O'REILLY®

«Данная книга займет достойное место на книжной полке любого пишущего на Rust программиста. Она поможет преодолеть весьма крутую кривую обучения, особенно в части владения и времени жизни. Больше всего мне нравятся работающие примеры и глубокое рассмотрение реальных задач.»

*— **Раф Левьен,** Google*

Rust — новый язык системного программирования, сочетающий высокую производительность и низкоуровневый контроль, характерные для C и C++, с безопасной работой с памятью и потоками. Современная гибкая система типов Rust препятствует появлению в программах таких ошибок, как разыменование нулевого указателя, двойное освобождение, висячий указатель и прочих, причем все проверки производятся на этапе компиляции, избавляя программу от накладных расходов на этапе выполнения. В многопоточной программе компилятор Rust обнаруживает гонки за данными, благодаря чему писать конкурентный код становится гораздо проще. В этой книге, написанной двумя опытными системными программистами, объясняется, как Rust смог навести мост между производительностью и безопасностью, и как вы можете воспользоваться этим для своих целей.

Прочитав эту книгу, вы узнаете:

- как Rust представляет значения в памяти;
- все о владении, передаче владения, заимствовании и времени жизни;
- argo, rustdoc, автономные тесты и публикация кода на сайте crates.io, репозитории Rust-пакетов с открытым исходным кодом;
- высокоуровневые средства, благодаря которым Rust является продуктивным и гибким языком: универсальный код, замыкания, коллекции и итераторы;
- конкурентность в Rust: потоки, мьютексы, каналы и атомарные типы — гораздо безопаснее, чем в C и C++;
- небезопасный код и сохранение целостности объемлющего его обычного кода;
- развернутые примеры, демонстрирующие совместную работу всех языковых средств.

Джим Блэнди (Jim Blandy) *работает в проекте Mozilla над средствами разработчика для браузера Firefox. Сопровождал GNU Emacs и GNU Guile, а также GDB — отладчик GNU. Является одним из проектировщиков системы управления версиями Subversion.*

Джейсон Орендорф (Jason Orendorff) *пишет на C++ для проекта Mozilla, в котором является владельцем модуля движка JavaScript в Firefox. Активный член сообщества разработчиков в Нэшвилле, время от времени организует местные встречи технических специалистов.*

Интернет-магазин:
www.dmkpress.com
Книга – почтой:
orders@alians-kniga.ru
Оптовая продажа:
"Альянс-книга"
тел.(499)782-38-89
books@alians-kniga.ru

