потоках, то можно было бы ожидать, что она будет выполняться примерно в четыре раза быстрее, при условии что на компьютере есть по меньшей мере четыре ядра. Однако, по изначальной идее, это не так. Python – это один-единственный процесс, и поэтому у него есть ограничения. Для того чтобы увидеть какие-либо улучшения, нужно понять, что происходит в Python, а затем посмотреть, как Rust может оказать помощь.

8.5 Глобальная блокировка

Давайте вернемся к обсуждению темы потоков исполнения. Из предыдущего раздела известно, что существует два типа потоков исполнения, один из которых поддерживается операционной системой,

а другой – средой исполнения. Python создает свои потоки исполнения и управляет ими через библиотеку `asyncio` или через системные потоки с помощью класса `ThreadPoolExecutor`. Выполнение интерпретатора Python на нескольких потоках может вызывать странные проблемы, и поэтому в 2003 году создатель языка Python ввел глобальную блокировку (Global Interpreter Lock, аббр. GIL, дословно «глобальная блокировка со стороны интерпретатора»). Этот инструмент, несмотря на свою простоту, имеет масштабные последствия для конкурентных программ. Его простота позволяет однопоточным Python'овским приложениям работать быстро, а конкурентным приложениям – безопасно. Глобальная блокировка позволяет исполнять только один поток, в то время как все остальные спят и ждут входные или выходные ресурсы.

Независимо от того, насколько умело предпринимаются попытки распределить работу по потокам в Python, глобальная блокировка не разрешает выполнять работу параллельно, поскольку Python не может гарантировать, что код будет иметь защищенный доступ к памяти, наряду с другими проблемами, возникающими при параллельном программировании с коллективным пространством памяти (см. рис. 8.6). Если бы несколько потоков могли исполняться одновременно, то можно было бы наблюдать множественный доступ к одним и тем же ячейкам в памяти, что приводило бы к различным проблемам с памятью.

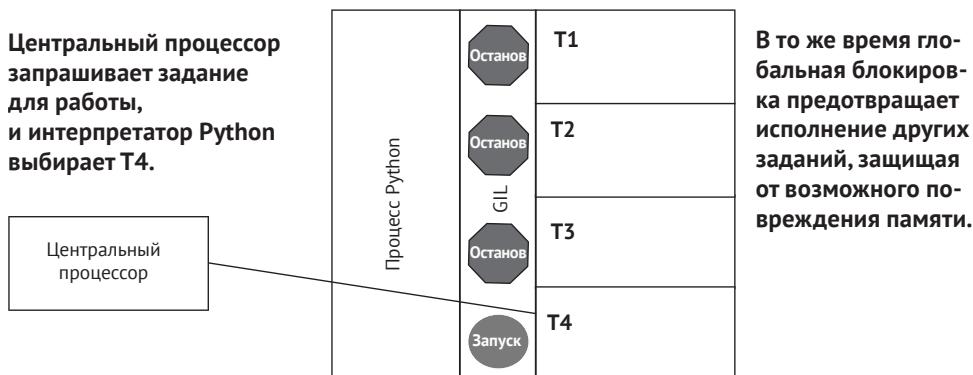


Рисунок 8.6 GIL – это глобальная блокировка, которую интерпретатор выдает, чтобы разрешить исполнение заданий

Как работает глобальная блокировка (GIL)? Глобальное значение внутри интерпретатора Python – это *мьютекс*, или замок на заданный ресурс, который предотвращает одновременный доступ к нему нескольких потоков исполнения. Представьте этот замок как пропуск в зал. Только один студент может бродить по залу в данный момент времени. Наличие ключа позволяет потоку внутри Python обращаться к ресурсам, доступным для интерпретатора, но запрещает доступ к этому ресурсу любому другому потоку. Затем Python может продолжить работу только после снятия замка. Далее он может передать ключ другому процессу, если тот готов. Передача глобальной блоки-

ровки другому процессу позволяет тому исполняться. Подобно мьютексу, глобальная блокировка защищает от одновременного доступа двух процессов к разделу памяти. Отсутствие этого ключа позволило бы двум процессам получать доступ к одним и тем же значениям, что может приводить к различным проблемам. В нашем примере с уплотнителем консервных банок можно представить, что есть две руки, которые могут ударять по банкам, но только один молоток. Прежде чем ударять, нужно схватить молоток.

Что требуется, так это возможность выполнять параллельные потоки в обход глобальной блокировки. Для того чтобы это сделать, нужно обеспечить защищенность памяти внутри Python. А для этого нужен модуль, который может обрабатывать параллельные потоки. Это можно сделать с помощью языков С и С++, но хотелось бы выбрать более защищенный путь, используя язык Rust.

8.6 Библиотека PyO3

На протяжении всей этой книги вы наблюдали способность языка Rust присоединяться к приложениям, написанным на других языках, и постепенно устранять проблемы, наделяя эти приложения возможностью использовать всю защищенность и скорость, предоставляемые языком Rust, без необходимости полной перестройки системы. В главе 6 была применена библиотека под названием PyO3, которая предоставила необходимые привязки языка Rust, чтобы выполнять Python'овский исходный код внутри приложения на языке Rust с целью улучшения системы. Теперь будет сделано обратное: исходный код на языке Rust будет выполнен внутри Python. Зачем это делать? По причине сложности переработки всей системы, о чем говорилось в начале главы. Нередко коллективы разработчиков собираются вместе или создают группу инженеров, хорошо владеющих одним конкретным языком, либо экосистему, поддерживающую тот или иной язык программирования. Тем не менее когда мы сталкиваемся с необходимостью масштабирования и выявляем часть кода как узкое место, ее следует перерабатывать. На данный момент исходный код на Python был переработан в части производительности, насколько это позволяет Python из-за глобальной блокировки. Библиотека PyO3 предоставляет инструменты, необходимые для повышения производительности приложения, используя защищенность и скорость языка Rust в обход глобальной блокировки. Язык Rust делает это, предоставляя способность снимать глобальную блокировку, давая возможность другим потокам исполняться. Ранее был представлен пул потоков в Python, но вы увидели, что создание потоков не сильно помогает сокращать общее время исполнения. Python не позволяет отключать глобальную блокировку непосредственно в исходном коде, но, как вы увидите, эту блокировку можно обходить с помощью языка Rust.

Переработка исходного кода – чрезвычайно мощный инструмент. Подумайте об эволюции многих программ. О переработке уже гово-

рилось во многих контекстах, но, как это часто бывает, легче сказать, чем сделать, особенно в случае с такими языками, как Python, которые превосходны в построении прототипов, но плохо масштабируются. Ruby – еще один пример скриптового языка, который легко осваивать и использовать, но имеет трудности с масштабированием. Эта проблема связана с природой самих языков. Разница между этими простыми в освоении языками и такими системными, как C, C++ и даже Rust, всегда была в стремлении повышать производительность и облегчать работу. Несмотря на то что язык Rust позиционируется как замена языкам C и C++, на нем все же не так-то просто писать, как на Python.

Тем не менее если у вас есть предприятие, которое вы хотите развивать, то, как правило, не стоит выбрасывать все, что вы сделали, чтобы переписать на каком-то другом языке. Использование другого языка приводит к дополнительным затратам на персонал, а также расширяет область знаний, необходимых для полного понимания системы. Как вы убедились на примере языков C и C++, замена части исходного кода на язык Rust может немного облегчать этот переход.

Наберите команду `maturin new`, чтобы начать проект, перейдите в этот каталог и наберите команду `cargo add image num-complex`. Это все, что нужно для начала. Откройте файл `src/lib.rs` и добавьте функцию Мандельброта.

Листинг 8.8 lib.rs: первоначальная функция Мандельброта

```
use image::{Rgb, RgbImage};           ← Импортирует библиотеку image.
use num_complex::Complex64;          ← Импортирует библиотеку по работе с комплексными числами.
use pyo3::prelude::*;

use std::path::Path;

#[pyfunction]
fn mandelbrot_func(size: u32, p: &str, range_x0: f64,           ← Добавляет макрокоманду для
                    range_y0: f64, range_x1: f64, range_y1: f64) {      ← функции, чтобы экспортить в Python.
    let mut img = RgbImage::new(size, size);
    let size_f64 = size as f64;
    let x_range = (range_x1 - range_x0).abs();
    let x_offset = x_range / 2.0;

    let y_range = (range_y1 - range_y0).abs();
    let y_offset = y_range / 2.0;
    let path = Path::new(p);
    for px in 0..size {
        for py in 0..size {           ← Выполняет итерации по пикселям,
            let x0 = px as f64 / size_f64 * x_range - x_offset;   ← чтобы вставлять их в изображение.
            let y0 = py as f64 / size_f64 * y_range - y_offset;
            let c = Complex64::new(x0, y0);

            let mut i = 0u8;
```

```

let mut z = Complex64::new(0.0, 0.0);
while i < 255 {
    z = (z * z) + c;
    if z.norm() > 4.0 {
        break;
    }
    i += 1;
}
img.put_pixel(px, py, Rgb([i, i, i])); ← Помещает
                                            пиксели в изо-
                                            бражение.

img.save(path).unwrap(); ← Сохраняет изображение
                           в файловой системе.

}

```

Макрокоманда `pymodule` вверху функции позволяет вызывать этот метод из Python. PyO3 упакует ее за вас, после того как будет создан модуль, в котором она будет находиться. Для этого будет добавлено еще немного исходного кода.

Листинг 8.9 lib.rs: создание модуля Python

```

#[pymodule] ← Создает модуль
for Python.
fn mandelbrot(_py: Python, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(mandelbrot_func, m)?); ← Добавляет функцию
    Ok(())
}

```

Для того чтобы это скомпилировать, будем снова полагаться на инструмент `maturin`. Наберите команду `maturin development`, чтобы скомпилировать библиотеку, и добавьте ее в свою среду. Далее скопируйте `main.py` из предыдущего листинга и внесите несколько изменений, чтобы вызывать новый модуль.

Листинг 8.10 main.py: импорт нового модуля

```

from mandelbrot import mandelbrot_func ← Импортирует функцию
import asyncio
# Удалить Python'овскую реализацию алгоритма Мандельброта

async def mandelbrot(size: int, path: str,
                      range_x0: float, range_y0: float,
                      range_x1: float, range_y1: float):
    return executor.submit(mandelbrot_func, size, path,
                           range_x0, range_y0,
                           range_x1, range_y1) ← Вызывает
                                         функцию
                                         Rust.

```

При выполнении этого исходного кода с использованием метода хронометража обнаруживается, что, просто используя Rust, время вычисления сократилось с 25 секунд до 23 секунд! Все еще не идеально, потому что по-прежнему действует глобальная блокировка. Для более радикального сокращения времени нужен пакет PyO3, чтобы сообщать Python о том, что источнику можно доверять – мы в безопасности. Для этого давайте создадим одну дополнительную функцию, которую будем экспортить.

Листинг 8.11 lib.rs: добавление потокобезопасной функции

```
#[pyfunction]
fn mandelbrot_fast(
    py: Python<'_>,
    size: u32,
    path: &str,
    range_x0: f64,
    range_y0: f64,
    range_x1: f64,
    range_y1: f64,
) {
    py.allow_threads(|| mandelbrot_func(size, path, range_x0, range_y0,
    range_x1, range_y1))
}

#[pymodule]
fn mandelbrot(_py: Python, m: &PyModule) -> PyResult<()> {
    ...
    m.add_function(wrap_pyfunction!(mandelbrot_fast, m)?);
}

ok(())
}
```

← Сообщает Python о том, что для этой функции не нужна глобальная блокировка.

Затем нужно создать несколько потоков, в которых будут выполняться вычисления на Python. Ниже импортируется исполнитель пула потоков. Здесь будет использовано четыре, чтобы наглядно увидеть, какого рода улучшения можно получить. Этот метод также работает для другой созданной функции, но вы обнаружите, что время исполнения будет таким же, так как глобальная блокировка не отключена. Однако выполнение быстрой функции отключает блокировку и позволяет задействовать истинную конкурентность.

Листинг 8.12 main.py: использование исполнителя потока

```
return executor.submit(mandelbrot_fast, size, path,
                      range_x0, range_y0,
                      range_x1, range_y1) ←
...
...
```

Отправляет функцию и параметры в поток для будущего исполнения.

Когда она завершит работу, общее время составит молниеностные 6 секунд. Это невероятное улучшение по сравнению с первоначальными 46 секундами для чистой версии на языке Python и 23 секундами для реализации на языке Rust без отключения глобальной блокировки. Вдобавок по ходу была выполнена итеративная миграция исходного кода, чтобы включить язык Rust в существующее приложение на Python, что позволило увеличить производительность почти в 5 раз.

В рамках переработки исходного кода необходимо выявлять аспекты системы, которые можно улучшить, не затрагивая систему в целом. Написанное выше приложение, очевидно, не отличается высоким качеством, но оно подчеркивает тот факт, что в таких языках, как Python, часто отдается предпочтение гибкости и возможности строить прототипы. Но чем больше требований предъявляется к системам, тем сложнее они становятся. Описанную выше переработку можно было бы расширить до полной замены изначального языка на язык Rust, чтобы полностью обойти ограничения языка Python. Однако эту библиотеку можно использовать и в других местах. Либо, возможно, отсутствует поддержка в написании всей системы целиком на Rust. В этой главе была обследована тема переработки систем, чтобы их масштабировать от прототипа до продукта. Хотя язык Python блестит во всей красе при прототипировании, было обнаружено, что язык Rust может делать это лучше. Выявление узких мест в исходном коде на Python поможет вам и вашему коллективу определять, насколько язык Rust является решением проблем со скоростью.

Следует помнить, что переработки исходного кода – это процесс, а не конечная цель, и поэтому он никогда не заканчивается. Язык Rust позволяет постепенно предпринимать эти небольшие шаги, чтобы повышать видимость изменений и переходить к решению, которое подходит для проекта лучше всего.

Краткий итог

- Язык Python отлично подходит для прототипирования, но он страдает от проблем с производительностью.
- Для повышения производительности используется масштабирование путем добавления дополнительного оборудования или расширения существующего аппаратного обеспечения.
- Масштабирование нуждается во внесении изменений в исходный код, чтобы пользоваться преимуществами этих изменений.
- Возможности языка Python по выполнению конкурентных процессов ограничены глобальной блокировкой.

- Язык Rust способен обходить эту блокировку, повышая общую производительность из-за присущей ему защищенности памяти.
- Используя язык Rust и библиотеку RuO3, можно обходить принятую в Python глобальную блокировку, чтобы разблокировать конкурентные процессы.
- Переработка Python'овских приложений с целью создания шаблонов конкурентности с защищенным доступом к памяти, используя язык Rust, сокращает задержку и повышает производительность.



WebAssembly для переработки исходного кода JavaScript

Эта глава охватывает следующие ниже темы:

- написание библиотеки Rust для использования в JavaScript;
- интеграция WebAssembly в существующий проект и компонент на языке JavaScript;
- написание веб-компонента полностью на языке Rust и его импорт в существующий проект.

Отыскание единого языка, с помощью которого можно было бы разрабатывать все части приложения, было целью многих создателей языков программирования. «Напиши один раз и запускай в любом месте» – таков был девиз языка Java, потому что в то время казалось, что коль скоро система может выполнять виртуальную машину Java, приложение будет работать и на ней. Очевидно, что у этого подхода были свои ограничения, но, по сути, именно он делает Java такой популярной платформой и по сей день. Идея кросплатформенного программного обеспечения не нова; на самом

деле целью ранних компиляторов было наделение программистов способностью писать приложение один раз и компилировать его для работы на других машинах.

Как вы убедились, язык Rust работает по той же схеме. Вместо того чтобы работать как Java, то есть использовать виртуальную машину для выполнения приложения, в Rust используются другие цели компиляции. Вдобавок примеры, которые были рассмотрены до сих пор, в какой-то степени опирались на интеграцию языка Rust с языком С в части импорта библиотек. В этой главе будет обследован новый подход «Напиши один раз и запускай в любом месте», но вместо написания Java (вздохнем с облегчением) вы будете работать с технологией, которая была создана специально для Всемирной паутины.

9.1 Что такое WebAssembly?

В 2018 году Консорциум Всемирной паутины (World Wide Web Consortium, аббр. W3C) опубликовал спецификацию, которая позволила нацеливать компиляцию на специальную разновидность байт-кода, который мог исполняться в браузере. Идея заключается в том, чтобы такие компилируемые языки, как C++, Go и Rust, могли нацеливать свои компиляторы не на процессоры AMD или Intel, а на написание двоичных файлов в байт-коде Web Assembly (Wasm). Байт-код Wasm нацелен на системный интерфейс с WebAssembly (WebAssembly System Interface, аббр. WASI), который, по сути, представляет собой среду для исполнения указанного байт-кода.

Сейчас мы наблюдаем, как вокруг байт-кода Wasm развивается несколько технологий, а также несколько довольно интересных проектов. Разработчики обнаруживают, что могут помещать в веб-браузер практически все, что угодно, включая целевые операционные системы! Wasm используется для выполнения кода в облачных работниках, а разработчики некоторых библиотек JavaScript заняты переработкой частей своего кода под использование Wasm. Загрузка Wasm нуждается в том, чтобы JavaScript подтягивал библиотеку и ее инициализировал, как показано на рис. 9.1.

Итак, почему вас как разработчика на языке Rust это должно волновать? Скажем так, несмотря на то что крупная часть исходного кода системного уровня написана на языке Java или С, наиболее часто используемыми языками программирования веб-приложений являются языки на основе JavaScript, которые работают в веб-браузере (рис. 9.2). Как уже упоминалось ранее, Wasm разрабатывался как универсальный двоичный код, также предназначенный для исполнения в браузере. Благодаря этому появляется возможность писать исходный код на языке Rust, способный взаимодействовать с исходным кодом на языке JavaScript либо заменять его части, позволяя разработчикам перерабатывать его части на языке Rust.