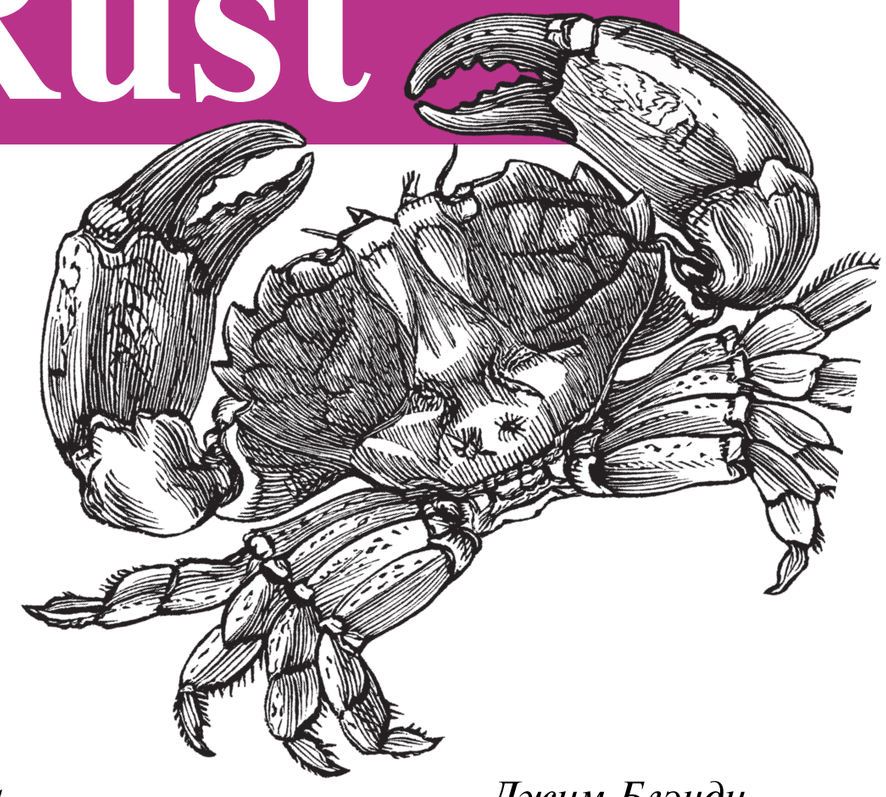
ДМК издательство
www.дмк.рф

ISBN 978-5-97060-236-2
9 785970 602362 >

Программирование на языке Rust

O'REILLY®

Программирование на языке Rust



Джим Блэнди
Джейсон Орендорф

ДМК издательство

Джим Блэнди, Джейсон Орендорф

Программирование на языке Rust

Jim Blandy and Jason Orendorff

Programming Rust

Fast, Safe Systems Development

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Джим Блэнди, Джейсон Орендорф

Программирование на языке Rust

Быстрое и безопасное системное программирование

ДМК Пресс
издательство
Москва, 2018

УДК 004.45:004.438Rust
ББК 32.972.11
Б68

Блэнди Дж., Орендорф Дж.
Б68 Программирование на языке Rust / пер. с анг. А. А. Слинкина. – М.: ДМК Пресс, 2018. – 550 с.: ил.

ISBN 978-5-97060-236-2

Rust – новый язык системного программирования, сочетающий высокую производительность и низкоуровневый контроль, характерные для C и C++, с безопасной работой с памятью и потоками. В начале книги рассмотрены типы данных и основные конструкции языка – выражения, модули, структуры, перечисления и образцы. Далее описываются характеристики и универсальные типы. В следующих главах приводятся сведения о замыканиях и итераторах. Наконец в последних главах книги вы найдете исчерпывающую информацию о коллекциях, обработке текста, вводе-выводе, конкурентности, макросах и небезопасном коде.

Издание предназначено для системных программистов, созревших для поиска альтернативы C++.

УДК 004.45:004.438Rust
ББК 32.972.11

ISBN 978-1-491-92728-1 (анг.)
ISBN 978-5-97060-236-2 (рус.)

Содержание

Предисловие

Язык Rust предназначен для системного программирования.

В наши дни эта фраза требует пояснений, потому что многие современные программисты не знают, что такое системное программирование. Однако же оно лежит в основе всего, что мы делаем.

Вы опускаете крышку ноутбука. Операционная система замечает это, приостанавливает все работающие программы, выключает экран и переводит компьютер в режим ожидания. Позже вы поднимаете крышку: на экран и на все остальные компоненты снова подается питание, и все программы продолжают работу с того места, где остановились. Мы считаем это само собой разумеющимся. Однако системные программисты написали немало кода, для того чтобы все происходило именно так.

Системное программирование предназначено для разработки:

- операционных систем;
- драйверов;
- файловых систем;
- баз данных;
- кода, работающего в очень дешевых устройствах или устройствах, требующих повышенной надежности;
- криптографических приложений;
- мультимедийных кодеков (программ, которые читают и записывают аудио, видео и графические файлы);
- мультимедийных приложений (например, для распознавания речи и редактирования фотографий);
- управления памятью (например, для реализации сборщика мусора);
- отрисовки текста (преобразования совокупности текста и шрифтов в набор пикселей);
- высокоуровневых языков программирования (например, JavaScript и Python);
- средств сетевого программирования;
- средств виртуализации и контейнеров программ;
- математических моделей;
- игр.

Короче говоря, системное программирование – это программирование в условиях *ограниченности ресурсов*, когда каждый байт и каждый такт процессора имеют значение.

Объем системного кода, участвующего в поддержке самого простого приложения, ошеломляет.