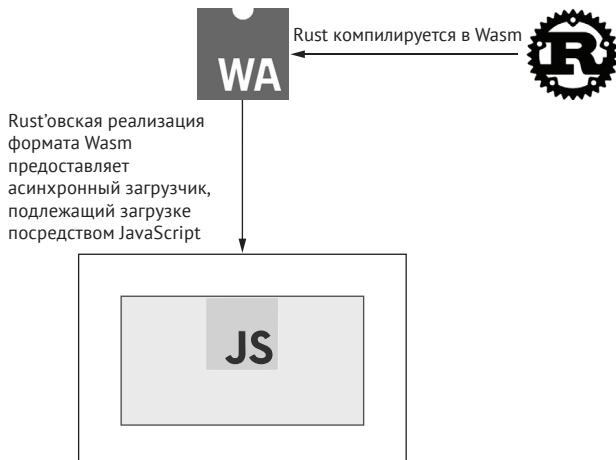


Рисунок 9.1 Wasm, загружаемый во фронтальную часть веб-приложения, написанную на языке JavaScript

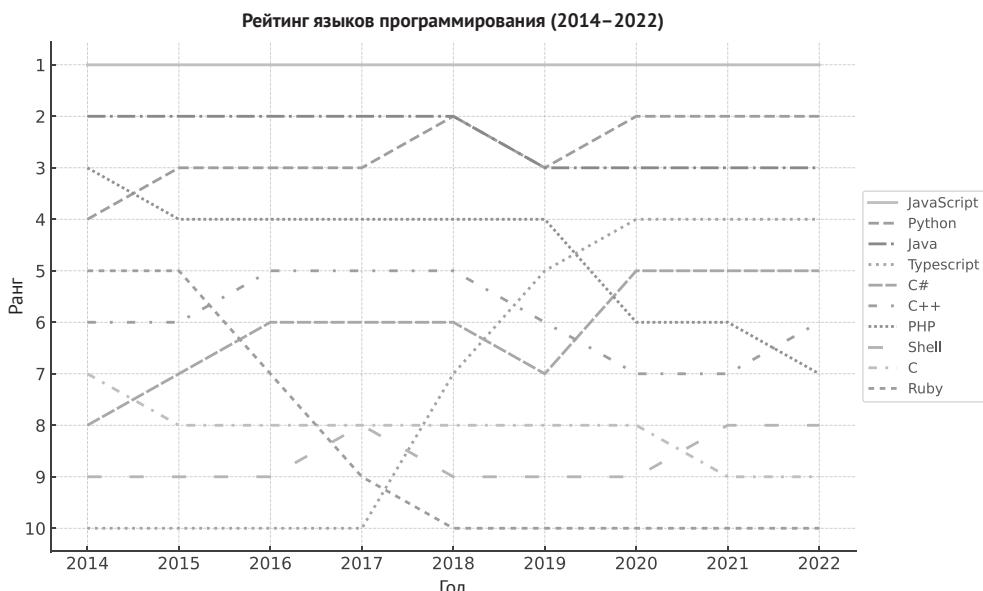


Рисунок 9.2 Обзор наиболее часто используемых языков в 2022 году от веб-сайта Octoverse веб-платформы Github

У этой взаимосвязи Rust/Wasm есть и обратная сторона: поскольку Wasm – это универсальный двоичный код, который предположительно должен работать где угодно, библиотеку Wasm можно выполнять в Rust. Следовательно, можно переработать старый исходный код, импортировав его части в Rust через Wasm. Сначала вы увидите, как писать функцию Rust и импортировать ее в JavaScript через Wasm. Затем, в следующей главе, код, который был скомпилирован в Wasm, будет исполнен в приложении Rust.

9.2 Перенос из JavaScript в Rust

Прежде чем углубиться в сам исходный код, важно разобраться в мире языка JavaScript и в том, чем его переработка отличается от того, что вы делали до сих пор. До сих пор все внимание было сосредоточено на коде, который выполняется не в веб-браузере, а в терминале. Языки C++ и Python на самом деле представляют собой код на языке C, тогда как JavaScript – это самостоятельный скриптовый язык, созданный для веб-браузеров. Он присутствует на 98 % веб-сайтов и является важным инструментом Всемирной паутины. Язык JavaScript изначально был разработан еще в 1995 году и с течением времени постепенно менялся. Тем не менее положенное в его основу предназначение – быть «языком веб-браузера» – не менялось вплоть до 2009 года, когда появилась среда исполнения Node.js. С тех пор границы между фронтальной и серверной разработкой на JavaScript по-настоящему начали размываться. Были написаны дополнительные инструменты, помогающие сделать JavaScript более надежным, такие как TypeScript, который добавил в JavaScript типы, аналогичные тем, что есть в языке Rust.

Таким образом, здесь представлен еще один широко распространенный язык, который медленно эволюционировал (а в некоторых случаях и деэволюционировал) с течением времени и находится в одном ряду с языками C++ и Python в мире исходного кода, который может стать неуправляемым и только выиграет от переработки. Разница в том, что вместо того, чтобы фокусироваться на серверном коде веб-приложения, все внимание будет сосредоточено на фронтальной его части, и будет переработан JavaScript, чтобы обеспечить защищенность памяти, скорость и систему типов, которые делают Rust таким надежным.

Как узнать, когда следует перерабатывать JavaScript в Rust? Какие варианты использования следует искать? Ответ будет таким же, как и в случае с решениями, которые вы бы приняли при выполнении миграции с Python на Rust или с C++ на Rust: защищенность и скорость. Разница лишь в том, что теперь нужно будет начать думать о браузере, а не о терминале (хотя Wasm можно использовать и в среде исполнения Node.js). JavaScript не является типобезопасным языком и подвержен ошибкам во время исполнения. Кроме того, он работает медленнее, чем скомпилированные программы. Wasm и Rust также более безопасны, чем JavaScript, в том, что касается управления памятью приложения. Поэтому если вы ищете какие-либо из этих улучшений или если есть внутренняя логика, которую можно перенести во фронтальную часть веб-приложения, чтобы уменьшить нагрузку на серверную часть, то рекомендуется рассмотреть возможность переработки.

При разработке библиотеки Wasm для веб-сайта обычно используется тот же процесс, что и при разработке пользовательского интерфейса, но вместо JavaScript и HTML будет использоваться язык Rust, скомпилированный в Wasm с HTML. В будущем вы сможете разрабатывать целые компоненты на языке Rust.

9.3 Язык Rust в браузере

Большинство современных веб-компонентов прорисовываются на стороне клиента с использованием данных, передаваемых по протоколу HTTP. Чаще всего данные передаются путем отправки сообщений в формате JSON с использованием протокола REST. Однако это не единственный способ. В 1997 году была создана модель данных под названием RDF, которая помогает систематизировать метаданные вокруг произвольных объектов. Данная модель легла в основу новостных лент (каналов) RSS (RDF Site Summary, дословно «Резюме веб-сайта на основе RDF»), предоставляющих пассивный способ уведомления других систем об обновлениях веб-сайта. Для агрегирования этих новостных лент и их вывода на экраны пользователей для чтения или сохранения на потом используются различные инструменты.

В этой главе будет разработан инструмент, который использует новостную ленту RSS (в формате RDF) и создает компонент для вывода списка статей, недавно опубликованных в хранилище научных работ с открытым доступом arXiv, и предоставления подробной информации о них. Сначала будет написана функция извлечения статей на основе поискового термина, предоставляющая ссылки на саму статью. После написания функции она будет экспортирована, чтобы ее использовать в JavaScript, и размещена в качестве веб-компонента. Для начала давайте рассмотрим структурирование данных и извлечение результатов поиска.

9.3.1 Запрос данных

Сначала создадим новое приложение Rust, выполнив следующие ниже команды.

Листинг 9.1 Создание нового проекта

```
cargo new papers --lib
cd papers
```

Откройте `Cargo.toml` и добавьте следующие ниже библиотеки.

Листинг 9.2 Cargo.toml: зависимости для библиотеки

```
[package]
name = "papers"
version = "0.1.0"
edition = "2021"

[lib]
crate-type = ["cdylib"]
# See more keys and their definitions
➥at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
reqwest = { version = "0.11", features = ["json"] }
```

Request – это библиотека для выполнения вызовов по протоколу HTTP.

```

serde = { version = "1.0", features = ["derive"] }
serde-wasm-bindgen = "0.5.0"
serde-xml-rs = "0.6.0"
wasm-bindgen = "0.2.87"

[dev-dependencies]
tokio-test = "*"

```

Конвертирует исходный код на языке Rust в Wasm.

Библиотека синтаксико-структурного разбора разметки XML.

Используется в асинхронном тестировании.

Serde – это главная библиотека кодирования/декодирования для различных вызовов.

Wasm bindgen позволяет конвертировать объекты JSON из JsValue в структуру.

Теперь, когда есть проект, целесообразно определить структуры. Для этого сначала надо посмотреть, как выглядит реальная новостная лента.

Листинг 9.3 arXiv: пример сообщения от вызываемого сервиса

```

<feed xmlns="http://www.w3.org/2005/Atom">
    <link href="http://arxiv.org/api/query?search_query%3Dall%3Atype"
        < lineararrow /> rel="self" type="application/atom+xml"/>
    <title type="html">
        ArXiv Query: search_query=all:type
    </title>
    <id>http://arxiv.org/api/MPA5fUXeKVs0FQAFa0fw4Eh7V44</id>
    <updated>2023-06-13T00:00:00-04:00</updated>
    <opensearch:totalResults
        xmlns:opensearch="http://a9.com/-/spec/opensearch/1.1/">
        229748
    </opensearch:totalResults>
    <opensearch:startIndex
        xmlns:opensearch="http://a9.com/-/spec/opensearch/1.1/">
        0
    </opensearch:startIndex>
    <opensearch:itemsPerPage
        xmlns:opensearch="http://a9.com/-/spec/opensearch/1.1/">
        10
    </opensearch:itemsPerPage>
    <entry>
        <id>http://arxiv.org/abs/cs/0507037v1</id>
        <updated>2005-07-14T08:58:31Z</updated>
        <published>2005-07-14T08:58:31Z</published>
        <title>Type Inference for Guarded Recursive Data Types</title>
        <summary> ... </summary>
        <author>
            <name>Peter J. Stuckey</name>
        </author>
        <author>
            <name>Martin Sulzmann</name>
        </author>
        <link href="http://arxiv.org/abs/cs/0507037v1"
            rel="alternate"
            type="text/html"/>

```

```

<link title="pdf" href="http://arxiv.org/pdf/cs/0507037v1"
      rel="related"
      type="application/pdf"/>
<arxiv:primary_category
    xmlns:arxiv="http://arxiv.org/schemas/atom"
    term="cs.PL"
    scheme="http://arxiv.org/schemas/atom"/>
<category term="cs.PL" scheme="http://arxiv.org/schemas/atom"/>
    <category term="cs.LO" scheme="http://arxiv.org/schemas/atom"/>
</entry>
</feed>
```

Из этого примера видно, что корнем файла является тег `feed`, а каждый результат – это `entry`. Из записи `entry` требуется получить информацию, состоящую из списка авторов, идентификатора, названия статьи и общих сведений о том, когда запись была обновлена и когда была опубликована. Учитывая эти поля, можно определить следующие ниже структуры.

Листинг 9.4 lib.rs: определение базовых структур для поиска

```

#[derive(Default, Debug, Clone, PartialEq, Serialize, Deserialize)]
pub struct Feed {
    pub entry: Vec<Entry>, ← Список записей, подлежащих выводу на экран
} и обертыванию в родительскую структуру, ана-
логичную приведенному выше файлу XML.

#[derive(Default, Debug, Clone, PartialEq, Serialize, Deserialize)]
pub struct Entry {
    pub id: String,
    pub updated: String,
    pub published: String,
    pub title: String,
    pub summary: String,
    pub author: Vec<Author>,
}

#[derive(Default, Debug, Clone, PartialEq, Serialize, Deserialize)]
struct Author {
    pub name: String,
}
```

С учетом этой структуры затем можно создать функцию, которая извлекает результаты поиска с разбивкой на страницы (под разбивкой на страницы подразумевается разделение по размеру и начальному местоположению). Для этого будет использована библиотека `reqwest` (поддерживаемая Wasm), которая будет извлекать результаты. Полученные результаты будут конвертированы из XML в JSON для компонента. Используя их API по работе с RDF, можно передавать поисковые запросы, а также данные разбивки на страницы (начальный и максимальный результаты). Вся эта функциональность будет добавлена в библиотеку. Прямо сейчас давайте напишем функцию.

Листинг 9.5 lib.rs: доставка и разбор результатов поиска статей

```
async fn search(term: String, page: isize, max_results: isize) ->
    Result<Feed, reqwest::Error> {
    let http_response = reqwest::get(format!("http://export.arxiv.org/api/query?search_query=all:{}&start={}&max_results={}", term, page * max_results, max_results)).await?;
    let b = http_response.text().await?;
    let feed: Feed = serde_xml_rs::from_str(b.as_str()).unwrap();
    return Ok(feed)
}
```

Конвертирует
в структуру Feed.

Вызывает конечную точку экспорта с заданной темой, страницей и количеством статей.

Сохраняет текстовый ответ.

Наконец, можно написать тест, чтобы удостовериться, что все работает, как положено.

Листинг 9.6 lib.rs: добавление модульных тестов для поиска

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

    macro_rules! aw {
        ($e:expr) => {
            tokio_test::block_on($e)
        };
    }

    #[test]
    fn test_search() {
        let res = aw!(search("type".to_string(), 0, 10)).unwrap();
        assert_eq!(res.entry.len(), 10);
        print!("{:?}", res)
    }
}
```

Макрокоманда, допускающая блокировку внутри асинхронного теста.

Использует макрокоманду, чтобы блокировать вплоть до получения ответа и верификации результатов.

ПРИМЕЧАНИЕ. Блокировка – это ситуация, когда система ожидает до тех пор, пока не будет возвращен результат, в отличие от асинхронных вызовов, которые во время ожидания переключаются на другой процесс.

Это довольно простая функция, которую можно применять, чтобы пользоваться преимуществом асинхронных возможностей языка Rust и мощных библиотек синтаксико-структурного разбора. Данный метод будет положен в основу компонентов, которые будут разработаны в этой главе, и в инструментах, которые будут построены в следующей. Несмотря на простоту метода, его можно использовать самыми разными способами, что делает его идеальным для целей демонстрации возможностей переносимости формата Wasm.

9.3.2 Компиляция в Wasm

Теперь, когда есть функция, которая выполняет нужную функциональность поиска, можно взглянуть на то, как она выглядит в веб-браузере. Для этого надо ее скомпилировать в байт-код Wasm и применить функцию загрузки JavaScript. После того как будет определена функция, которую нужно экспорттировать, это делается довольно просто. Она определяется несколькими разными способами, но для того чтобы обеспечить меньший размер интерфейса, давайте ограничимся передачей объекта JSON, который сейчас определим ниже.

Листинг 9.7 lib.rs: определение объекта как объекта JSON

```
#[derive(Default, Debug, Clone, PartialEq, Serialize, Deserialize)]
pub struct Search {
    pub term: String,
    pub page: isize,
    pub limit: isize,
}
```

Далее нужно определить функцию, которую привязка Wasm может генерировать, чтобы передавать объект JSON. Для этого будет применена макрокоманда, заданная в библиотеке `wasm_bindgen`. Функции будет передано специальное значение `JsValue`, которое затем вернет аналогичный объект. Эта функция тоже будет асинхронной. Иными словами, она будет возвращать объект-обещание¹ JavaScript, который необходимо исполнить перед возвратом данных.

Листинг 9.8 lib.rs: создание функции поиска

```
#[wasm_bindgen]    ← Этот атрибут будет использоваться
pub async fn paper_search(val: JsValue) -> JsValue{           во время компиляции для создания
    let term: Search = serde_wasm_bindgen::from_value(val).unwrap(); ← функции Wasm, которая будет
    let resp = search(term.term, term.page, term.limit).await.unwrap(); ← выставлена в JavaScript.
    serde_wasm_bindgen::to_value(&resp).unwrap()    ← Десериализует
}                                ← объект JSON
                                    ← в структуру языка
                                    ← Rust.
                                    ← Кодирует ответ
                                    ← в JsValue.
                                    ← Вызывает функцию поиска и ожидает результатов.
```

Как вы видите, внутри этой функции значение `JsValue` конвертируется в структуру `Search`. Это делается специальной библиотекой `serde`. После получения результата из функции поиска значения перекодируются в JSON и возвращаются. Это все, что было нужно! Теперь с помощью следующей ниже команды можно выполнить компиляцию в Wasm.

¹ Объект-обещание представляет окончательное завершение (или отказ) асинхронной операции и ее результирующее значение. – Прим. перев.

Листинг 9.9 Консоль: сборка и компиляция в Wasm для веба

```
cargo install wasm-pack
wasm-pack build --target web
```

← Цель `--target web` предоставит необходимые файлы самозагрузки, чтобы загрузить файл Wasm в браузер и его использовать.

Если заглянуть в выходной каталог `pkg`, то можно увидеть, что была создана специальная готовая к использованию библиотека `npm`. В файле `papers.js` будет набор самозагружочного кода, который помогает загружать модуль Wasm. И если аналогичным образом открыть файл `papers.d.ts`, то можно увидеть ожидаемые типы и функции, экспортимые этим пакетом. Далее давайте удостоверимся, что эта функция работает в JavaScript.

9.3.3 Загрузка Wasm в браузер

Теперь, когда есть функция поиска, давайте посмотрим, как она работает в браузере. Прежде чем добавлять этот код в более сложный компонент JavaScript, сначала убедимся, что он работает с простым JavaScript. Для этого будет создана облегченная HTML-страница, на прямую загружен Wasm, снабженный элементом поиска, и выведено на экран содержимое в виде списка. Для этого давайте создадим простой файл `index.html`.

Листинг 9.10 index.html: вызов библиотеки Wasm из JavaScript

```
<!DOCTYPE html>
<html lang="en-US">
  <head>
    <meta charset="utf-8" />
    <title>Feed example</title>
  </head>
  <body>
    <div id="listContainer">
      <ul id="list"></ul>
    </div>
    <script type="module">
      import init, { paper_search } from "./pkg/papers.js";
      init().then(() => {
        var list = document.getElementById('list');
        paper_search({ "term": "type", "page": 0, "limit": 10 }).then( <-- Инициализирует
          (result)=>{                                         модуль и ожидает
            result.entry.forEach((r)=> {                   загрузки Wasm.
              var a = document.createElement('a');
              a.target = '_blank';
              a.href = r.id;
              a.innerText = r.title;
              var li = document.createElement('li')
              li.appendChild(a)
              list.appendChild(li)
            })
          }
        )
      )
    </script>
  </body>
</html>
```

После урегулирования извлекает элемент списка.

В случае успеха выполняет итерации по результатам и добавляет элементы в список.

Инициализирует модуль и ожидает загрузки Wasm.

Загружает JavaScript и файл Wasm.

Вызывает функцию поиска и ожидает результатов.