Эта книга – не учебник по системному программированию. В ней рассматриваются многочисленные детали управления памятью, которые могут показаться

излишне заумными тому, кто раньше не занимался системным программированием. Но профессиональный системный программист найдет в языке Rust нечто удивительное: новый инструмент, который устраняет серьезные и давно знакомые проблемы, преследовавшие нашу отрасль в течение многих десятилетий.

Для кого написана эта книга

Если вы – системный программист, созревший для поиска альтернативы C++, то эта книга для вас.

Если у вас есть опыт разработки на каком-нибудь языке программирования, будь то C#, Java, Python, JavaScript или еще что-то, то эта книга будет полезна и вам. Однако изучить Rust недостаточно. Чтобы взять от языка максимум, необходимо иметь хоть небольшой опыт системного программирования. Мы рекомендуем читать эту книгу параллельно с реализацией каких-то побочных проектов системного программирования на Rust. Займитесь чем-то таким, чего никогда не делали раньше, чем-то, где могут в полной мере проявиться быстродействие, конкурентность и безопасность Rust. На какие-то идеи может навести приведенный выше перечень.

Зачем мы написали эту книгу

Мы решили написать книгу, которой нам недоставало, когда мы взялись за изучение Rust. Наша цель – четко и ясно представить новые важные идеи Rust во всей полноте, не пренебрегая деталями, чтобы свести к минимуму познание методом проб и ошибок.

Обзор содержания книги

Первые пять глав содержат введение в Rust и знакомят с фундаментальными типами данных, а также с базовыми понятиями владения и ссылки. Эти главы рекомендуется читать последовательно.

В главах 6–10 рассматриваются основные конструкции языка: выражения, обработка ошибок, крейты и модули, структуры, перечисления и образцы. Все читать необязательно, но главу, посвященную обработке ошибок, пропускать не стоит – поверьте нам.

В главе 11 рассматриваются характеристики и универсальные типы – последние две концепции, знать о которых необходимо. Характеристики (trait) похожи на интерфейсы в Java или C#. Кроме того, в Rust это основной способ включения пользовательских типов в язык. В главе 12 показано, как с помощью характеристик поддерживается перегрузка операторов, а в главе 13 рассматриваются многочисленные служебные характеристики.

Понимание характеристик и универсальных типов – ключ к остальной части книги. В главах 14 и 15 описываются замыкания и итераторы – два важнейших средства, без которых вам не обойтись. Прочие главы можно читать в любом порядке или лишь по мере необходимости. В них рассматриваются коллекции, строки и обработка текста, ввод-вывод, конкурентность, макросы и небезопасный код.

Графические выделения

В книге применяются следующие графические выделения:

Курсив
: Новые термины, имена и расширения имен файлов.

`Моноширинный`
: Листинги программ, а также элементы кода в основном тексте: имена переменных и функций, базы данных, типы данных, переменные окружения, предложения и ключевые слова языка.

`Моноширинный полужирный`
: Команды и иной текст, который пользователь должен вводить точно в указанном виде.

`Моноширинный курсив`
: Текст, вместо которого следует подставить значения, заданные пользователем или определяемые контекстом.

Скачивание исходного кода примеров

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

Мы высоко ценим, хотя и не требуем, ссылки на наши издания. В ссылке обычно указываются название книги, имя автора, издательство и ISBN, например: «Программирование на языке Rust» Джима Блэнди, Джейсона Орендорф (O'Reilly, ДМК Пресс). Copyright © 2018 Jim Blandy and Jason Orendorff, 978-1-491-92728-1 (англ.), 978-5-97060-236-2 (рус.).

Если вы полагаете, что планируемое использование кода выходит за рамки изложенной выше лицензии, пожалуйста, обратитесь к нам по адресу dmkpress@gmail.com.

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 прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной

из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Благодарности

Книга, которую вы держите в руках, стала значительно лучше благодаря усилиям официальных технических рецензентов: Брайана Андерсона (Brian Anderson), Мэтта Брабека (Matt Brubeck), Дж. Дэвида Эйзенберга (J. David Eisenberg) и Джека Моффита (Jack Moffit). Было также много неофициальных рецензентов, которые читали ранние варианты и давали ценные советы: Джефф Уолден (Jeff Walden), Джо Уолкер (Joe Walker), Николас Пьеррон (Nicolas Pierron), Эдди Брюэл (Eddy Bruel), Ярослав Шнайдр (Jaroslav Šnajdr), Джеффри Лим (Jeffrey Lim) и Джан-Карло Паскутто (Gian-Carlo Pascutto). Книга по программированию, как и любое другое предприятие в этой области, много выигрывает от сообщений об ошибках. Спасибо.

Мы благодарны корпорации Mozilla и нашим непосредственным руководителям – без них эта работа не состоялась бы.

Спасибо сотрудникам издательства O'Reilly, помогавшим довести этот проект до конца, а особенно нашим редакторам Брайану Макдональду и Джеффу Блэйелу (Jeff Bleiel).

И никакими словами не выразить горячую благодарность нашим женам и детям за их непоколебимую любовь, энтузиазм, терпение, снисходительность и готовность прощать.

Глава 1

Почему появился Rust?

> Есть контексты – например, те, на которые ориентирован Rust, – когда десяти- или хотя бы двукратное превосходство в скорости решает все. От этого зависит судьба программной системы на рынке – точно так же, как на рынке оборудования.
>
> *— Грейдон Хоар*

> В наши дни все компьютеры стали параллельными… Параллельное программирование – это и есть программирование.
>
> *— Майкл Маккул и др.*
> *«Структурное параллельное программирование»*

> Дефект в анализаторе TrueType использовался противниками для шпионажа; безопасность должна быть свойственна любой программе.
>
> *— Энди Уинго*

За те пятьдесят лет, что мы используем языки высокого уровня для написания операционных систем, языки системного программирования прошли долгий путь, но две проблемы оказались особенно трудными.

- Трудно написать безопасный код. Особенно трудно корректно управлять памятью в программах на C и C++. С последствиями – брешами в системе защиты – пользователи сталкиваются на протяжении десятков лет. Началось это с червя Морриса в 1988 году.
- Очень трудно писать многопоточный код, а ведь это единственный способ задействовать возможности современных компьютеров. Даже опытные программисты подходят к многопоточному коду с опаской, поскольку конкурентность может стать причиной новых классов ошибок, а хорошо знакомые ошибки становится гораздо труднее воспроизвести.

Для того и придуман Rust: безопасный конкурентный язык, не уступающий по производительности C и C++.

Rust – новый язык системного программирования, разработанный Mozilla и сообществом. Подобно C и C++, Rust предоставляет средства точного контроля над использованием памяти и поддерживает близкое соответствие между примитивными операциями языка и машинными командами, что позволяет заранее оценить быстродействие написанного кода. Rust ставит перед собой те же цели,

которые Бьярн Страуструп сформулировал для C++ в статье «Abstraction and the C++ machine model»:

> *Вообще говоря, реализации C++ придерживаются принципа нулевых издержек: не платить за то, чем не пользуешься. И более того: то, чем пользуешься, нельзя закодировать вручную более эффективно.*

К этому Rust добавляет еще две цели: безопасная работа с памятью и надежная конкурентность.

Ключом к выполнению обещанного является новаторская система проверяемых на этапе компиляции *владения*, *передачи* и *заимствования*, тщательно спроектированная так, что дополняет гибкую систему статической типизации в Rust. Система владения устанавливает время жизни каждого значения, что делает ненужным сборку мусора в ядре языка и обеспечивает надежные, но вместе с тем гибкие интерфейсы для управления такими ресурсами, как сокеты и описатели файлов. Передача (move) позволяет передать значение от одного владельца другому, а заимствование (borrowing) – использовать значение временно, не изменяя владельца. Поскольку многие программисты раньше не встречались с подобными механизмами, мы подробно остановимся на них в главах 4 и 5.

Те же самые правила владения лежат в основе модели надежной конкурентности в Rust. В большинстве языков связь между мьютексом и данными, которые он защищает, описывается в комментариях; Rust может на этапе компиляции проверить, что программа удерживает мьютекс в течение всего времени, пока обращается к данным. Как правило, язык лишь рекомендует не использовать структуру данных, после того как она передана другому потоку; Rust проверяет, что структура действительно не используется. Rust способен предотвратить гонки за данные на этапе компиляции.