```

        })
    },
    (errgog)=>console.errgog(errgog)) <-- В противном случае
}); регистрирует ошиб-
</script> ку в журнале.
</body>
</html>

```

Как вы видите, для создания страницы используется традиционный JavaScript. При этом получилось обойтись без современных фреймворков, которые в настоящее время используются во многих приложениях для выполнения JavaScript, но исходный код представляет собой отличный пример того, как встраивать эту функцию в качестве обычной библиотеки JavaScript. Будем надеяться, что это заставит вас задуматься об используемых внутри надоедливых функциях JavaScript, которые можно было бы переписать на языке Rust и загружать в таком ключе. Подобного рода функции JavaScript используются практически везде, делая их первым кандидатом для переработки. Хотя эта функциональность характеризуется высокой переносимостью, она не всегда вписывается в более крупный проект JavaScript. Для этого можно воспользоваться современной библиотекой компонентов, такой как React.

9.4. Создание компонента библиотеки React

Компонентно-ориентированная разработка существует с момента зарождения разработки программного обеспечения в 1968 году. Ее концепция проста: разграничивать компетенции внутри программной системы путем сборки изолированных пакетов, служб, ресурсов или модулей, которые имеют схожие функции или данные. Сегодня многие языки, такие как JavaScript, имеют фреймворки или библиотеки, которые помогают в создании компонентов. Одной из самых популярных из них является библиотека React.

Библиотека React существует уже более десяти лет и изменила подход специалистов к разработке пользовательских интерфейсов. Она зарекомендовала себя как отличный инструмент для построения компонентов и широко распространена во всей Всемирной паутине. За последние пару лет стали популярны и другие библиотеки, такие как Vue.js, поэтому пример, который предстоит написать, может отличаться для одной из этих библиотек.

Для начала будет создано новое веб-приложение посредством инструмента под названием Vite. Vite – это один из многих современных фреймворков JavaScript, который предоставляет инструменты для самозагрузки веб-приложений. Он будет задействован для самозагрузки нового приложения JavaScript с использованием компонентной библиотеки React. За счет этого будет получен минимум частей, необходимых для экспериментов с Wasm. Прежде всего должна быть установлена специальная библиотека `npm`, что можно сделать, следуя

инструкциям по установке, приведенным в документации по `npm` по адресу: <https://mng.bz/eBMw>.

Давайте начнем с того, что откроем терминал в проекте `papers` и наберем на клавиатуре следующее.

Листинг 9.11 Консоль: создание нового приложения React

```
npm create vite@latest

Need to install the following packages:
  create-vite@latest
Ok to proceed? (y) y
✓ Project name: ... papers-list
✓ Select a framework: > React
✓ Select a variant: > JavaScript
```

Выполнение этой команды позволит создать базовое приложение. Прежде чем продолжить, нужно изменить способ создания Wasm. Прямо сейчас он сконфигурирован под сборку, используя флаг `web`, в результате чего предоставляется загрузчик, который необходимо вызвать для используемой библиотеки Wasm. Вместо этого будет применена опция `bundler`, в результате чего исходный код помещается в модуль, который можно легко импортировать и использовать в пакете JavaScript.

Поскольку язык JavaScript существует уже давно, имеются разные способы построения исходного кода на данном языке. Изначально исходный код строился путем загрузки нескольких скриптов через браузер, что требовало от каждой страницы отслеживать используемые ею библиотеки и методы их взаимодействия. В предыдущем примере это делалось с помощью тега `script`. На протяжении многих лет многие библиотеки писались в модульном формате, где библиотеки и написанный исходный код используются компилятороподобным инструментом, который собирает их в единый исполняемый скрипт. Этот компилятороподобный инструмент называется комплектатором `bundler`, так как его работа состоит в комплектации скриптов. При этом исходный код скорее трактуется им не как скрипт, а как библиотека. Таким образом, ввиду того, что исходный код желательно использовать как библиотеку внутри компонента, при компиляции Wasm будет использован флаг `bundler`.

Флаг `bundler` применяется следующим образом.

Листинг 9.12 Консоль: комплектация библиотеки в пакет

```
wasm-pack build --target bundler
cd pkg
npm link
cd ../papers-list
```

Далее нужно отредактировать файл `package.json`, чтобы добавить в проект библиотеку Wasm в качестве относительного импорта. Добавьте следующий ниже код в секцию `dependencies`.

Листинг 9.13 package.json: добавление локальной зависимости

```
"dependencies": {
    "papers": "file:../pkg",
    ...
}
```

Затем добавьте следующие ниже библиотеки и запустите установку.

Листинг 9.14 Консоль: компоновка и компиляция библиотеки Wasm

```
npm install vite-plugin-wasm vite-plugin-top-level-await --save-dev
npm link papers
npm install
```

Наконец, остается последний шаг настройки, после чего можно будет написать компонент. Откройте `vite.config.js` и добавьте необходимые модули Wasm.

Листинг 9.15 vite.config.js: конфигурирование приложения под использование Wasm

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'
import wasm from "vite-plugin-wasm";
import topLevelAwait from "vite-plugin-top-level-await";

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [
    react(),
    wasm(),
    topLevelAwait()
  ],
})
```

Теперь давайте создадим компонент. Сначала целесообразно создать компонент со статическими данными, чтобы иметь возможность с ним ознакомиться и убедиться, что он работает. Вдобавок он предоставляет трафарет, который легко обновляется с помощью переменных. Будет создан компонент под названием `List`. Итак, в папке `src` создайте новый файл под названием `List.jsx` и добавьте в него следующий ниже исходный код.

Листинг 9.16 List.jsx: создание компонента со статическими данными

```
import React, { useEffect, useState } from 'react'
const List = () => {
  const [entries, setEntries] = useState([{id:"abc", title:"title"}])
```

Управление состоянием списка статей.

```

const [page, setPage] = useState(0) <-- Управление состоянием
                                         счетчика страницы.

return (
  <>
  <ul>
    {entries?.map((v, i) => { <-- Просматривает список записей
      return <li key={i}>
        <a href={`${v.id}`} target='_blank'>{v.title}</a>
      </li>
    })} <-- и прорисовывает ссылку для
          каждой из них.

  </ul>
  <button onClick={() => setPage((page) => page + 1)}>More</button>
  </>
)
}

export default List;

```

Использует кнопку для увеличения счетчика страницы.

Теперь в окне терминала наберите `npm start dev` и откройте окно браузера по адресу порт/хост, указанному в терминале. Будем надеяться, что вы увидите прорисовку ссылки. Давайте добавим файл Wasm. Следует помнить, что приложение должно доставить и загрузить файл. Для этого нужно добавить инструкцию `import`, создающую будущее¹ JavaScript, которое должно разрешиться перед использованием библиотеки. Итак, за пределами компонента `List` нужно добавить инструкцию `import`.

Листинг 9.17 List.jsx: импорт библиотеки Wasm

```

import React, { useEffect, useState } from 'react'

const wasm = await import('papers')

```

Вы заметите наличие переменной `page`, значение которой увеличивается на единицу при нажатии кнопки `More`. При изменении этой переменной требуется, чтобы React обновлял состояние компонента на основе этого эффекта. Для этого будет создан перехватчик `useEffect`.

Листинг 9.18 List.jsx: использование Wasm для доставки документов

```

const List = () => {
  const [entries, setEntries] = useState([]) <-- Вначале создает
                                                пустой список.

  const [page, setPage] = useState(0)

  useEffect(() => { <-- Наблюдатель за обновлениями компонента,
    if(wasm){ <-- чтобы повторно прорисовывать данные при их изменении.

      wasm.paper_search({ "term": "type",
        <-- Вызывает функцию
        "page": page, "limit": 10 }).then( <-- поиска, передавая
          (result)=>setEntries(result.entry), <-- страницу и лимит.

        <-- Устанавливает записи из результата поиска.
      )
    }
  })
}

export default List;

```

¹ Представляет собой значение, которое будет доступно в какой-то момент в будущем, обычно как результат асинхронной операции. – Прим. перев.

```

        (еффог)=>console.еffог(еффог)) <--|
      } } [page]) <--| Наблюдает и обновляет
      ...           | при изменении переменной
}               | страницы.

```

Выводит на экран со-
общение об ошибке
в случае ее возникно-
вения.

Сохраните и понаблюдайте за перезагрузкой страницы. Сейчас вы увидите несколько статей. Если кликнуть по кнопке More (Показать еще), то можно будет увидеть обновленную страницу! Таким образом, после незначительной настройки исходный код на языке Rust был полностью интегрирован в приложение на языке JavaScript. Благодаря этому сотрудству языков Rust и JavaScript через посредничество Wasm появилось несколько инструментов, помогающих в создании компонентов, которые позволяют написать свой собственный компонент React на языке Rust. Давайте посмотрим, как это выглядит.

9.5 Веб-компоненты полностью на языке Rust

Yew – это библиотека, разработанная для создания веб-компонентов пользовательского интерфейса, которые компилируются в Wasm. Предназначение библиотеки Yew – задействовать все преимущества языка Rust в области защищенности в веб-приложениях. Большинство шаблонов разработки мигрировали от модели прорисовки на серверной стороне к модели на клиентской стороне, вследствие чего появился разрыв между серверным и фронтальным кодом, который большинство языков не в состоянии преодолеть, поскольку большая часть фронтального кода выполняется на языке JavaScript. С появлением Wasm это перестало быть насущной проблемой. Сейчас пишутся целые компонентные фреймворки, которые действуют подобно фреймворкам в React, но написаны на языке Rust.

Библиотека Yew поможет создать компонент, аналогичный тому, который был создан в React; первостепенное отличие заключается в том, как обрабатываются состояния и действия компонентов. Состояниями будут **Fetching**, **Success** и **Failure**, а действиями – **IncrementPage**, **SetFeedState** и **GetSearch**. Затем у компонентов Yew должно быть три метода: **create**, **update** и **view**. Методы **create** и **update** используются соответственно для установки начального состояния и изменения состояния, тогда как **view** использует это состояние для прорисовки компонента. Этот процесс происходит в рамках классической структуры модель–представление–контроллер, в которой модель содержит состояние, контроллер управляет действиями, а представление прорисовывается на основе состояния.

Сначала нужно добавить Yew в файл **Cargo.toml**.

Листинг 9.19 Cargo.toml: добавление компонентной библиотеки Yew

```
[dependencies]
...
yew = "0.19.0"
```

Давайте начнем с создания перечислений для действий и состояний.

Листинг 9.20 lib.rs: создание перечислений для различных состояний

```
use yew::prelude::*;

...
pub enum Msg {
    IncrementPage,
    SetFeedState(FetchState<Feed>),
    GetSearch(isize),
}

pub enum FetchState<T> {
    Fetching,
    Success(T),
    Failed(reqwest::Error),
}
```

Импортирует пакет Yew.

Определяет возможные типы сообщений.

Определяет различные состояния страницы.

Сам компонент должен содержать какое-то состояние. В данном случае это будет состояние доставки, `Fetch`, а также информация о том, на какой странице мы находимся в данный момент. Списковая структура, `List`, будет выглядеть следующим образом.

Листинг 9.21 lib.rs: создание структуры начального состояния

```
pub struct List {
    page: isize,
    feed: FetchState<Feed>,
}
```

Теперь нужно реализовать тип компонента для структуры `List`. Ниже будет определено два значения, которые будут использоваться для прорисовки компонента. Это `Messages` и `Properties`. `Messages` – это тип действий, которые могут происходить при обновлении, тогда как `Properties` могут быть значениями, которые будут отслеживаться библиотекой Yew на предмет обновлений. Будет предоставлена базовая структура `List`, вмещающая свойства базовых значений, которые предстоит использовать в компоненте. Тогда реализация компонента `Component` нуждается в реализации функций, помогающих компоненту выполнять прорисовку. В этом примере `Properties` не используются, но более подробная информация об их применении находится по адресу www.yew.rs. Вместо них будет использоватьсь эта базовая структура `List`, в которой есть лента `feed` и текущая страница `page`. Кроме того, необходимо реализовать три метода: `create`, `update` и `view`. Поэтому давайте создадим базовый скелет, а затем заполним его методами.

Листинг 9.22 lib.rs: очертание базового компонента

```

Задает Реализация компонента
свойствам библиотеки Yew.
отсутствие Задает тип используемого
значения. значения Message.

impl Component for List {
    type Message = Msg;
    type Properties = ();
    fn create(ctx: &Context<Self>) -> Self {
        fn update(&mut self, ctx: &Context<Self>, msg: Self::Message) -> bool {
            fn view(&self, ctx: &Context<Self>) -> Html {
                }
            }
        }
}

```

Давайте сначала разберемся с потоком данных в компоненте. Начнем с начального состояния, установленного методом `create`, который также начнет процесс поиска, когда страница будет равна 0. Это приведет к тому, что этап `View` будет прорисовывать в режиме `Fetching`, в котором на экран будет выводиться сообщение о загрузке. Любое внутреннее изменение состояния управляет методами обновления, которые затем будут инициировать изменения в представлении `View`. Представление может содержать кнопку, которая инициирует событие и обрабатывается обновлением. Высокоуровневую схему происходящего можно увидеть на рис. 9.3.

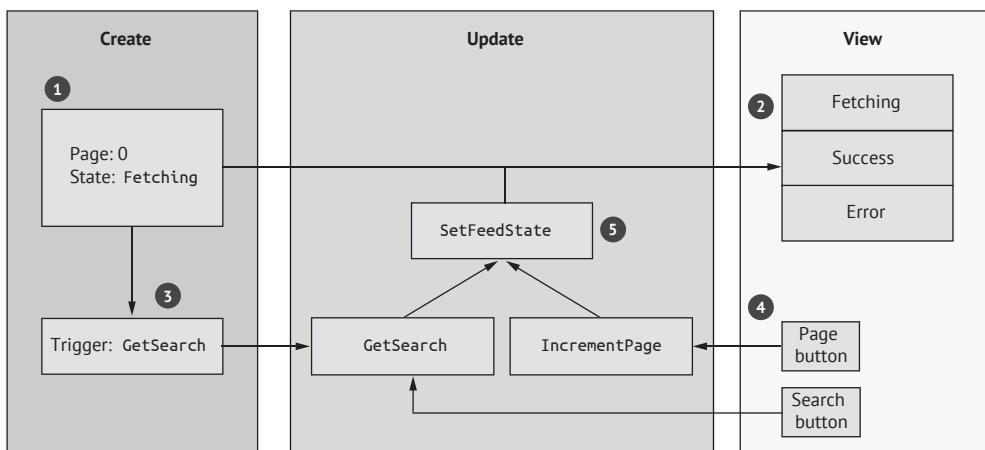


Рисунок 9.3 Поток данных в компоненте

Давайте начнем с создания представления и будем двигаться назад к фазам обновления и инициализации (создания). Благодаря этому можно будет понять, какие представления нужны и какие действия будут двигать эти изменения. С учетом этого представления нужно бу-

дёт соотнести различные состояния, установленные в перечислении `FetchState`. Затем каждое состояние будет генерировать HTML с помощью макрокоманды.

Листинг 9.23 lib.rs: реализация представления

```
impl Component for List {
    ...
    fn view(&self, ctx: &Context<Self>) -> Html {
        match &self.feed {
            FetchState::Fetching => html! { "Fetching" },
            FetchState::Success(data) => html! {
                <div>
                    <ul>
                        > { for data.entry.iter().map(|e| html!{
                            <li>
                                <a target="_blank" href={e.id.to_string()}>
                                    {e.title.to_string()}</a>
                            </li>
                        })}
                    </ul>
                    <button class="button" onclick={ctx.link().callback(
                        |_| Msg::IncrementPage)}>
                        { "More" }
                    </button>
                </div>
            },
            FetchState::Failed(err) => html! { err },
        }
    }
}
```

The diagram uses callout arrows to explain the code's behavior:

- Функция view требуется для вывода компонента на экран.** (Annotation above the first brace of the match block)
- Соотносит различные значения состояния страницы.** (Annotation to the right of the `FetchState` variants)
- Если есть данные, то обрабатывает и выводит их на экран.** (Annotation to the `FetchState::Success` branch)
- Просматривает каждую запись, как это делалось в компоненте React, чтобы создать список.** (Annotation to the `for` loop inside the `Success` branch)
- Если выполняется доставка, то выводит на экран слово Fetching.** (Annotation to the `Fetching` variant)
- Отправляет метод IncrementPage при нажатии на кнопку.** (Annotation to the button's `onclick` handler)
- В случае отказа выводит на экран сообщение об ошибке.** (Annotation to the `Failed` variant)

При доставке пользователь получает сообщение о том, что выполняется доставка данных. Аналогичным образом выводятся на экран все полученные ошибки. Эти состояния довольно просты и не требуют пояснений, но необходимы для информирования клиентов о происходящем. При получении данных делается нечто похожее на то, что делает компонент React: выполняются итерации по результатам и создается ссылка с кнопкой, которая вызывает действие по обновлению состояния страницы. Переменная `ctx` дает возможность подключаться к системе управления состоянием, которая принимает сообщение и вызывает функцию обновления, чтобы изменить состояние.

С учетом этого теперь можно увидеть различные мутации, которым может подвергнуться система. Один метод поможет устанавливать состояние, а два других будут манипулировать состоянием и запрашивать дополнительное обновление.

Листинг 9.24 lib.rs: реализация функций обновления

```
impl Component for List {
    ...
    fn update(&mut self, ctx: &Context<Self>, msg: Self::Message) {
        match msg {
            Msg::SetFeedState(fetch_state) => {
                self.feed = fetch_state;
            }
            Msg::IncrementPage => {
                self.page += 1;
            }
            ctx.link().send_message(Msg::GetSearch(self.page));
            false => {
                // Отправляется false, поэтому обновления не происходит до тех пор, пока не будет обновлено состояние.
            }
            Msg::GetSearch(page) => {
                ctx.link().send_future(async move {
                    match search("type".to_string(), page, 10).await {
                        Ok(data) => Msg::SetFeedState(
                            FetchState::Success(data)),
                        Err(err) => Msg::SetFeedState(
                            FetchState::Failed(err)),
                    };
                    // В противном случае выводит на экран сообщение об ошибке.
                });
                ctx.link().send_message(Msg::SetFeedState(
                    FetchState::Fetching));
            }
            true => {
                // В то время как это происходит, нужно вывести на экран состояние «fetching» (доставка данных).
            }
        }
    }
}
```

Функция обновления изменяет состояние страницы посредством передачи сообщений.

Возвращается true, поэтому компонент будет прорисовываться повторно после изменения состояния.

Отправляет вызов будущего, которое должно разрешиться, и обрабатывает изменение состояния.

В случае успеха передает данные для прорисовки.

Нужно соотнести все возможные отправки сообщений.

SetFeed подвергает состояние компонента мутации, устанавливая текущее состояние ленты feed.

IncrementPage увеличивает счетчик страниц и отправляет новое сообщение для поиска.

Отправляется false, поэтому обновления не происходит до тех пор, пока не будет обновлено состояние.

Вызывает функцию поиска и проверяет возвращенное значение.

Как можно заметить, этот метод возвращает булево значение. Оно используется компонентом для определения необходимости выполнять повторную прорисовку, которая должна происходить только при изменении состояния. Так, в первом методе просто присваивается состояние, ничего особенного. Это, в свою очередь, инициирует обновление представления в зависимости от состояния. Второй метод изменяет состояние страницы, но затем отправляет сообщение, чтобы вызвать функцию `GetSearch`. Контроль за этим мог бы осуществляться с помощью свойств, но вместо них здесь мы хотим продемонстрировать, как вызывать обновления из других обновлений наряду с возвратом значения `false`, чтобы представление не обновлялось.

`GetSearch` – это главный метод, который будет использоваться для вызова изначальной функции извлечения ленты. Указанный вызов обернут в асинхронный метод `async`; иными словами, нужно предоставить замыкание, которое будет исполнено, когда оно разрешится. После того как оно разрешится, состояние будет обновлено, и в нем будут представлены либо данные, либо сообщение об ошибке. Пока это происходит, состояние устанавливается равным доставке данных, чтобы пользователь понимал, что происходит.

Будем надеяться, что на данный момент вы видите то, как этот компонент перетекает из состояния представления, и способы воздействия на представление. Подводя итог, имеется функция, которая определяет внешний вид компонента в зависимости от заданного состояния; это и есть представление. Изменение состояния в функции обновления происходит с помощью внешнего инициатора (триггера). Это, в свою очередь, влияет на состояние, приводя к запуску представления и изменению его внешнего вида. Завершающая необходимая часть состоит в настройке начального состояния компонента при его создании. Она позволит выполнять два важнейших задания: создавать начальную структуру и отправлять начальный запрос на доставку данных.

Листинг 9.25 lib.rs: реализация начального состояния

```
impl Component for List {
    ...
    fn create(ctx: &Context<Self>) -> Self { ←
        ctx.link().send_message(Msg::GetSearch(0)); ←
        Self { ←
            page: 0,
            feed: FetchState::Fetching,
        }
    }
    ...
}
```

Создает начальную структуру и задает ее значения.

Метод `create` устанавливает начальное состояние компонента.

При запуске требуется получить первую страницу результатов, поэтому отправляется сообщение об обновлении.

Вот и все! Компонент готов, но нужно добавить еще один последний метод, который позволит выставить его в модуле Wasm.

Листинг 9.26 lib.rs: создание функции компонента

```
#[wasm_bindgen]
pub fn list_component() -> Result<(), JsValue> {
    yew::start_app::<List>(); ←
    Ok(())
}
```

Выставляет компонент в Wasm.

После этого можно собрать модуль Wasm заново.

Листинг 9.27 Консоль: сборка и обновление библиотеки

```
wasm-pack build --target bundler
cd pkg
npm link
cd ../papers-list
npm link papers
npm install
```

Откройте файл `App.jsx` и измените код на следующий ниже.

Листинг 9.28 App.jsx: монтирование компонента Wasm

```
import './App.css'

const wasm = await import('papers')

function App() {
    return (
        <div>
            <div>[wasm.list_component()]</div> ←
        </div>
    )
}
```

Просто вызвать метод компонента, и Yew сделяет все остальное!

Вот и все! Запустите свой сервер разработки и посмотрите, как это работает, – в точности как компонент React.

9.6**Еще раз о переработке JavaScript**

Подытоживая все изложенное, выше было продемонстрировано, как использовать библиотеку Rust, чтобы создать асинхронный метод (`async`) для извлечения документа RDF и его разбивки на страницы. Затем он был добавлен в веб-браузер и использован в качестве библиотеки Rust для поставщика данных, а также в качестве компонента. Язык Rust обеспечивает уровень защищенности и проверки качества кода «прямо из коробки», для чего в JavaScript требуется большое количество инструментов, чтобы справляться одинаково хорошо.

При рассмотрении эволюции различных проектов, которые были здесь выполнены, возможно, вам будет трудно понять, на каком этапе процесса вы находитесь и какое решение понадобится. Первый вариант использования появляется, когда есть алгоритм или процесс, который вы написали или переписали заново на языке Rust для выполнения в браузере в качестве скрипта. Это классический JavaScript или веб-модель, в которых работа веб-страницы состоит в обеспечении загрузки скриптов, чтобы их могли использовать другие скрипты, и, следовательно, контекст загружается только для этой страницы. Второй сценарий заключается в экспорте исходного кода Rust в качестве модуля или библиотеки, которые могут быть импортированы в другие

проекты на JavaScript, такие как компонент React. Это современный подход и наиболее вероятный сценарий применения разработчиками. Модули – это тот способ, которым управляет большинство крупных проектов JavaScript, и интеграция модулей Wasm станет более крупным расширением этого шаблона в будущем.

Наконец, на языке Rust разрабатывается целый веб-компонент. Данная технология все еще находится в зачаточном состоянии, и поэтому трудно определить траекторию развития этого шаблона. Тем не менее возможность разработки веб-компонента на языке Rust чрезвычайно полезна для сценариев, в которых разработка продукта с использованием только одного языка или ограниченного числа языков является привлекательной. В табл. 9.1 представлены различные варианты использования и шаблоны.

Таблица 9.1 Варианты использования фронтального кода Wasm

Вариант использования	Формат	Инструмент
Простая веб-страница	Скрипт	Флаг <code>web</code> инструмента <code>wasm-pack</code>
Библиотечная интеграция	Модуль	Флаг <code>bundler</code> инструмента <code>wasm-pack</code>
Элемент пользовательского интерфейса	Компонент	<code>Yew</code>

Теперь, когда налажена работа Rust по генерированию модуля Wasm, используемого во фронтальной части веб-приложения, можно посмотреть, как использовать Wasm в серверной части на предмет гораздо более масштабной переработки исходного кода.

Краткий итог

- Wasm – это универсальный формат байт-кода, который может выполняться в большинстве веб-браузеров.
- Wasm можно писать на языке Rust, чтобы использовать в простом JavaScript, посредством флага `--web` при использовании инструмента `wasm-pack`.
- Wasm применяется для написания библиотек JavaScript, которые можно импортировать в крупные веб-приложения, написанные на языке Rust, используя опцию `bundler` при компиляции посредством инструмента `wasm-pack`.
- Wasm используется в компонентах путем загрузки экспортированного модуля, что позволяет создавать более переносимый код и интегрировать его в современные фреймворки и библиотеки.
- В формате Wasm и на языке Rust с применением библиотеки Yew можно писать полноценные веб-компоненты.

10

Интерфейс с WebAssembly для переработки исходного кода

Эта глава охватывает следующие ниже темы:

- написание модуля WebAssembly (Wasm) для исполнения в виртуальной среде исполнения;
- интеграция модуля Wasm в исполняемый файл Rust для вывода данных;
- использование памяти Wasm для нечисловых данных.

Язык программирования Java был выпущен в 1995 году под смелым лозунгом «Напиши один раз и запускай в любом месте» (Write Once Run Anywhere, аббр. WORA). Концепция написания исходного кода, который может выполняться где угодно, не нова; это уже делалось раньше на других языках, таких как Smalltalk, и сегодня кажется малозначительным, учитывая обширные возможности менеджеров пакетов, интерпретируемых языков и изощренных компиляторов, доступных разработчикам. Но то, что сделал язык Java, было поистине удивительным. В течение нескольких лет он стал одним из самых

популярных языков; он оставался таким на протяжении двух десятилетий и почти успешно внедрился во все возможные программы, включая веб-браузер.

ПРИМЕЧАНИЕ. Язык Java в какой-то момент был настолько популярен, что повлиял на название перспективного веб-языка, сейчас известного как JavaScript, хотя эти два языка никак друг с другом не соотносятся.

Когда в компании Sun Microsystems разрабатывался язык Java, там обратили внимание на недавно появившуюся Всемирную паутину. В Sun осознали, что в этой области наравне с наличием нового оборудования, такого как сотовые телефоны, есть большой потенциал роста. Компания Sun столкнулась с проблемами, связанными с языками C и C++, работающими на своем специализированном оборудовании, и решила создать язык с собственной *виртуальной машиной*, которая может интерпретировать специализированный язык сборки, именуемый сборочным, или *ассемблерным, кодом*. Виртуальная машина преобразовывает ассемблерный код в *байт-код* хоста, чтобы выполнять приложения практически на любом устройстве, поддерживаемом виртуальной машиной.

Разработка на языке Java дала разработчику возможность писать приложение, которое компилировалось в модуль, именуемый JAR (Java Archive), то есть в zip-файл, содержащий байт-код Java, группированный по классам, как показано на рис. 10.1. Затем этот архив мог быть отправлен на другой компьютер и исполнен или использован в качестве библиотеки. Файлы могли исполняться при условии, что на компьютере присутствует среда исполнения Java (Java Runtime Environment,abbr. JRE). Указанная среда могла работать даже в браузере в виде небольшого приложения, именуемого апплетом.

Язык Java по-прежнему широко распространен в отрасли и все так же является одним из самых продуктивных языков программирования, когда-либо разработанных. Он проложил путь для многих методов разработки, которые применяются сегодня. Примечательно, что по-прежнему остается важным понятие возможности писать исходный код и выполнять его где угодно. Язык Java смог создать новую платформу разработки и для других разработчиков. Scala, Clojure, Kotlin и несколько других языков компилируются в байт-код Java и могут выполняться везде, где присутствует среда исполнения Java (JRE).

История Java может дать представление о будущем WebAssembly (Wasm). В предыдущей главе рассматривалось использование языка Rust для написания исходного кода на JavaScript, но на самом деле этот процесс был введением в гораздо более широкую возможность написания исходного кода, который выполняется где угодно и на любом языке, который нравится.

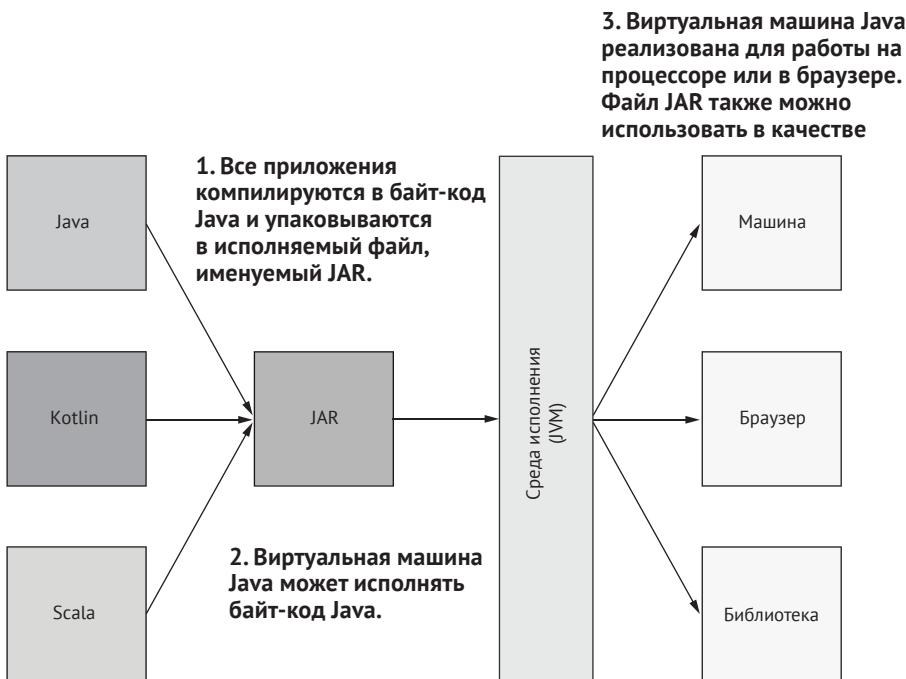


Рисунок 10.1 Языки, компилируемые в JAR, и возможность их выполнения в нескольких местах

В каждой главе этой книги демонстрируются возможности языка Rust по интеграции с другими языками, такими как Python, C++ и JavaScript, но это только верхушка айсберга. Несмотря на то что операционная совместимость между Rust и различными языками на основе С огромна и впечатляет, мир байт-кода Wasm превратил язык Rust в один из ключевых языков для более широкой операционной совместимости. Язык Rust добился этого, создав модули WebAssembly, которые представляют собой библиотеки или исполняемые файлы, аккуратно упакованные под использование совместимыми средами исполнения Wasm или библиотеками, во многом похожими на JAR в экосистеме языка Java.

В 2019 году компания Mozilla анонсировала инициативу под названием Системный интерфейс с WebAssembly (WebAssembly System Interface,abbr. WASI). Эта инициатива, по сути, вывела Wasm из браузера и применила практически ко всем средам исполнения и языкам, при условии что имеется поддерживающая библиотека или двоичный код. Как и виртуальная машина Java (JVM), системный интерфейс WASI утверждает набор протоколов, которые позволяют языку ассемблера взаимодействовать с опорной системой. Этот набор протоколов называется *двоичным интерфейсом приложений* (application binary interface, abbr. ABI), в рамках которого два двоичных модуля могут взаимодействовать через машинный код или ассемблерный код. По сравнению с высоковысокими API эти протоколы являются очень низкоуровневыми.

Таким образом, у вас есть модуль WebAssembly, который может работать в веб-браузере или в командной оболочке с добавлением одного готового к исполнению двоичного файла, либо модуль, который может быть импортирован и выполнен любым поддерживаемым языком. Мощь такого рода гибкости можно увидеть через призму того, как сегодня используются контейнеры в разработке.

Многие рассматривают связку Wasm + WASI как следующий шаг в создании универсально выполняющихся приложений. Некоторые разработчики переходят на контейнеры, чтобы укомплектовывать свои приложения для выполнения в любом месте. Однако двоичный формат Wasm и системный интерфейс WASI предоставляют альтернативу, помогающую осуществлять постепенную миграцию устаревших систем, позволяя разработчикам создавать свою собственную среду исполнения, чтобы встраивать старую деловую логику или менее быстродействующий исходный код (рис. 10.2).

Рассмотрим нынешний тренд написания приложений в виде функций в качестве службы. И здесь снова для данного языка создаются общая среда исполнения и целевой объект, а код загружается во временный контейнер. Временный контейнер выполняется в абстракции поверх опорной операционной системы. В этом сценарии выполняющее функцию хост-приложение¹ ограничивает среду исполнения контейнеров только предопределенным способом. Предоставляемый этими службами ограниченный API обычно нуждается в единой точке входа, которую необходимо реализовать. Код предоставляется хост-системе² в сыром виде, и система компилирует и монтирует его во время исполнения. Затем хост-система пытается вызвать функцию на основе языка и других конфигурационных данных, требуемых провайдером.

Как вы увидите, двоичный формат Wasm нуждается в определении интерфейса. Этот интерфейс будет использоваться для того, чтобы клиенты могли вызывать конкретные функции в среде исполнения. Вместо среды исполнения, подобной среде JRE или среде исполнения контейнеров, можно будет определять интерфейсы и способы взаимодействия с опорной системой. Вы сможете предоставлять диапазон, необходимый для того, чтобы сделать свою систему настолько гибкой, насколько потребуется, давая возможность любому создавать API-подобные приложения, или настолько ограниченной, насколько потребуется, чтобы оградить сегмент устаревшего кода. Вы сможете выбирать язык для среды исполнения Wasm, а ваши пользователи смогут писать исходный код на любом удобном для них языке.

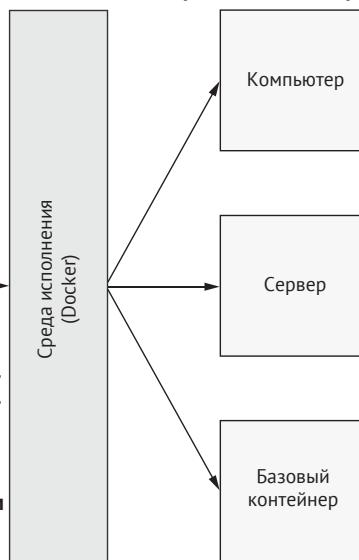
¹ Хост-приложение (host application) – это главное приложение, обеспечивающее среду, в которой работают другие приложения или компоненты, нередко взаимодействуя с ними через определенные интерфейсы или протоколы. – Прим. перев.

² Хост-система (host system) – это инфраструктура, программное обеспечение или платформа, которая поддерживает работу приложений, предоставляя необходимые ресурсы, такие как хранилище, вычислительные мощности и сетевые возможности. – Прим. перев.

1. Все приложения компилируются и помещаются в контейнер со всеми зависимостями, необходимыми приложению для выполнения.



2. Контейнеры предоставляют абстракцию полной системы для выполнения приложения без знания об определении опорного компьютера.



3. Контейнеры могут выполняться с помощью надлежащей среды исполнения – на машине, используемой в качестве базового образа, или на специализированном контейнерном сервере, таком как Kubernetes.

Рисунок 10.2. Среды исполнения контейнеров аналогичны виртуальным машинам

ПРИМЕЧАНИЕ. Для того чтобы ознакомиться с обновленным списком поддерживаемых языков, перейдите по этой ссылке: <https://github.com/appcypher/awesome-wasm-langs>.

Дабы акцентировать внимание на этой мысли, Соломон Хайкс, один из создателей платформы Docker, в 2019 году написал в Твиттере следующее:

Если бы связка Wasm+WASI существовала в 2008 году, то нам не пришлось бы создавать платформу Docker. Вот насколько это важно. Веб-сборка на сервере – это будущее вычислений. Стандартизованный системный интерфейс был недостающим звеном. Будем надеяться, что WASI справится с этой задачей!

Наличие универсальной среды исполнения – это задача, которую разработчики пытались решить с помощью различных технологий, но Wasm, похоже, является именно тем решением, которое многие искали; сейчас он находится на переднем крае, и язык Rust является одним из самых важных языков.

10.1 Универсальная среда исполнения системного интерфейса WASI

Так что же такое системный интерфейс WASI? В главе 9 уже был обследован вопрос о том, как Wasm работает в контексте браузера. Код

WebAssembly находится в скомпилированном формате, который может интерпретироваться веб-браузером или любой средой исполнения, способной интерпретировать этот скомпилированный код. Среда исполнения, способная интерпретировать двоичный код WebAssembly, делает это с помощью системного интерфейса с WebAssembly, или WASI. Теперь код вызывается из среды исполнения WebAssembly внутри веб-браузера с использованием JavaScript. Вместо этого можно писать исполняемый файл или библиотеку, которые могут использоваться любым другим языком, поддерживающим WASI. Системный интерфейс WASI – это инструмент, который дает возможность создавать собственную среду исполнения или использовать среду исполнения, предоставленную кем-то другим. Интерфейс понимает опорный двоичный код WebAssembly и интерпретирует его так же, как среда исполнения Java делает с байт-кодом. В данном случае WASI интерпретирует код Wasm так же, как среда исполнения Java интерпретирует байт-код Java. Разница в том, что можно написать свою собственную виртуальную машину, подобную виртуальной машине Java.