Rust не является настоящим объектно-ориентированным языком, хотя некоторые объектно-ориентированные черты в нем присутствуют. Rust не является функциональным языком, хотя стремится сделать результат вычисления более явно выраженным, как в функциональных языках. В определенной степени Rust напоминает C и C++, но многие идиомы этих языков к Rust неприменимы, так что типичный код на Rust имеет лишь поверхностное сходство с кодом на C или C++. Лучше отложить суждение о том, что же такое Rust, на потом – когда вы освоитесь с языком.

Чтобы оценить дизайн языка в реальных условиях, корпорация Mozilla разработала на Rust новый движок браузера, Servo. Потребности Servo и цели Rust отлично согласуются: браузер должен работать быстро и обрабатывать данные из ненадежных источников безопасно. Servo использует безопасную конкурентность Rust, чтобы положить все ресурсы машины на службу задачам, которые было бы непрактично распараллеливать на C или C++. На самом деле Servo и Rust разрабатывались вместе: в Servo использовались самые последние возможности языка, а Rust эволюционировал с учетом пожеланий разработчиков Servo.

Типобезопасность

Rust – типобезопасный язык. Но что понимается под «типобезопасностью»? Безопасность – это, конечно, хорошо, но от чего именно мы стараемся обезопасить себя?

Ниже приведено определение «неопределенного поведения» из стандарта языка C 1999 года, известного под названием «C99»:

> **неопределенное поведение**
> поведение, являющееся следствием использования непереносимой или некорректной программной конструкции либо некорректных данных, для которого в настоящем Международном стандарте нет никаких требований.

Рассмотрим следующую программу на C:

```
int main(int argc, char **argv) {
  unsigned long a[1];
  a[3] = 0x7ffff7b36cebUL;
  return 0;
}
```

Эта программа обращается к элементу за концом массива `a`, поэтому согласно C99 ее поведение не определено, т. е. она может делать все что угодно. Сегодня утром запуск этой программы на ноутбуке Джима закончился печатью сообщения

```
undef: Error: .netrc file is readable by others.
undef: Remove password or make file unreadable by others.
```

После чего программа «грохнулась». На компьютере Джима даже нет файла `.netrc`. Если вы попробуете запустить ее сами, то, возможно, результат будет совсем иным.

В сгенерированном компилятором C машинном коде функции `main` массив `a` размещен в стеке на три слова раньше адреса возврата, поэтому запись значения `0x7ffff7b36cebUL` в `a[3]` изменяет адрес возврата из `main`, так что он указывает на какой-то код из стандартной библиотеки C, который пытается прочитать пароль из файла `.netrc`. После возврата из `main` выполнение возобновляется не с команды, следующей за вызовом `main`, а с машинного кода, соответствующего таким строкам из библиотеки:

```
warnx(_("Error: .netrc file is readable by others."));
warnx(_("Remove password or make file unreadable by others."));
            goto bad;
```

Но из того, что обращение к элементу массива влияет на поведение последующего предложения `return`, вовсе не следует, что компилятор C не отвечает стандарту. «Неопределенная» операция не просто возвращает неопределенный результат, она дает программе карт-бланш на *произвольное* поведение.

Стандарт C99 предоставляет компилятору такое право, чтобы он мог генерировать более быстрый код. Чем возлагать на компилятор ответственность за обнаружение и обработку странного поведения вроде выхода за конец массива, стандарт предполагает, что программист должен позаботиться о том, чтобы такие ситуации никогда не возникали.

Но опыт показывает, что с этой задачей мы справляемся неважно. Будучи студентом университета штата Юта, исследователь Пень Ли (Peng Li) модифицировал компиляторы C и C++, так чтобы оттранслированные ими программы сообщали о некоторых видах неопределенного поведения. Обнаружилось, что этим грешат почти все программы, в т. ч. и весьма уважаемые, авторы которых стремились

соблюдать высочайшие стандарты. На практике неопределенное поведение часто ведет к брешам в системе безопасности, допускающим написание эксплойта. Червь Морриса распространялся с одной машины на другую, применяя вариант описанной выше техники, и такого рода эксплойты часто встречаются и по сей день.