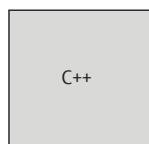
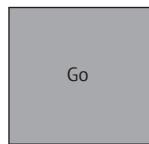
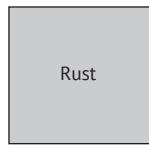
Системный интерфейс WASI позволяет разработчику определять способ взаимодействия внешних библиотек с опорным кодом, который предоставляется хостом. Буква *I* в WASI означает *интерфейс*, и это то, что разработчик будет определять для кода хоста, который он пишет. В программном обеспечении определяются интерфейсы, но не обязательно создается реализация. Точно так же хост предоставляет способ взаимодействия с внешней библиотекой или приложением, не зная, как работает библиотека. При определении интерфейса с набором заданных параметров и выходных данных можно реализовать другой код, который будет его вызывать. Таким образом, в Wasm может быть написана библиотека, которая соответствует интерфейсу, определяемому хост-приложением. Затем библиотека может быть подменена, не затрагивая хост-приложения. Эта способность будет демонстрироваться на протяжении всей главы. Вдобавок модули Wasm можно подменять *без остановки хост-приложения*. Создание интерфейса позволяет выполнять все, что угодно, при условии что оно реализует этот интерфейс, как показано на рис. 10.3.

Исполнение произвольного кода внутри другого куска кода имеет опасные последствия. В течение многих лет мы пытались предотвращать атаки с использованием межсайтового скриптинга¹ или SQL-инъекций, ограничивая то, что пользователи могли вводить в различные формы и поля. Поэтому идея создания среды исполнения, в которой может исполняться любой произвольный файл Wasm, кажется сомнительной. Однако в стандарт Wasm встроен механизм обеспечения безопасности с целью предотвращения атак. Модули Wasm выполняются в изолированной среде, которая принимает входные данные

¹ Слово «межсайтовый» в термине «межсайтовый скриптинг» (Cross-Site Scripting, XSS) относится к тому, что атака происходит на одном веб-сайте, а ее последствия проявляются на другом или нескольких других веб-сайтах, как правило, через вредоносный код, внедряемый на страницу жертвы. – Прим. перев.

только в двух формах: в виде прямых аргументов (например, для функций командной оболочки) либо через буфер памяти, зарезервированной исключительно в изолированной среде внешнего модуля. Хост должен просить модуль управлять памятью в некотором объеме и предоставлять функции чтения из этой памяти. Подобно контейнерам, виртуальной машине Java и другим виртуальным средам исполнения, модуль работает так, как если бы он был единственным приложением во вселенной, не имея понятия о том, что находится за его пределами. Благодаря такой дополнительной защите можно исполнять файлы Wasm, не опасаясь, что приложение совершил что-либо вредоносное без явного разрешения вызывающей службы.

1. Поддерживаемые языки могут быть скомпилированы в двоичный формат WebAssembly.



2. Wasm может быть укомплектован и поставлен в виде единого файла, аналогичного файлам JAR или контейнерам, и выполнен в системе, поддерживающей интерфейс с Web Assembly.



3. Как вы убедились в предыдущей главе, файлы Wasm можно выполнять в браузере посредством определенной виртуальной машины на компьютере или в выделенной среде исполнения.



Рисунок 10.3. Wasm, работающий в среде исполнения WASI

Цель двоичного формата Wasm – сделать код переносимым и модульным. Как разработчик вы обычно хотите писать исходный код на удобном для вас языке и использовать полезные для вас библиотеки. Однако у других языков имеются преимущества в определенных областях или библиотеках, к которым просто нет доступа. В качестве примера возьмем язык Python, на котором очень просто писать и который отлично подходит для машинного обучения. Тем не менее Python – это скриптовый язык, и, как вы увидели в главе 7, его можно улучшать путем замены исходного кода на языке Python модулем, написанным на языке C++ или Rust. Однако с появлением системного интерфейса WASI отпадает необходимость ограничиваться только

этими языками. Теперь можно использовать функцию, написанную на языке Go, чтобы выполнять некие эффективные операции, без изменения опорного исходного на языке Python.

У разработчика на языке Rust или C++ может возникнуть аналогичная потребность использовать исходный код на языке Python или инструмент языка Go и обеспечить безопасность кода. В этом случае можно воспользоваться защищенностью языка Rust и изолированной средой Wasm, чтобы изолировать и исполнять небезопасный код в ограниченной среде. Благодаря обширной поддержке WebAssembly со стороны языка Rust с его помощью можно проводить «окончательную переработку» исходного кода. Поскольку в Rust есть весь инструментарий поддержки двоичного формата Wasm и системного интерфейса WASI, можно перерабатывать части своей системы и использовать ее в Rust или переписывать исходный код на языке Rust, компилировать его в Wasm и передавать другому языку на исполнение. Давайте посмотрим, как это работает.

Взглянем на WASI с точки зрения виртуальной машины. WASI предоставляет интерфейс, который позволяет другому приложению использовать модуль Wasm, подобно тому, как устанавливается двоичный файл в рамках изолированной платформы, такой как Docker. Опорный контейнер может содержать базовый образ, который предоставляет запущенному приложению такие ресурсы, как файловая система, утилиты и даже библиотеки. WASI может делать то же самое. Давайте рассмотрим легковесную среду WasmEdge в качестве примера среды выполнения двоичного формата WebAssembly, которая также предоставляет некоторые библиотеки для использования при разработке.

Среда исполнения WasmEdge поддерживается Фондом облачно-ориентированных вычислений (Cloud Native Computing Foundation, аббр. CNCF), чтобы помочь создавать стандартную среду исполнения для двоичного кода WebAssembly, а также комплект инструментов разработки (SDK). В этой главе среда исполнения WasmEdge будет использована для того, чтобы помочь с большей частью шаблонного и низкоуровневого инструментария Wasm, о котором нам как разработчикам нет нужды слишком беспокоиться. Нюанс заключается в том, что все, что будет делаться в приводимом примере, может быть реализовано с помощью рядовых инструментов WebAssembly. Если вашему приложению требуется меньше абстракций, то вам, возможно, потребуется более внимательно изучить спецификации системного интерфейса WASI.

Среда исполнения WasmEdge предоставляет SDK для Rust, JavaScript, Go, C, C++, Java и нескольких других языков, так что это отличный выбор для переработки исходного кода в данной книге. Проект WasmEdge развивается, как и вся экосистема двоичного кода Wasm, поэтому важно быть в курсе меняющегося ландшафта. Для того чтобы начать работу со средой WasmEdge, надо установить библиотеку, используя следующую ниже команду Linux (или найдите команду, которой будет соответствовать ваше ОС, по адресу <https://wasmedge.org/docs/start/install>).

Листинг 10.1 Команда: установка среды исполнения WasmEdge

```
curl -sSf https://raw.githubusercontent.com/WasmEdge/WasmEdge/master/  
→utils/install.sh | bash
```

После установки пакета WasmEdge у вас будет доступ к среде исполнения WasmEdge, а также к некоторым ее инструментам. Следуйте инструкциям на экране, чтобы завершить установку. Проверка работоспособности кода выполняется путем выполнения готового модуля Wasm. Для того чтобы его протестировать, давайте скачаем файл Wasm и выполним его.

Листинг 10.2 Команда: тестирование среды исполнения WasmEdge

```
wget https://github.com/second-state/rust-examples/releases/latest/  
→download/hello.wasm
```

Вы должны увидеть, как на экране будет напечатано «Привет, мир!». Что произошло? Ранее в этой главе сравнивались файлы Wasm с файлами JAR в том смысле, что они представляют собой скомпилированный предупакованный код, который может исполняться в соответствующих средах исполнения без учета архитектуры опорной системы (Windows, Linux, 64-битовая, 32-битовая версии). Можно обратиться к исходному коду модуля Rust и посмотреть, как он работает, но вовсе не нужно беспокоиться о том, как он собран или как он будет исполнен, поскольку среда исполнения занимается этим за нас. Аналогичную ситуацию можно представить, скачав и исполнив файл JAR для Java.

Листинг 10.3 Команда: исполнение файла JAR

```
java -jar hello.jar
```

В предыдущей главе Wasm использовался для выполнения кода на JavaScript в веб-браузере. Поскольку целевой объект компиляции должен был принимать язык JavaScript, был предоставлен код загрузки файла Wasm в среду исполнения браузера, чтобы иметь возможность исполнять логику и извлекать значения из API. Точно так же создается среда хоста вне браузера, чтобы вместо этого служить в качестве utility командной оболочки. Давайте посмотрим, как разложить работу на части, чтобы создать платформенно-независимый инструмент командной оболочки, который получает библиотеки Wasm для деловой логики указанного инструмента.

10.2 Из браузера на машину

Давайте вернемся к проекту из предыдущей главы. Там было решено разработать модуль Wasm, который вызывал бы внешний API и выполнял разбор результатов, чтобы показывать их в веб-браузере.

Нередко в ситуациях, когда действует конечная точка API и данные транслируются в формат, который может использоваться вашим предприятием, появляется возможность распространить этот подход на другие области. Если взять ту же идею и совместить ее со связкой Wasm/WASI, то можно создать высокопереносимую и расширяемую систему.

Для начала создадим новый зонтичный проект под названием *workspace* (рабочее пространство), в котором будут размещены библиотечный и двоичный файлы. В рабочем пространстве коллективно используются один и тот же файл *Cargo.lock* и выходной каталог, чтобы помогать в организации пакетов и держать соотносящиеся пакеты вместе. Давайте создадим новый проект.

Листинг 10.4: Команда: создание нового проекта

```
mkdir journal
cd journal
```

Затем создаются библиотечный и двоичный файлы.

Листинг 10.5 Команда: создание нового двоичного и библиотечного файлов

```
cargo new paper_search_lib --lib
cargo new paper_search
touch Cargo.toml
```

Создает библиотеку, используя код из предыдущей главы.

Создает новый двоичный файл для приема библиотеки.

Создает зонтичный файл Cargo.

Далее в файл *Cargo.toml* добавим рабочие пространства, используя следующее ниже содержимое.

Листинг 10.6 Cargo.toml: файл рабочих пространств

```
[workspace]
members = [
    "paper_search",
    "paper_search_lib"
]
```

members – это библиотеки, которыми вы хотите, чтобы менеджер пакетов Cargo управлял за вас.

Наконец, нужно проследить за наличием целевого объекта, которым будет *wasm32-wasi*.

Листинг 10.7 Команда: установка целевого объекта

```
rustup target add wasm32-wasi
```

Откройте файл *lib.rs*. Ниже будет добавлена часть исходного кода из предыдущей главы, чтобы создать функцию поиска. Сначала добавим структуры.

Листинг 10.8 paper_search_lib/src/lib.rs: копирование структур

```
use serde::{Deserialize, Serialize};
use std::env;
use std::error::Error;
use std::fmt::{self, Debug, Display, Formatter};

#[derive(Default, Debug, Clone, PartialEq, Serialize, Deserialize)]
pub struct Feed {
    pub entry: Vec<Entry>,
}

#[derive(Default, Debug, Clone, PartialEq, Serialize, Deserialize)]
pub struct Entry {
    pub id: String,
    pub updated: String,
    pub published: String,
    pub title: String,
    pub summary: String,
    pub author: Vec<Author>,
}

#[derive(Default, Debug, Clone, PartialEq, Serialize, Deserialize)]
pub struct Author {
    pub name: String,
}
```

Этот исходный код предоставляет XML-структуру для возвращаемых элементов. Затем скопируем функцию поиска.

Листинг 10.9 paper_search_lib/src/lib.rs:
копирование функции поиска

```
pub async fn search(term: String, page: isize, max_results: isize) -> Result<Feed, reqwest::Error> {
    let http_response = reqwest::get(format!(
        "http://export.arxiv.org/api/query?search_query=all:{}&start={}&max_results={}",
        term,
        page * max_results,
        max_results
    ))
    .await?;
    let b = http_response.text().await?;
    let feed: Feed = serde_xml_rs::from_str(b.as_str()).unwrap();
    return Ok(feed);
}
```

После того как исходный код будет готов, нужно добавить зависимости, открыв файл `paper_search_lib/Cargo.toml` и добавив следующие ниже библиотеки.

Листинг 10.10 paper_search_lib/Cargo.toml:
библиотечные зависимости

```
[package]
name = "paper_search_lib"
version = "0.1.0"
edition = "2021"

[dependencies]
tokio_wasi = { version = "1.21", features = ["rt", "macros", "net", "time"] }
reqwest_wasi = "0.11"
serde = { version = "1.0", features = ["derive"] }
serde-xml-rs = "0.6.0"
```

Благодаря этому была создана библиотека, которую модуль будет вызывать. Важно обратить внимание на то, как изолируются и обособляются различные сегменты работы, чтобы обеспечить отсутствие между ними конфликтов и чтобы деловая логика не затерялась в прикладной логике. Поскольку модуль Wasm является доставляемым двоичным файлом, желательно сделать его независимым. Данный модуль будет скомпилирован в формате Masm, но будет зависеть от некоторых библиотек из среды исполнения WasmEdge.

Давайте посмотрим, как создается простой исполняемый файл Wasm, который может выполняться в среде WasmEdge. Сейчас нужно добавить немного содержимого в файл [paper_search/Cargo.toml](#).

Листинг 10.11 paper_search/Cargo.toml: двоичные зависимости

```
[package]
name = "paper_search"
version = "0.1.0"
edition = "2021"

[build]
target="wasm32-wasi" <-- Сборка всегда
                           нацелена на Wasm.

[target.wasm32-wasi] <-- В частности, требуется, чтобы
                           исполнитель был определен
                           для WasmEdge.

runner = "wasmedge" <-- Импортирует функциональ-
                          ность из предыдущей главы.

[dependencies]
tokio_wasi = { version = "1.21", features = ["rt", "macros", "net", "time"] }
paper_search_lib = { path = "../paper_search_lib" } <--
```

Теперь можно написать двоичный файл, который вызывает библиотечную функцию и печатает результаты в стандартный вывод (которым обычно является консоль). Он будет скомпилирован в двоичный формат Wasm, а затем его можно будет выполнить через установленного исполнителя WasmEdge.

Листинг 10.12 paper_search/src/main.rs:
получение результатов поиска

```
use std::error::Error;
use std::fmt::{self, Debug, Display, Formatter};

fn main() -> Result<(), Box> {
    let res: Vec<String> = search("rust".to_string(), 0, 10).unwrap(); ←
    for entry in res.iter() { ←
        println!("{}?", entry); ←
    } ←
    Ok(())
}
```

Извлекает результаты
поиска с использова-
нием термина.

Распечатывает резуль-
таты на экране.

Выше было создано простое приложение WebAssembly, которое станет основой для остальной части инструмента. Этот модуль будет выставлять наружу только функцию `main`, которую среда WasmEdge будет вызывать. Модуль `paper_search`, в свою очередь, будет использовать опорную деловую логику, которая была написана в `paper_search_lib`. Позже будут представлены функции, для того чтобы этот модуль можно было использовать как библиотеку. После выполнения кода можно увидеть, как результаты будут распечатаны в стандартном выводе. Что означает стандартный вывод в терминах модуля? Ответ зависит от опорной реализации. Нельзя сказать точно, как среда WasmEdge обрабатывает эти результаты. Выходные данные зависят от опорной среды исполнения, которая на данный момент будет просто тем, что предоставляет нам среда WasmEdge. Определение среды WasmEdge в качестве целевого исполнителя сообщает языку Rust о том, что нужно компилировать, используя функции, определенные в комплекте инструментов разработки для WasmEdge, что обеспечивает более оптимальную операционную совместимость с системой. По сути, среда WasmEdge выполняет требования интерфейсов ABI, о которых упоминалось в начале главы, и в модуле будет реализация этих интерфейсов.

Функция поиска была обособлена, чтобы позже ее расширить и использовать этот модуль также в качестве библиотеки. Поэтому будет лучше, если сделать ее доступной на ранней стадии.

Листинг 10.13 paper_search/src/main.rs:
функция Wasm для доставки результатов

```
pub fn search(
    term: String,
    page: isize,
    max_results: isize,
) -> Result<Vec<String>, Boxполнять не нужно, поэтому он  
будет заблокирован. Для этого  
нужен поток исполнения.
```

```

let feed: paper_search_lib::Feed = rt.block_on(async {paper_search_lib::
    search(term, page, max_results)
    .await}).unwrap(); ← Ожидание будет на этом потоке до тех
let res = feed
    .entry
    .into_iter()
    .map(|e| format!("{} {}", e.title, e.id))
    .collect::<Vec<String>>();
return Ok::<Vec<String>, Box<dyn std::error::Error>>(res);
}
} Форматирует значения ленты новостей в строковые литералы.

```

Построив точку входа в модуль, теперь его можно вызвать, но сначала необходимо скомпилировать модуль, чтобы использовать среду исполнения Wasm. Затем модуль можно будет вызывать через WasmEdge.

Листинг 10.14: Команда сборки в Wasm

```

cargo build --target wasm32-wasi ← Компилирует в Wasm.
wasmedge target/wasm32-wasi/debug/paper_search.wasm ←
    Выполняет файл Wasm, используя
    среду WasmEdge.

```

Вы должны увидеть, как на экране будут напечатаны данные! За кулисами среда WasmEdge монтирует модуль Wasm и вызывает функцию `main`. И это все. Думайте об этом почти как об интерпретаторе, подобном тому, который используется в языке Python, за исключением того, что исходный код уже скомпилирован. При использовании универсальной среды исполнения есть доступ к простой функциональности, такой как исполнение модуля и печать результатов. Среда исполнения WasmEdge не имеет понятия о самом коде, но знает, как взаимодействовать с операционной системой. Эта похожая на виртуальную машину структура обеспечивает переносимость при условии наличия этого двоичного файла в других системах. Вы могли бы установить свой модуль Wasm на телефон или в Windows, или в Mac, и все это работало бы.

Однако приведенный выше пример оставляет желать лучшего. Прямо сейчас нет возможности взаимодействовать с модулем. Давайте это исправим, передав аргументы в модуль. Аргументы передаются так же, как и в любом другом приложении или двоичном коде, – через список `args`.

Листинг 10.15 paper_search/src/main.rs: модуль Wasm

```

use std::env;
use std::error::Error;
use std::fmt::{self, Debug, Display, Formatter};

fn main() -> Result<(), Box<dyn std::error::Error>> {

```

```

let mut args: Vec<String> = env::args().skip(1).collect();
args.reverse();
let term = args.pop().unwrap_or("rust".to_string());

let res: Vec<String> = search(term, 0, 10).unwrap();
for entry in res.iter() {
    println!("{}:?", entry);
}
Ok(())
}

```

Извлекает результаты поиска, используя указанный термин.

Извлекает входные аргументы для поискового термина; в противном случае по умолчанию используется значение rust.

Распечатывает результаты на экране.

Опять же, это простое приложение, но в нем много чего происходит. И здесь снова обнаруживаем, что среда WasmEdge предоставляет инструменты для извлечения аргументов в виде строкового литерала внутри пакета `env`. Но вот интересный факт о Wasm: у него нет строкового типа. Как будет показано в следующем далее разделе, Wasm не имеет понятия о строковых литералах или символах как о примитивах языка; поэтому выполнение чего-то столь простого, как передача строкового литерала, фактически возлагает работу по реализации и обработке этой работы на сам двоичный файл среды исполнения. Таким образом, среда WasmEdge, если выполнять ее двоичный файл, предоставляет ввод строковых данных бесплатно, но позже это будет реализовано самостоятельно.

Давайте посмотрим, как передача аргумента изменит результаты.

Листинг 10.16 Команда: сборка Wasm

```
wasmedge target/wasm32-wasi/debug/paper_search.wasm test
```

Передает аргумент.

Достигнутое на данный момент эквивалентно написанию автономного приложения. Входные данные принимаются посредством аргументов, а выходные данные пишутся на консоль. То, что было построено, уже является довольно мощным инструментом. Двоичный файл в среде WasmEdge предоставляет дополнительные возможности по расширению взаимодействия с опорной системой, позволяя при этом выполнять любой модуль Wasm. Но что, если вместо вызова двоичного файла потребуется использовать модуль как библиотеку?

10.3 Библиотека Wasm

Давайте еще раз рассмотрим функцию поиска. Прямо сейчас модуль поиска выполняется в виде двоичного файла, во многом так же, как если бы скомпилировать приложение Rust и исполнить его посредством функции `main`. Но рассмотрим использование среды WasmEdge, чтобы обойти функцию `main` и вместо этого исполнять выставленную наружу функцию внутри библиотеки. Делая так, мы выходим за рамки спо-

собностей среды WasmEdge передавать строковые значения и вместо этого должны полагаться на поддерживаемые примитивы, которые предоставляет Wasm. Давайте начнем с создания функции, которая просто обеспечивает статический поиск, где можно указывать страницу и значение смещения. При этом необходимо передавать только поддерживаемые примитивы и выставить функцию наружу, используя макрокоманду `#[no_mangle]`, которая уберегает имя подлежащей исполнению функции от искажения.

Листинг 10.17 paper_search/src/main.rs: библиотека Wasm

```
use std::env;
use std::error::Error;
use std::fmt::{self, Debug, Display, Formatter};

fn main() -> Result<(), Box

```

Как уже упоминалось ранее, этот исходный код выставляет наружу не функцию `main`, а библиотечные функции. Позже вы увидите, какая работа требуется виртуальной машине для трансляции и вставки строкового литерала в модуль. До сих пор среда WasmEdge делала это за нас, когда мы исполняли модуль в виде двоичного файла. Но теперь эту функциональность нужно обойти, используя флаг WasmEdge `--reactor`, который позволяет вызывать отдельную функцию. На данный момент среда WasmEdge больше не предоставляет инструменты для трансляции строкового литерала в формат, который библиотека способна обрабатывать. Нам не предоставляется набор аргументов, передаваемых в главное приложение, и вместо этого мы интерпретируем значения, вставленные в функцию. Все это означает потерю

функциональности при непосредственном вызове функции, в противоположность исполнению модуля. Поэтому остается набор поддерживаемых Wasm примитивов:

- **i32**: 32-битовое целое число;
- **i64**: 64-битовое целое число;
- **f32**: 32-битовое число с плавающей точкой;
- **f64**: 64-битовое число с плавающей точкой;
- **v128**: 128-битовый вектор целочисленных данных, данных с плавающей точкой или один-единственный 128-битовый тип.

За пределами примитивных типов ответственность ложится на модуль и среду исполнения. Нам предоставляется возможность создавать собственные контракты между тем, как мы хотим, чтобы система работала, и модулями, которые мы выбираем для выполнения в этой системе. Итак, если вы ожидаете, что в среду исполнения будет передаться строковый литерал, или JSON, или что-то еще, то нужно написать библиотеку или механизм обработки этой ситуации. Позже будет обследован вопрос о добавлении управления памятью, но сначала давайте обсудим сложные данные, чтобы объяснить, куда будет перемещаться приложение и почему надо начинать с простого процесса разбивки на страницы.

Среда WasmEdge позволяет вызывать эти библиотеки непосредственно из модуля Wasm, хотя теряется возможность использовать строковые литералы. Давайте скомпилируем и протестируем новую библиотеку модуля, выполнив следующую ниже команду.

Листинг 10.18 Команда: сборка Wasm

```
cargo build --target wasm32-wasi
wasmedge --reactor target/wasm32-wasi/debug/paper_search.wasm
→static_search 0 1
```

Флаг **reactor** позволяет вызывать
функции непосредственно внутри
модуля и передавать аргументы.

Вы должны увидеть те же результаты, что и раньше, но теперь на прямую вызывается библиотека, а не главная точка входа в двоичный файл. Таким образом, теперь вы научились собирать файл Wasm, который функционирует и как двоичный файл, и как библиотека. Но как его выполнить в Rust? Хотя в исходном коде все еще есть жестко привязанный строковый литерал, вы, возможно, начинаете видеть, где можно добавить в систему гибкости. Можно определить библиотеки, которые принимают аргументы из среды исполнения и передаются в произвольную библиотеку. Далее, выйдя за рамки использования существующей среды исполнения, вы начнете писать свою собственную.

10.4 Потребление Wasm

Доклад Стива Клабника «Rust, WebAssembly и будущее бессерверных технологий» (Steve Klabnik, «Rust, WebAssembly, and the Future

of Serverless», <https://mng.bz/X7YI>) начинается с обсуждения темы среды исполнения, перегруженности данного термина и неправильного его понимания в рамках разработки программного обеспечения. Почти у каждого языка, если только этот язык не является языком ассемблера, есть среда исполнения. Даже компилируемые языки предлагают своего рода среду исполнения для простых задач, таких как управление памятью или другие низкоуровневые операции. *Среда исполнения* – это часть кода, которая исполняется для того, чтобы помогать выполнять другой код, будь то на языке ассемблера или другом промежуточном языке. В случае с предыдущими примерами применялась среда исполнения WasmEdge. В главе 9 использовалась среда исполнения JavaScript в браузере. Тем не менее с WASI есть возможность начать писать свою собственную среду исполнения и внедрять файлы Wasm, как показано на рис. 10.4. Именно здесь можно изменить историю переработки исходного кода.

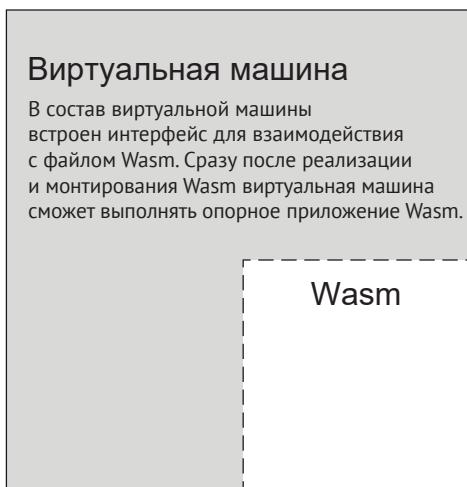


Рисунок 10.4. Выполнение Wasm с помощью комплекта инструментов разработки (SDK)

Большинство обсуждавшихся в этой книге проектов демонстрировали методы вкладывания языка Rust в другой язык, такой как C или Python, или вызова одного из этих языков из языка Rust. Эти методы являются очень консервативными и превосходными способами переработки исходного кода, но подумайте о мощи, которую придает возможность написания собственной среды исполнения. Внезапно появляется способность выполнять любой код, который требуется, при условии что он компилируется в Wasm – это означает, что написанная на C надоедливая деловая логика может быть скомпилирована в модуль Wasm и выполняться в исходном коде Rust, как если бы это была любая другая библиотека. После сборки инструментов для поддержки гибкой структуры с использованием Wasm свой код можно будет рассматривать не просто как узел в более крупной паутине, а как платформу.

Рассмотрение способности встраивать внешнюю логику в созданную систему должно вызывать озабоченность вокруг степени состыкованности систем. Важными факторами в том, как система будет использовать Wasm, становятся конструктивные и архитектурные особенности, а разрабатываемые интерфейсы определяют, насколько гибкой или жесткой должна быть платформа. Тем не менее интеграция любого внешнего языка или библиотеки в собственный исходный код обеспечивает определенный уровень управления состыковкой и зависимостями. В предыдущих примерах переноса надоедливого исходного кода C можно было бы обратиться к материалу предыдущей главы и включить его в сборку, чтобы он был тесно состыкован. Встраивание файла Wasm также тесно состыковано, но модуль изолирован и защищен. Здесь можно выполнять модуль изолированно, а не перебирать систему, чтобы потреблять старый и, возможно, небезопасный исходный код C. В конечном счете цель состоит в рассмотрении возможности полностью уйти от данной библиотеки или языка либо определить, как адаптировать язык к новой системе.

Если вы обнаружите, что сформировали общий шаблон, который может поддерживаться единой средой исполнения, значит, вы нашли возможность портировать или использовать любой язык в своей компании, объединив языки C, C++ и Go для выполнения в одной-единственной среде исполнения Rust Wasm. Или подумайте о том, чтобы написать свою среду исполнения на любом из этих языков и использовать язык Rust для написания модуля! Затем Rust можно использовать в любой другой среде исполнения Wasm, при условии что она соблюдает соответствующие интерфейсы. Описание механизма работы среды исполнения и разработка исходного кода, поддерживаемого средой исполнения, имеют очень абстрактный характер, поэтому продемонстрировать эти концепции поможет наглядный пример. Далее будет написана собственная среда исполнения в виде инструмента командной оболочки, который принимает «поисковые» модули. Интерфейс со средой исполнения будет нуждаться в нескольких функциях, которые инструмент командной оболочки сможет вызывать, и он будет обладать способностью быть гибким и импортировать модули Wasm при запуске приложения. Поэтому внутри инструмента командной оболочки будет работать любой модуль, при условии что тот предоставляет функцию поиска, поддерживающую следующие входные данные: термин, страница и смещение. Затем можно будет смонтировать модули Wasm, которые соблюдают этот интерфейс, без перекомпиляции опорного приложения среды исполнения.

Для начала давайте создадим простой двоичный файл, который обертывает эту функциональность. Назовем его `journal_cli`. Он должен находиться в родительском каталоге, аналогичном проектам `paper_search_lib` и `paper_search`, о которых писалось ранее. Здесь `journal` будет представлять ресурсы, которые, возможно, потребуется собрать в будущем. Этот инструмент будет использовать файл Wasm для обеспечения возможностей поиска и позволит выполнять приложение без его пересборки.

Листинг 10.19 Команда: создание нового двоичного и библиотечного файлов

```
cd ..  
cargo new journal_cli
```

Нужно находиться в родительском каталоге (`journal`) на одном уровне с проектом `paper_search` и в том же каталоге, что и файл `Cargo.toml` рабочего пространства.

Следующую ниже библиотеку требуется исключить из других рабочих пространств, так как она потребляет файл Wasm. Для этого нужно отредактировать файл `Cargo.toml` родительского уровня.

Листинг 10.20 Cargo.toml: исключение библиотеки

```
[workspace]  
  
members = [  
    "paper_search_lib",  
    "paper_search"  
]  
  
exclude = ["journal_cli"]
```

Перейдите в каталог `journal_cli` и добавьте следующую ниже библиотеку в файл `Cargo.toml`.

Листинг 10.21 journal_cli/Cargo.toml: добавление SDK

```
[package]  
name = "journal_cli"  
version = "0.1.0"  
edition = "2021"  
  
[dependencies]  
wasmedge-sdk = "0.11.2"
```

Единственная необходимая зависимость – это комплект инструментов разработки (SDK) для WasmEdge. Указанный комплект позволит создать виртуальную среду для выполнения файла Wasm и предоставит функции, необходимые для взаимодействия с модулем Wasm. Можно начать думать об этом как о собственной версии двоичного файла WasmEdge, который использовался ранее для вызова нашего первого модуля Wasm.

Но вместо того, чтобы `wasmedge` обеспечивал взаимодействие с модулем, будут определены способы обработки входных и выходных данных. Следующий ниже исходный код немного перегружен, но большая его часть настраивает виртуальную среду Wasm под выполнение в ней модуля. Процесс работы приложения состоит в том, что сначала отыскивается и загружается файл Wasm, а затем создается виртуальная среда. После того как файл будет выбран и загружен в виртуальную среду, будет вызвана функция, которая исполняет Wasm как исполняемый файл.

Откройте файл `main.rs` и добавьте следующий ниже исходный код.

Листинг 10.22 journal_cli/src/main.rs: вызов функции main в Wasm

```

use std::env;
use std::path::PathBuf;
use wasmedge_sdk::{
    config::{CommonConfigOptions,
        ConfigBuilder,
        HostRegistrationConfigOptions},
    params, VmBuilder, WasmVal
};

fn main() -> Result<(), Box<dyn std::error::Error>>{
    let mut args: Vec<String> = env::args().skip(1).collect();
    args.reverse();
    let target = args.pop().unwrap_or(
        "paper_search".to_string());

    let filename = format!("{}.wasm", target);
    let wasm_file: PathBuf = [
        "..", "target",
        "wasm32-wasi",
        "debug", filename.as_str()]
        .iter()
        .collect();           ▶ Динамически загружает созданный
                            ранее файл Wasm посредством аргументов, что позволяет подменять
                            файлы Wasm в будущем.

    Загружает конфигурацию хоста для виртуальной машины WasmEdge. ▶
    let config = ConfigBuilder::new(CommonConfigOptions::default())
        .with_host_registration_config(
            HostRegistrationConfigOptions::default().wasi(true))
        .build()?;
    assert!(config.wasi_enabled());           ▶ Создает конфигурацию для загрузки WASI.

    let mut vm = VmBuilder::new().with_config(config).build()?;
    vm.wasi_module_mut()
        .expect("Not found wasi module")           ▶ Создает новую виртуальную машину с конфигурацией, позволяющей загружать файл Wasm.
        .initialize(None, None, None);
    vm.register_module_from_file(target.as_str(), &wasm_file)?
        .run_func(Some(target.as_str()), "_start", params!());           ▶ Вызывает функцию main без передачи параметров. Функция аналогична вызову wasmedge hello-world.wasm из терминала.

    Ok(())
}

```

Теперь зайдите в каталог `journal_cli`, наберите команду `cargo run`, и вы должны увидеть, как всплынут результаты. Среда исполнения `journal_cli` исполнила опорную «главную» функцию в модуле точно так же, как при непосредственном вызове двоичного файла WasmEdge с помощью `wasmedge paper_search.wasm`.

Разработанный инструмент командной оболочки показывает, как много заложено в то, что WasmEdge приобрел в своей конкретно-прикладной среде исполнения, и как начать строить свои собственные. Но, что интересно, печатаемое на экране не является результатом работы исходного кода инструмента командной оболочки, а посту-

пает из модуля Wasm. Созданная среда исполнения никоим образом не захватывает и не манипулирует этими выходными данными; вместо этого она предлагает модулю Wasm механизм (через настроенную виртуальную машину), который предоставляет функциональность функции `print`, которая, в свою очередь, обращается к `STDOUT`. Виртуальная среда делает для нас многое, и средой исполнения можно манипулировать, чтобы по желанию менять поведение системы. Представьте себе сценарий, в котором требуется использовать тот же модуль поиска, но так, чтобы виртуальная среда писала результат в файл или его сжимала.

А что делать с написанной ранее библиотекой? Как получить к ней доступ? Исходный код опять-таки нуждается в небольшой доработке, однако нужно только поменять `"_start"` на `"static_search"`, так как это имя функции, и можно включить параметры.

Листинг 10.23 journal_cli/src/main.rs: вызов функции main в Wasm

```
fn main() -> Result<(), Box<dyn std::error::Error>>{
    ...
    vm.register_module_from_file(
        target.as_str(), &wasm_file)?
        .run_func(
            Some(target.as_str()),
            "static_search", params!(0, 1))?;
    Ok(())
}
```

Здесь используется макрокоманда `params!`, чтобы закодировать значения, которые передаются в функцию поиска.

После изменения цели выполнения теперь можно получить доступ к методу поиска, который использовался ранее. Функциональность остается прежней, что и при использовании флага `--reactor` ранее в WarmEdge. Вместо применения метода `main` теперь функция вызывается непосредственно в модуле. Теперь есть среда исполнения и подлежащий выполнению модуль Wasm, но давайте добавим еще один модуль, чтобы продемонстрировать возможности написания собственной среды исполнения и гибкость, которую она предлагает.

Прежде чем двигаться дальше, вернем исходный код из `static_search` в `_start`, поскольку именно так будет вызываться следующий файл Wasm.

Листинг 10.24 journal_cli/src/main.rs: вызов функции main в Wasm

```
fn main() -> Result<(), Box<dyn std::error::Error>>{
    ...
    vm.register_module_from_file(target.as_str(), &wasm_file)?
        .run_func(Some(target.as_str()), "_start", params!());
    Ok(())
}
```

10.5 Еще о Wasm

Файл Wasm загружается и регистрируется в виртуальной среде исполнения, определенной комплектом SDK для WasmEdge, что позволяет защищенно исполнять код и создает барьер между разработанным приложением командной оболочки и библиотекой поиска, предоставленной в модуле Wasm. Вследствие этого появляется возможность подменить файл Wasm другим, не меняя опорный код инструмента командной оболочки, указав на другой модуль Wasm. В целях демонстрации давайте создадим еще одну библиотеку быстрого поиска и протестируем эту функциональность. Вместо поиска статей, как это делалось в предыдущем разделе, будем искать книги. При этом будет реализован тот же тип функциональности поиска, что и в библиотеке поиска статей. Затем этот пример можно будет использовать для демонстрации подмены модулей Wasm без изменений в инструменте командной оболочки, как показано на рис. 10.5.

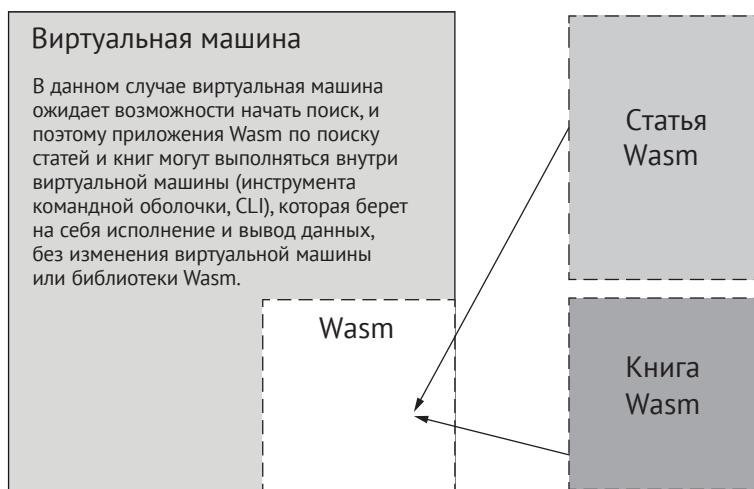


Рисунок 10.5 Wasm, выполняющий дополнительные модули

Давайте вернемся к родительскому каталогу `journal` и создадим еще одну библиотеку Wasm под названием `book_search`. Здесь будет собрана библиотека, аналогичная библиотеке `paper_search`.

Листинг 10.25 Команда: создание нового двоичного файла

```
cd ..  
cargo new book_search
```

Вы должны находиться в корневом каталоге проекта (`journal`).