Теперь определим некоторые термины. Если программа написана так, что ни на каком пути выполнения неопределенное поведение невозможно, то будем говорить, что программа *корректна* (well defined). Если встроенные в язык проверки гарантируют корректность программы, то будем называть язык *типобезопасным* (type safe).

Тщательно написанная программа на C или C++ может оказаться типобезопасной, но ни C, ни C++ не является типобезопасным языком: в приведенной выше программе нет ошибок типизации, и тем не менее она демонстрирует неопределенное поведение. С другой стороны, Python – типобезопасный язык, его интерпретатор тратит время на обнаружение выхода за границы массива и обрабатывает его лучше, чем компилятор C:

```
>>> a = [0]
>>> a[3] = 0x7ffff7b36ceb
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>>
```

Python возбудил исключение, а это уже не неопределенное поведение: в документации по Python сказано, что такое присваивание элементу `a[3]` должно приводить к исключению `IndexError`, что мы и видели. Разумеется, модуль `ctypes`, дающий неограниченный доступ к машине, может стать причиной неопределенного поведения в Python, но сам базовый язык типобезопасен. Таковы же языки Java, JavaScript, Ruby и Haskell.

Отметим, что типобезопасность не зависит от того, когда язык проверяет типы: на этапе компиляции или выполнения. Язык C делает это на этапе компиляции и не является типобезопасным, Python – на этапе выполнения и является таковым.

По иронии судьбы, основные языки системного программирования, C и C++, не типобезопасны, тогда как большинство других популярных языков типобезопасно. Учитывая, что C и C++ предназначены для реализации фундамента системы, что им доверено обеспечивать безопасность границ, на которых происходит контакт с ненадежными данными, типобезопасность была бы весьма важным свойством.

На решение этой проблемы, существующей много десятков лет, – получить типобезопасный язык системного программирования – и нацелен Rust. Он проектировался для реализации тех фундаментальных уровней системы, которым необходимы высокая производительность и точный контроль над ресурсами, но вместе с тем базовые гарантии предсказуемости, которые дает типобезопасность. Далее мы подробно рассмотрим, как Rust справляется с этой задачей.

Выбранная в Rust форма типобезопасности влечет удивительные последствия для многопоточного программирования. Известно, как трудно написать правиль-

ную многопоточную программу на C и C++, поэтому разработчики обращаются к многопоточности только тогда, когда однопоточная программа не способна достичь необходимой производительности. Но Rust гарантирует отсутствие гонок за данные и обнаруживает некорректное использование мьютексов и других примитивов синхронизации на этапе компиляции. В Rust можно пользоваться конкурентностью, не опасаясь сделать программу понятной только самым квалифицированным программистам.

В Rust имеется механизм обхода правил безопасности для тех случаев, когда использование простого указателя абсолютно необходимо. Такой код называется небезопасным, и, хотя в большинстве Rust-программ это не нужно, в главе 21 мы все же обсудим, как его писать и как он укладывается в общую схему безопасности Rust.

Как и в других статически типизированных языках, применение типов в Rust отнюдь не ограничено предотвращением неопределенного поведения. Опытный программист на Rust применяет типы, для того чтобы использование значений было не только безопасным, но и соответствовало целям приложения. В частности, характеристики и универсальные типы, описанные в главе 11, дают лаконичный, гибкий и эффективный способ описать общие свойства группы типов, а затем воспользоваться этой общностью.

В этой книге мы ставим перед собой цель научить вас не просто писать программы на Rust, а применять язык так, чтобы написанные вами программы были безопасны и правильны и обладали предсказуемой производительностью. Наш опыт показывает, что Rust – большой шаг вперед в области системного программирования, и мы хотим помочь вам извлечь из этого все преимущества.