Откройте родительский файл `Cargo.toml` и добавьте новую библиотеку.

Листинг 10.26 journal/Cargo.toml: включение новой библиотеки

```
[workspace]

members = [
    "paper_search_lib",
    "paper_search",
    "book_search"   ← Добавляет библиотеку к элементам рабочего пространства.
]

exclude = ["journal_cli"]
```

В файл `book_search/Cargo.toml` будет добавлено несколько библиотек, похожих на поиск статей.

Листинг 10.27 book_search/Cargo.toml: зависимости для поиска книг

```
[package]
name = "book_search"
version = "0.1.0"
edition = "2021"

[build]
target="wasm32-wasi"
[target.wasm32-wasi]
runner = "wasmedge"

[dependencies]
tokio_wasi = { version = "1.21", features = ["rt", "macros", "net",
"time"]}
reqwest_wasi = "0.11"
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
```

Самое большое отличие заключается в том, что в поиске статей используется XML, а здесь после выполнения запроса будет приниматься JSON. Опять же, задействуется другая деловая логика и создаются API-интерфейсы, которые соответствуют обобщенному инструменту поиска. Исходный код должен быть довольно простым. Сначала определяются элементы объекта JSON, которые, как ожидается, будут возвращаться из запроса.

Листинг 10.28 book_search/src/main.

rs: ключевые сущности поиска книг

```
use serde::{Deserialize, Serialize};
use std::env;
use std::error::Error;
use std::fmt::{self, Debug, Display, Formatter};

#[derive(Default, Debug, Clone, PartialEq, Serialize, Deserialize)]
pub struct SearchResult {
```

```

    pub results: Vec<Book>,
}

#[derive(Default, Debug, Clone, PartialEq, Serialize, Deserialize)]
pub struct Book {
    pub id: i32,
    pub title: String,
}

```

После этого напишем очень похожую функцию `main`, которая вызывает функцию поиска и печатает результаты.

Листинг 10.29 book_search/src/main.rs:

функция main для поиска книг

```

fn main() -> Result<(), Box<dyn std::error::Error>> {
    let mut args: Vec<String> = env::args().skip(1).collect();
    args.reverse();
    let term = args.pop().unwrap_or("rust".to_string()); ← Извлекает термин из входного аргумента.

    let res: Vec<String> = search(term).unwrap(); ← Выполняет поисковый запрос с термином.

    for entry in res.iter() { ← Печатает результаты для каждого значения в STDOUT.
        println!("{}", entry);
    }
}

Ok(())
}

```

Далее следует копия функции поиска.

Листинг 10.30 book_search/src/main.rs: http-вызов поиска книг

```

pub fn search(
    term: String
) -> Result<Vec<String>, Box<dyn std::error::Error>> {
    let rt = tokio::runtime::Builder::new_current_thread()
        .enable_all() ← Асинхронное выполнение не требуется, поэтому вызов будет заблокирован. Для этого нужен поток исполнения.
        .build()
        .unwrap();

    let searchresult: SearchResult = rt.block_on(async {call_api(term)
        .await}).unwrap(); ← Блокировка этого потока будет продолжаться до тех пор, пока не придет ответ от API.

    let res = searchresult
        .results
        .into_iter()
        .map(|e| format!("{}", e.title))
        .collect::<Vec<String>>(); ← Форматирует значения книг в строковые литералы.

    return Ok::<Vec<String>, Box<dyn std::error::Error>>(res);
}

pub async fn call_api(term: String) -> Result<SearchResult,
reqwest::Error> {
    let http_response = reqwest::get(format!(
        "http://gutendex.com/books/?search={}",

```

```

    term
  )) .await?;   ↗ Вызывает API поиска книг с заданным
    let b = http_response.text().await?;   ↗
    let res: SearchResult = serde_json::from_str(b.as_str()).unwrap();   ↗ СерIALIZУЕТ ВЫХОДНЫЕ ДАННЫЕ
    return Ok(res);   ↗ В РАНЕЕ ОПРЕДЕЛЕННЫЕ СТРУКТУРЫ.
}

```

Выглядит знакомо, не правда ли? Определения JSON предоставляются и десериализуются из результатов поиска книг. Давайте создадим файл Wasm, а затем протестируем вызов инструмента командной оболочки.

Листинг 10.31 Команда: создание файлов Wasm

```

cd ..
cargo build --target wasm32-wasi   ↗ Должны быть в корневом каталоге
                                    проекта (journal).
Компилирует все целевые объекты
Wasm (paper_search и book_search).

```

Теперь вернемся к инструменту командной оболочки (CLI). Поскольку он был сделан динамичным в зависимости от того, какую библиотеку он использует, должна иметься возможность выполнить команду, но передать новое имя библиотеки.

Листинг 10.32 Команда: поиск книг

```

cd journal_cli
cargo run book_search

```

При исполнении кода вы должны увидеть несколько результатов, которые отличаются от результатов из библиотеки `paper_search`, потому что инструмент командной оболочки загружает другой модуль Wasm, как раньше. Передача в `paper_search` должна дать результаты из модуля Wasm `paper_search`, который использовался ранее. Выполнив определение простого API, заданное созданным объектом `vm`, можно изменять опорное поведение инструмента командной оболочки без перекомпиляции. Он загружает модуль и вызывает нужную функцию, чтобы выполнить поиск. Каждая опорная функция работает одинаково, но вызывает совершенно разные конечные точки в совершенно разных форматах. Прямо сейчас все это работает, потому что модули Wasm исполняются напрямую, но что, если потребуется использовать опорные функции, как это делалось ранее? Для этого нужно будет углубиться в память Wasm.

10.6 Память Wasm

Для начала нужно рассказать о том, как Wasm и WASI управляют памятью. Как уже упоминалось ранее, Wasm функционирует в *изолированной* среде, имея в виду, что доступ к аппаратному обеспечению и службам реального компьютера осуществляется с помощью опорной виртуальной

ной машины. Вдобавок каждый модуль Wasm управляет памятью внутри модуля. Благодаря этому виртуальная машина способна добираться до модуля, чтобы писать по адресу и читать из адреса памяти. Это предполагает, что виртуальная машина отвечает за чтение данных из модуля, не опасаясь, что исполнение внутри модуля повлияет на опорную систему. Также не нужно беспокоиться о том, что несколько модулей Wasm могут захватывать память или манипулировать ею в другом модуле. Ответственность за это ложится на виртуальную машину, что придает чрезвычайную важность конструктивному исполнению вашей виртуальной машины.

Структура памяти Wasm представляет собой простой [AggauBuffer](#) с возможностью динамического изменения размера, в котором хранятся необработанные байты данных. Как уже упоминалось ранее, в стандарте Wasm есть несколько примитивных типов, но ни один из них не является строковым или символьным примитивом. Их значения варьируются от системы к системе и часто принимают форму байтовых данных, которые, опять же, варьируются от машины к машине. Тем не менее вы, возможно, помните, что Wasm поддерживает пятое значение: [v128](#), или вектор с 128 битами; виртуальная машина использует его для таких данных, как строковые литералы. Память модуля может увеличиваться и изменяться с помощью различных инструкций памяти или среды исполнения в рамках хоста.

Для того чтобы использовать память внутри модуля, нужна способность резервировать память, извлекать ее местоположение, писать в эту ячейку памяти, выполнять функцию и читать результаты из местоположения в памяти. Подобные низкоуровневые операции обычно выходят за рамки диапазона возможностей многих разработчиков, поэтому это может показаться немного утомительным, но помните, что здесь строится виртуальная машина, и поэтому потребуется написать немного низкоуровневого системного кода для управления памятью. Вместе с тем, для того чтобы смягчить необходимость переписывать эти функции и сократить повторяющийся код, можно создать библиотеки и инструменты. Это сложно, потому что Wasm – это язык ассемблера, который работает на конкретном целевом объекте, не имея понятия об опорной архитектуре машины, на которой он работает. Способ представления значения в памяти приложением или программой может отличаться в зависимости от 64- или 32-битовой машины, на которой они выполняются. Wasm не пытается соответствовать всем этим разным вариантам использования, потому что он должен быть простым и низкоуровневым.

Вызывающее приложение и сам файл Wasm должны договариваться о том, как резервировать память под данные и как читать их из памяти. При запуске каждого экземпляра файла Wasm для модуля будет резервироваться определенный объем памяти, при этом указатель на эти данные будет известен только этому модулю. Память модуля используется коллективно виртуальной машиной и модулем, как показано на рис. 10.6.

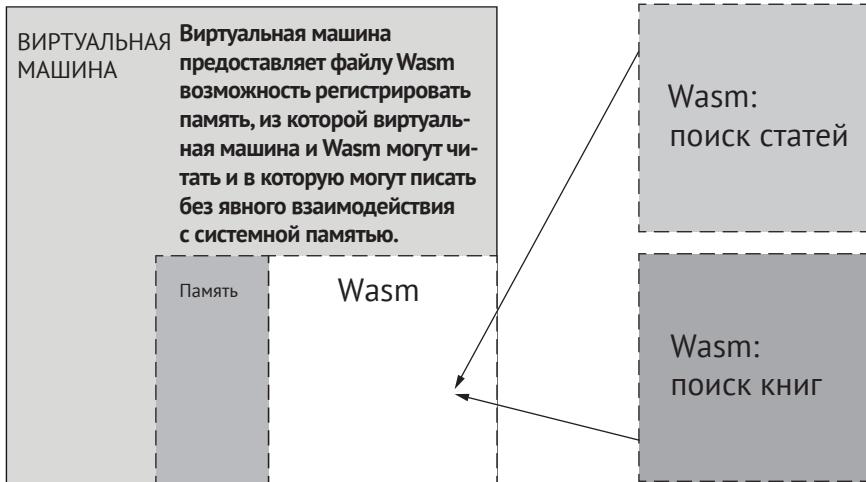


Рисунок 10.6 Память Wasm

Модуль должен предоставить способ резервирования памяти и возвращения вызывающей функции указателя на адрес. Давайте добавим этот метод в исходный код. Откройте файлы `paper_search/src/main.rs` и `book_search/src/main.rs` и добавьте следующее ниже.

Листинг 10.33 [paper_search|book_search]/src/main.rs:
функция резервирования памяти

```
#[no_mangle]
pub extern fn allocate(size: usize) -> *mut c_void {
    let mut buffer = Vec::with_capacity(size);
    let pointer = buffer.as_mut_ptr();
    mem::forget(buffer);

    pointer as *mut c_void
}
```

← Резервирует буфер на основе размера, предоставленного вызывающей службой.

← Очищает его содержимое.

← Находит указатель в линейной памяти.

Возвращяет значение указателя.

Теперь, как показано на рис. 10.7, хост-приложение может резервировать конкретный объем памяти для всего, что ему нужно передать.

Наличие этого инструмента позволит передавать строковый литерал в функцию и делать так, чтобы функция писала результаты обратно в память, откуда вызывающая функция может затем извлекать данные. Для этого нужно открыть эту функциональность в среде исполнения в рамках инструмента командной оболочки. В обязанности среды исполнения входит выставление памяти модулю и предоставление способности читать из этой памяти и писать в нее. Итак, откройте `journal_cli/src/main.rs` и перепишите функцию `main`.

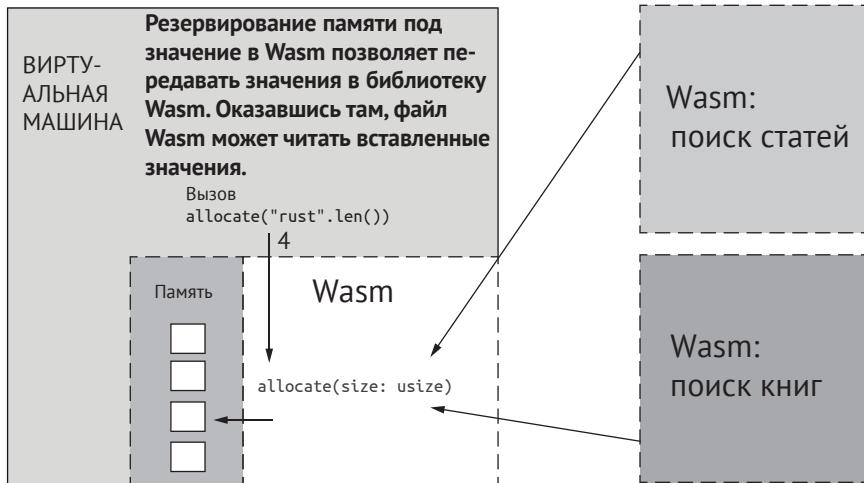


Рисунок 10.7. Распределение памяти в Wasm

Листинг 10.34 journal_cli/main.rs: переписывание вызова функции из инструмента командной оболочки

```
fn main() -> Result<(), Box<dyn std::error::Error>>{
    ...
    let term = args.pop().unwrap_or("type".to_string());
    let config = ConfigBuilder::new(CommonConfigOptions::default()
        .with_host_registration_config(HostRegistrationConfigOptions::default
            .wasi(true)) Создает конфигурацию с включенной опцией wasi.
        .build()?;
    assert!(!config.wasi_enabled()); Строит новую виртуальную машину точно так же, как это делалось ранее.
    let mut vm = VmBuilder::new().with_config(config).build()?;
    vm.wasi_module_mut() На этот раз требуется извлечь объект-модуль.
        .expect("Not found wasi module")
        .initialize(None, None, None);
    let m = vm.clone()
        .register_module_from_file(target.to_string(), &wasm_file)?;
    let env_instance = m.named_module(target.to_string())?;
    let exec = vm.executor(); Используя модуль, извлекает среду для захвата функций и памяти.
    Извлекает исполнителя для виртуальной машины.
    let mut memory = env_instance.memory("memory")?;
    let allocate = env_instance.func("allocate")?; Извлекает объект-память из среды.
    Извлекает из среды функцию резервирования памяти.
    let search = env_instance.func("memory_search")?;
    let term_len: i32 = term.len() as i32;
```

```

let iptr = allocate.run(exec, params!(term_len)?[0].to_i32()); ←
let uptr: u32 = iptr as u32;
memory.write(term, uptr); ←
Пишет термин
в память.

let iresptr = search.run(exec, params!(iptr)?[0].to_i32()); ←
let uresptr: u32 = iresptr as u32;
let val = memory.read_string(uresptr, 1024)?; ←
let val = val.trim_matches(char::from(0));
println!("{}:", val); Читает возвращаемое стро-
Ok(())
}

```

Выполняет функцию резер-
вирования памяти с длиной
поискового термина.

Выполняет
функцию по-
иска с ука-
зателем.

Читает возвращаемое стро-
ковое значение из памяти.

Как вы видите, все это настраивается немного сложно, но добрая часть из приведенного выше уже встречалась ранее. Для доступа к модулям памяти нужна более сложная виртуальная машина, поэтому пришлось развернуть еще несколько инструментов. Беря экземпляр нашего модуля, можно извлекать функции и абстракции, такие как память. Затем резервируется пространство, необходимое для передачи значения. Вызов функции резервирования памяти обеспечивает возможность записи в память без переполнения, как показано на рис. 10.8. Сразу после загрузки значения можно вызвать функцию поиска (которую еще предстоит написать!) и дождаться указателя на то место, где написан ответ. Мы читаем 1 Кб данных и печатаем результаты из хоста. Данные ответа можно возвращать более динамичными и сложными способами, но здесь они рассматриваться не будут.

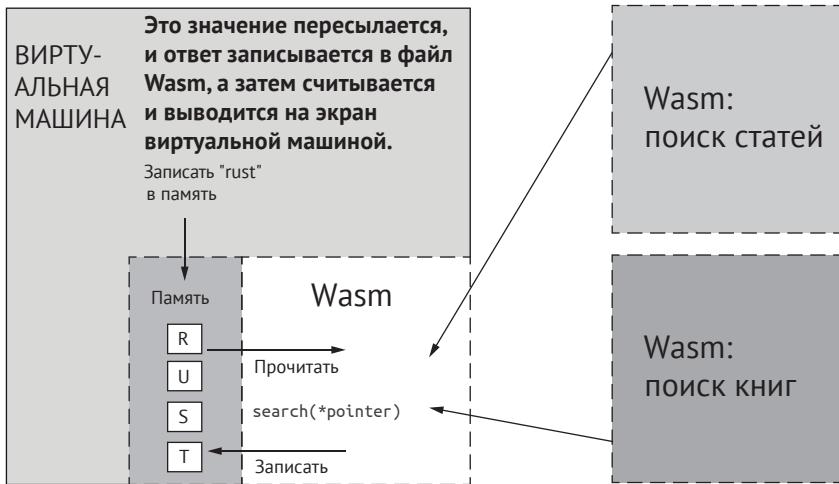


Рисунок 10.8 Операции Wasm по чтению и записи в буфер памяти

Если это выполнить, то из библиотеки Wasm будет получено сообщение об ошибке во время исполнения, потому что еще не написана функция поиска. Так что давайте сделаем это для поиска как статей, так и книг. Начиная с `paper_search/src/main.rs` будет добавлен новый метод поиска в памяти.

Листинг 10.35 paper_search/src/main.rs:
добавление поиска в памяти

```
use std::ffi::{CStr, c_char, CString};

#[no_mangle]
pub fn memory_search(term: *mut c_char) -> *mut c_char {
    let t = unsafe { CStr::from_ptr(term).to_bytes().to_vec() }; ◀
    let mut output = t.to_vec();
    let search_term: String = String::from_utf8(output).unwrap();
    let res_string = search(search_term, 0, 1).unwrap();
    let mut res: Vec<u8> = res_string.into_iter().nth(0).unwrap().into(); ◀
    res.resize(1024, 0); 3((C022-3)) ◀
    unsafe { CString::from_vec_unchecked(res)}.into_raw() ◀
}

Расширяет вектор до 1 Кб для результата.
```

Читает строковый литерал термина из памяти.

Вызывает функцию поиска с термином.

Получает ответ через указатель на вектор.

Эта функция получит указатель, который был передан из виртуальной машины и который был возвращен как составная часть вызова функции резервирования памяти. Затем результирующий строковый литерал читается из памяти и передается в функцию поиска. В данном случае интересует только одно значение, поэтому результаты лимитируются и из вектора извлекается значение. Затем оно пишется в ту же ячейку памяти, просто увеличивая размер до 1 Кб. По завершении нужно вернуть новый указатель на случай, если адрес памяти будет перемещен. Аналогичная функция будет написана для `book_search`.

Листинг 10.36 book_search/src/main.rs:
добавление поиска в памяти

```
use std::ffi::{CStr, c_char, CString};

#[no_mangle]
pub fn memory_search(term: *mut c_char) -> *mut c_char {
    let t = unsafe { CStr::from_ptr(term).to_bytes().to_vec() }; ◀
    let mut output = t.to_vec();
    let search_term: String = String::from_utf8(output).unwrap();
    let res_string = search(search_term).unwrap();
    let mut res: Vec<u8> = res_string.into_iter().nth(0).unwrap().into(); ◀
    res.resize(1024, 0); 3((C023-3)) ◀
    unsafe { CString::from_vec_unchecked(res)}.into_raw() ◀
}

Расширяет вектор до 1 Кб для результата.
```

Читает строковый литерал термина из памяти.

Вызывает функцию поиска с термином.

Получает ответ через указатель на вектор.

Эти функции практически одинаковы: единственное отличие состоит в том, что функция поиска в этой библиотеке не поддерживает разбивку на страницы. Поскольку написаны обе функции и обновлен

инструмент командной оболочки, можно перекомпилировать модули Wasm и их протестировать! Из корневого каталога проекта выполните следующее.

Листинг 10.37 Команда: сборка файлов Wasm

```
cd ..  
cargo build --target wasm32-wasi
```

Корень проекта (journal).
Перекомпилирует целевые объекты Wasm.

Затем измените каталоги на `journal_cli` и выполните поиск.

Листинг 10.38 Команда: поиск книг

```
cargo run book_search rust  
cargo run paper_search rust
```

И вы должны увидеть результаты! Хотя этот раздел кода вокруг управления памятью выглядит достаточно сложным, вы всегда можете написать коллективную библиотеку, которая упростит этот процесс или будет использовать библиотеку, которая реально помогает. Экосистема Wasm быстро меняется, и появляются различные инструменты и библиотеки, которые помогают в этом процессе. К сожалению, приведение их здесь привело бы лишь к устареванию списка. Несмотря на это, будем надеяться, что вы сможете по достоинству оценить гибкость, которую обеспечивает Wasm.

10.7 Это только начало

В отличие от языков C, C++, Python и JavaScript, WebAssembly находится на переднем крае технологий, и стандарты продолжают разрабатываться и изменяться. То, что было обследовано в последних двух главах, задействует современные технологии, но с уже сформировавшимися стандартами Wasm. Мы только начинаем осваивать возможности этой технологии, и со временем она будет оказывать все большее влияние. WebAssembly включается в процесс переработки кодовой базы, так как он наглядно показывает, насколько хорошо язык Rust подходит для переработки практически всего, что существует на сегодняшний день. Возможно, это не всегда просто, но зато доступна масса инструментов, которые позволяют перемещать исходный на язык Rust или, по меньшей мере, помогают вкладывать Rust в исходный код.

Краткий итог

- WASI – это стандарт по интеграции модулей Wasm в универсальную среду исполнения, что продемонстрировано посредством комплекта инструментов разработки (SDK) для WasmEdge и среды исполнения.

- WasmEdge является одной из реализаций этой среды исполнения и предоставляет комплект инструментов разработки для написания приложений, использующих Wasm, таких как созданный нами инструмент поиска на основе командной оболочки.
- Система типов Wasm обеспечивает определение векторов и ручное управление памятью, что позволяет использовать инструменты, обеспечивающие более высокий уровень гибкости, без использования сложной системы типов.
- Среда исполнения Wasm предоставляет коллективный изолированный модуль линейной памяти для виртуальной машины и модуль, который можно использовать для обеспечения безопасной среды исполнения.
- Для того чтобы использовать модель памяти, необходимо создать функции резервирования и высвобождения памяти в модуле, чтобы обмениваться сообщениями с заданной виртуальной машиной.

Предметный указатель

/calculate, конечная точка 145
&str, тип 56
#[test], атрибутная макрокоманда 235

А
Аннотация ко времени жизни
 в плагине веб-сервера NGINX 158
Архив Java (JAR) 305

Б
Байт-код 305
Библиотека-калькулятор 162
Библиотека, структурирование
 пути 186
Библиотека, структурирование
 восходящая видимость 202
Блокировка 268
Блок кода незащищенный 90
Браузер
 запрос данных на языке Rust 286
 из браузера на машину 312
 компиляция в Wasm с помощью языка
 Rust 290
 язык Rust 286
Браузер с использованием языка Rust
 загрузка в браузер 291

В
Взятие значения по указателю 90
Видимость восходящая 202
Владение и заимствование 37
Время жизни 45
 аннотации 57, 152
 контроль
 за мутабельностью 50
 ссылки и заимствование 49
Высвобождение памяти 42

Г
Горлышко бутылочное 265
График времени жизни 45

Д
Деструктуризация 66

З
Заимствование 49

И
Интерфейс двоичный
 приложений (ABI) 306
Интерфейс с внешними
 функциями (FFI) 23, 89, 98, 108, 121
Интерфейс с внешними
 функциями (FFI) 32
Интерфейс системный
 с WebAssembly (WASI) 306
Искажения имен 103

К
Калькулятор
 написание HTTP-ответа 167
Код ассемблерный 305
Код объектный 263
&, команда 266
Комментарий документарный 240
Компилатор 293
Компонент функциональный в serde 210
Конкурентность 272
Консорциум Всемирной
 паутины (W3C) 283

Л
Латание динамическое 254
Литерал строковый мутабельный 58
Литерал строковый неразобранный 246
Лямбда 78

М
Масштабирование 265
 горизонтальное 266
Машина виртуальная (VM) 305
Место узкое 265
Многопоточность исполнения 271
Множество Мандельбрата,
 генерирование на языке Python 262
Модуль 178
 несколько файлов 183
Модуль расширения 214
 пакет pyO3 215
Модуль расширения Python
 написание на языке Rust 214
Мутабельность 50
Мьютекс 275

Н

Нотация польская обратная (RPN) 92, 94

О

Обработка асинхронная 268

Обработка данных на Python 206

>, оператор 160

? , оператор 73, 83

Оператор взятия значения

по ссылке 150

Отбрасывание значения 42

П

Память динамическая 41

Память кучная 42

Память стековая 42

Перезаимствование 150

Переработка исходного кода

из браузера на машину 312

интерфейс Wasm (WebAssembly) для модулей 326

потребление Wasm 320

Переработка исходного кода на язык

Rust 20

когда не следует применять 28

процесс 29

Перераспределение памяти 58

Перечисление 62

единичный тип 69

обработка ошибок 66

общий обзор 62

поднятие паники при появлении ошибок 80

преобразование ошибок 75

типы ошибки 71

Поток POSIX 273

Поток зеленый 273

Поток исполнения 271

Потребление Wasm 320

Привязка 133

Программирование асинхронное языка Python с языком Rust

PyO3 276

Глобальная блокировка

параллельности интерпретатором 275

Пространство рабочее 313

Псевдоним пути 197

Путь 186

обзор 186

относительный и абсолютный 187, 191

Путь абсолютный 187

Путь относительный 187, 189, 193

Р

Разбор данных в формате JSON 208

Реализация компонента 297

Резервирование памяти 41

Реэкспорт 197

С

Сборка, оптимизированная 229

Сбор мусора 42

Символ вертикальной черты 78

Скрипт сборки 134

Случай тестовый 249

Совместимость

операционная 264, 306, 316

Среда изолированная 329

Среда исполнения 320

Среда исполнения Java (JRE) 305

Среда исполнения

универсальная WASI 308

Срез строковый 159

Ссылка 49

Структурирование библиотек
пути 186

Структурирование библиотек
модулей 178

Т

Тестирование

добавление тестов в существующий

исходный код 245

документарные тесты 240

исходного кода на Rust с

использованием языка Python 250

написание тестов с помощью языка

Rust 234

Тестирование, пакет Rust 234

Тест рандомный 258

Тип единичный 69

Тип строковый 56

мутируемые строковые литералы 58

Тип строковый в языке

мутируемые строковые литералы 58

У

Управление памятью

автоматизированное 42

Управление памятью в других языках 41

Управление памятью ручное 42

Ф

Фонд облачно-ориентированных
вычислений (CNCF) 311

Формат модульный 293

Функция идемпотентная 266

Ц

Цель сборки 131

Ч

Черта общая 119

Я

Язык динамический

- измерение и тестирование производительности в Rust 219
- интеграция с языком Rust, оптимизированные сборки 229

А

ABI (двоичный интерфейс приложений) 306

add, функция 240

admire_art, функция 39, 45, 46, 49, 52

App.jsx, файл 302

args, список 317

art1, переменная 45, 46, 50

Artwork, переменная 46

as_ptr, метод 91

assert_eq!, макрокоманда 235, 237

assert!, макрокоманда 235

asyncio, библиотека Python 268

async, ключевое слово 269

as_, префикс 61

auto-initialize, опция 220

Б

bananas.rs, файл 184

bar, функция 73

bench_py, функция 225

bindgen, инструмент/пакет 139

black_box, функция 223, 227

build_art, функция 55

build-dependencies, секция 139

build-script-test, пакет 135

bundler, опция 293

buy, функция 189

С

связывание с языком Rust 132

calculate

библиотека 133

пакет Rust 162, 163, 164

calculate, библиотека 127

cargo new, команда 37

cargo run, команда 38

Cargo.toml, файл 164, 313, 323

cdylib 164

cdylib, пакет 101, 102, 144

cfg, атрибутная макрокоманда 234

CNCF (фонд облачно-ориентированных вычислений) 311

concat!, макрокоманда 142

configure, скрипт 131

ConsumedTreat, тип 190

Content-Length, заголовок 169

Copy, общая черта 41

crates.io, реестр 163

crate-type, поле 164

crate, ключевое слово 188

create, метод 296

criterion_group!, макрокоманда 222

criterion_main!, макрокоманда 222

Criterion, пакет Rust 219

CStr, вспомогательная структура 106

ctx, переменная 299

Д

day_kind, модуль 185, 193, 197

Debug, общая черта 83, 179

dependencies, секция 99, 163, 221

derive, директива 84

dev-dependencies, секция 221

Display, общая черта 179

Е

eat, функция 190

env!, макрокоманда 142

env, пакет 318

Errort, тип 71, 84

Err, вариант 70, 73, 77

Err, ветвь 74

err, значение 77

evaluate, функция 163

expect, метод 85

extension-module, опция 220

Ф

feed, тер 288

Fetching, режим 298

FetchState, перечисление 299

FFI (интерфейс с внешними

функциями) 126

библиотека-калькулятор 162

написание HTTP-ответа 167

создание модуля веб-сервера

NGINX 128

чтение запросов веб-сервера

NGINX 145

fmt, модуль 178

forest, пакет Rust 199, 203

format!, макрокоманда 62, 167

free, функция 43

G

get_day_kind, функция 185
 get_name, функция 181, 196
 GetSearch, функция 300
 get_value, функция 153
 get, функция 170
 GIL (глобальная блокировка параллельности интерпретатором) 275
 goodbye, функция 181
 GotNegative 85
 greetings, проект 179
 GREET_LANG, переменная среды 136

H

hello, функция 181
 how_was_day, функция 196
 http, модуль 184
 HTTP-ответ 167

I

ignored, строка 224
 import, инструкция 295
 include, директива 139, 140
 include!, макрокоманда 138, 142
 index.html, файл 291
 input.rs, файл 183
 input, модуль 180, 183, 195

J

JAR (Java ARchive) 305
 JavaScript
 Wasm (WebAssembly), веб-компоненты полностью на языке 296
 переработка исходного кода 302
 переработка на языке Rust 285
 создание компонентов React 292
 journal_cli 322, 324
 JRE (среда исполнения Java) 305
 json, модуль Python 225

L

let, инструкция 38
 lib.rs, файл 184, 188, 313
 libsnack, пакет Rust 188
 List, компонент 294
 locals, словарь 226

M

main, функция 46
 main.rs, файл 185, 197, 323
 main_test.py, файл 251
 main, метод 325
 main, функция 48, 49, 50, 62, 180, 194, 243, 318, 319, 325
 malloc, функция 170

memory, битовое поле 170, 173

mod bananas 184
 mod day_kind 185
 mod forest 184
 module/config, файл 128
 module/config, файл 144
 module.rs, файл 183
 mod, инструкция 184
 mod, ключевое слово 179, 182, 184
 MonkeyPatch.context, метод 256
 MonkeyPatch, класс 255
 More, кнопка 295

N

NGINX
 скачивание исходного кода 127
 создание модуля 128
 чтение запросов веб-сервера 145
 по-ор (ничего не делать) 162, 187
 NotDivisible, вариант 65
 NotUnique 76
 num, переменная 66

O

offset_from, метод 162, 171
 offset, метод 171
 openssl, библиотека 133
 output.rs, файл 183, 185
 output, модуль 180, 183, 193

P

package.json, файл 293
 page, переменная 295
 panic!, макрокоманда 80
 panic, функция 82
 post_handler, тип 149
 print_day_kind_message, функция 185
 print_fizzbuzz, функция 62, 70, 73, 80, 82
 printf, функция 39
 println!, макрокоманда 39, 65
 print, функция 325
 pthread 273
 pub use, комбинация ключевых слов 197
 pub, ключевое слово 165, 181, 182, 188
 PyDict, словарь 226
 rymodule, макрокоманда 278
 pyO3, пакет Rust 215
 py_result, переменная 257
 py.run, функция 227
 pytest, файл 251
 Python
 асинхронный 268
 асинхронный с языком Rust 276

- генерирование множества
Мандельброта 262
обработка данных 206
планирование перемещения на языкок
Rust 207
тестирование исходного кода Rust 250
`python_sum`, функция 256
Python асинхронный
 масштабирование 265
 многопоточность исполнения 271
Python асинхронный с языкок Rust
 PyO3 276
 глобальная блокировка параллельности
 интерпретатором 275
- R**
- `randomized_test_case`, функция 258
React, создание компонентов 292
`read_body_handler`, функция 145, 147,
 166, 172
`regret`, функция 190
`request`, переменная 158
`reqwest`, библиотека 288
`Result`, тип 67, 70, 73, 75, 85
`rlib` 164
RPN (обратная польская нотация) 92, 94
Rust 36
 Wasm (WebAssembly), веб-компоненты
 полностью на языкке 296
 асинхронный Python 261
 в браузере 286
 владение и заимствование 37
 времена жизни 45
 измерение и тестирование
 производительности 219
 интеграция с динамическими языкоками,
 оптимизированные сборки 229
 переработка JavaScript
 на языкок Rust 285
 перечисления и обработка ошибок 66
 планирование перемещения
 с языкок Python 207
 связывание языкок С с языкок Rust 132
 структуривание библиотек 186
 строковые типы в языкке 56
 тестирование интеграций 233
 тестирование исходного кода с
 использованием языкок Python 250
 управление памятью
 в других языкоках 41
`rust_json.sum`, конкретно-прикладной
 метод 225
- `rust_json`, библиотека 226
`rust-json`, виртуальная среда 250
- S**
- `script`, тер 293
Search, структура 290
`serde`, библиотека 290
`set`, функция 170
`shop`, модуль 188
`solve`, функция 133, 165
`std`, пакет 178
`String`
 `with_capacity` 59
`String`, тип 69
`sum`, функция 252, 255
`super`, отрезок пути 188, 193
`switch`, инструкция 64
- T**
- `tests`, модуль 234, 237, 240
`test`, модуль 245
`ThreadPoolExecutor`, класс 275
`to_string()`, метод 39
`toString`, метод 84
`to_`, префикс 61
`treats`, модуль 188
`Treat`, тип 190
`trim`, метод 194
`try/except`, блок 66
- U**
- `u8`, значения 221
`u128`, значения 221, 223
`unchecked` 89
`unsafe` 89
`update`, метод 296
`useEffect`, перехватчик 295
`UsernameError`, тип 76
`use`, ключевое слово 182
`usize`, тип 162
- V**
- `v128` 330
`validate_lowercase`, функция 76
`validate_unique`, функция 76
`validate_username`, функция 76, 79
`value`, переменная 155
`view`, метод 296
`View`, этап 298
`VM` (виртуальная машина) 305
`void`, функция 70
- W**
- W3C (World Wide Web Consortium) 283

WASI (системный интерфейс с WebAssembly) 306
wasm_bindgen, библиотека 290
Wasm (WebAssembly) 283
библиотека 318
веб-компоненты полностью на языке 296
загрузка в браузер 286
из браузера на машину 312
интерфейс для переработки исходного кода 304
модули 326
память 329
переработка исходного кода JavaScript 302

потребление 320
создание компонентов React 292
универсальная среда исполнения WASI 308

welcome, функция 43
with_gil, функция 226

X

x, параметр 79
x, символьная переменная 90

Y

y, символьная переменная 90

Книги издательства «ДМК Пресс» можно купить оптом и в розницу
на складе издательства по адресу:
Москва, ул. Электродная, д. 2, стр. 12, офис 7,
тел. **+7 (499) 322-19-38**,
а также заказать на сайте **www.dmkpress.com**
с доставкой в любой регион РФ.

Лили Мара, Джоэл Холмс

**Переход на Rust.
Рефакторинг исходного кода с других языков**

Главный редактор *Мовчан Д. А.*
Зам. главного редактора *Яценков В. С.*
editor@dmkpress.com

Перевод *Логунов А. В.*
Корректор *Синяева Г. И.*
Верстка *Луценко С. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «NewBaskervilleC». Печать цифровая.
Усл. печ. л. 27,95. Тираж 100 экз.

Веб-сайт издательства: www.dmkpress.com

Эта книга научит вас расширять функциональность и повышать производительность приложений за счет поэтапного переноса кодовой базы на Rust с таких языков как C, C++, Python и JavaScript. Вы узнаете, как создавать автономные библиотеки Rust, наращивать скорость и стабильность работы и создавать защищенный, эффективный по потреблению памяти низкоуровневый исходный код. Книга начинается с рассмотрения уникального синтаксиса и концепций языка Rust. Далее авторы поясняют, как использовать Rust для обертывания опасного исходного кода, вызывать стандартные и прикладные библиотеки и даже применять формат байт-кода Wasm для исполнения кода на языке Rust в браузере.

*Для программистов среднего уровня.
Опыт работы с языком Rust не требуется.*

Вы научитесь:

- создавать библиотеки языка Rust, доступные для вызова из других языков;
- обрабатывать ошибки как значения, используя перечисления языка Rust;
- оптимизировать код для экономного использования памяти;
- повышать производительность с помощью механизма параллельного выполнения языка Rust.

Лили Мара – инженер-программист, специализирующийся на высокопроизводительных приложениях на языке Rust.

Джоэл Холмс – разработчик программного обеспечения, занимающийся разработкой облачно-ориентированных приложений.

«Никаких излишеств. В каждой главе чувствуется воодушевление, pragmatism и направляющая рука авторов».

Бастиан Грубер, Mozilla

«Устранение сложности не означает упрощение задачи. Это означает, что решение должно быть элегантным и доступным. В данной книге этот принцип применяется наивысшим образом».

Михал Рутка,

*внештатный разработчик
и архитектор облачных решений*

«Это настоящая Библия для понимания ключевых принципов языка Rust и методов переработки».

Самбасива Андалури, Oracle



ISBN 978-5-93700-228-0



9 785937 002280 >