Глава 2

Краткий обзор Rust

Опыт каждого человека строится на базе его языка.
— Анри Делакруа

В этой главе мы рассмотрим несколько коротких программ, чтобы познакомиться с тем, как синтаксис, типы и семантика Rust в совокупности позволяют писать безопасный, конкурентный и эффективный код. Мы опишем процедуру скачивания и установки Rust, продемонстрируем простой код с математическими операциями, поэкспериментируем с веб-сервером на основе сторонней библиотеки и организуем несколько потоков для построения множества Мандельброта.

Скачивание и установка Rust

Для установки Rust проще всего воспользоваться установщиком `rustup`. Зайдите на сайт https://rustup.rs и следуйте приведенным там инструкциям.

Можно вместо этого зайти на сайт https://www.rust-lang.org, нажать кнопку «Downloads» и скачать готовый пакет для Linux, macOS или Windows. Rust также включен в состав некоторых дистрибутивов операционных систем. Мы предпочитаем `rustup`, потому что этот инструмент специально предназначен для управления установкой Rust, как RVM для Ruby или NVM для Node. Например, после выпуска очередной версии Rust для перехода на нее нужно будет всего лишь набрать `rustup update`.

Как бы то ни было, после завершения установки должны появиться три новые программы:

```
$ cargo --version
cargo 0.18.0 (fe7b0cdcf 2017-04-24)
$ rustc --version
rustc 1.17.0 (56124baa9 2017-04-24)
$ rustdoc -version
rustdoc 1.17.0 (56124baa9 2017-04-24)
$
```

Здесь `$` – приглашение к вводу команды; в Windows вместо него выводится `C:\>` или что-то в этом роде. Выше мы выполнили все три установленные команды с флагом печати версии.

- `cargo` – диспетчер компиляции Rust, менеджер пакетов и вообще мастер на все руки. Он позволяет создать новый проект, собрать и запустить программу и вести учет внешних библиотек, от которых зависит ваша программа.

- `rustc` – компилятор Rust. Обычно компилятор вызывает Cargo, но иногда полезно запускать его непосредственно.
- `rustdoc` – средство документирования для Rust. Если в исходный код включены комментарии в определенном формате, то `rustdoc` построит по ним красиво отформатированную HTML-документацию. Как и в случае `rustc`, для запуска `rustdoc` обычно используется Cargo.

Для удобства Cargo может создать новый Rust-пакет со стандартными метаданными:

```
$ cargo new --bin hello
    Created binary (application) `hello` project
```

Эта команда создает пакет с именем `hello`, а флаг `--bin` означает, что пакет будет представлять собой исполняемый файл, а не библиотеку. Вот как выглядит верхний уровень каталога пакета:

```
$ cd hello
$ ls -la
total 24
drwxrwxr-x.  4 jimb jimb 4096 Sep 22 21:09 .
drwx------. 62 jimb jimb 4096 Sep 22 21:09 ..
drwxrwxr-x.  6 jimb jimb 4096 Sep 22 21:09 .git
-rw-rw-r--.  1 jimb jimb    7 Sep 22 21:09 .gitignore
-rw-rw-r--.  1 jimb jimb   88 Sep 22 21:09 Cargo.toml
drwxrwxr-x.  2 jimb jimb 4096 Sep 22 21:09 src

$
```

Как видим, Cargo создал файл `Cargo.toml` для хранения метаданных пакета. Пока что в этом файле почти ничего нет:

```
[package]
name = "hello"
version = "0.1.0"
authors = ["You <you@example.com>"]

[dependencies]
```

Если впоследствии программе понадобятся сторонние библиотеки, то их можно будет прописать в этом файле, и тогда Cargo возьмет на себя скачивание, установку и обновление этих библиотек. Подробно файл `Cargo.toml` будет рассмотрен в главе 8.

Cargo подготовил пакет к работе с системой управления версиями `git`, для чего создал подкаталог `.git` и файл `.gitignore`. Этот шаг можно пропустить, задав в командной строке параметр `--vcs none`.

Каталог `src` содержит исходный код на Rust:

```
$ cd src
$ ls -l
total 4
-rw-rw-r--. 1 jimb jimb 45 Sep 22 21:09 main.rs
```

Выходит, что Cargo уже начал писать программу от нашего имени. Файл `main.rs` содержит такой код:

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

В Rust даже не надо самому писать программу «Hello, World!». А то, что мы видим, – это заготовка новой Rust-программы: два файла, в сумме содержащие девять строк.

Мы можем вызвать команду `cargo run` из любого каталога пакета, чтобы собрать и запустить программу:

```
$ cargo run
  Compiling hello v0.1.0 (file:///home/jimb/rust/hello)
   Finished dev [unoptimized + debuginfo] target(s) in 0.27 secs
    Running `/home/jimb/rust/hello/target/debug/hello`
Hello, world!
$
```

Здесь Cargo вызвал компилятор Rust, `rustc`, а затем запустил созданный исполняемый файл. Исполняемый файл Cargo помещает в подкаталог `target` на верхнем уровне пакета:

```
$ ls -l ../target/debug
total 580
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 build
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 deps
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 examples
-rwxrwxr-x. 1 jimb jimb 576632 Sep 22 21:37 hello
-rw-rw-r--. 1 jimb jimb    198 Sep 22 21:37 hello.d
drwxrwxr-x. 2 jimb jimb     68 Sep 22 21:37 incremental
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 native
$ ../target/debug/hello
Hello, world!
$
```

По завершении работы Cargo может произвести уборку – удалить сгенерированные файлы:

```
$ cargo clean
$ ../target/debug/hello
bash: ../target/debug/hello: No such file or directory
$
```

Простая функция

Синтаксис Rust неоригинален, и это не случайно. Если вы знакомы с языком C, C++, Java или JavaScript, то легко разберетесь в общей структуре Rust-программы. Ниже показана функция, которая вычисляет наибольший общий делитель двух целых чисел, применяя алгоритм Евклида:

```
fn gcd(mut n: u64, mut m: u64) -> u64 {
    assert!(n != 0 && m != 0);
    while m != 0 {
        if m < n {
```

```
            let t = m;
            m = n;
            n = t;
        }
        m = m % n;
    }
    n
}
```

Ключевым словом `fn` начинается определение функции – в данном случае функции `gcd`, принимающей два параметра `n` и `m` типа `u64`, обозначающего 64-разрядное целое число без знака. После лексемы `->` указывается тип возвращаемого значения, в данном случае `u64`. По соглашению в Rust принят отступ 4 пробела.

Имена целых типов в Rust содержат указание на разрядность и наличие знака: `i32` – 32-разрядное целое со знаком, `u8` – 8-разрядное целое без знака (применяется для «байтовых» значений) и т. д. Типы `isize` и `usize` соответствуют целым размерам указателя со знаком и без знака: 32 разряда на 32-разрядной платформе и 64 – на 64-разрядной. В Rust также есть два типа с плавающей точкой, `f32` и `f64`, соответствующие числам одинарной и двойной точности в формате IEEE, как `float` и `double` в C и C++.

По умолчанию, после того как переменная инициализирована, ее значение нельзя изменять, но, поместив ключевое слово `mut` (сокращение от «mutable», произносится «мьют») перед параметрами `n` и `m`, мы разрешаем изменять их в теле функции. На практике большинству переменных значения не присваиваются, поэтому наличие `mut` в объявлении переменных, которым значения все-таки присваиваются, дает полезную информацию при чтении программы.

Тело функции начинается обращением к макросу `assert!`, который проверяет, что оба аргумента отличны от нуля. Знак `!` говорит, что это вызов макроса, а не функции. Как и макрос `assert` в C и C++, макрос `assert!` в Rust проверяет, что аргумент равен `true`, а если это не так, то завершает программу, печатая полезное сообщение, включающее информацию о месте, в котором произошла ошибка; такое внезапное завершение называется «паникой». В отличие от C и C++, где утверждения можно пропустить, Rust всегда выполняет проверки вне зависимости от того, как откомпилирована программа. Существует также макрос `debug_assert!`, сформулированные в нем утверждения пропускаются, если программа была откомпилирована в режиме максимального быстродействия.

Главное в нашей функции – цикл `while`, содержащий предложение `if` и присваивание. В отличие от C и C++, в Rust скобки вокруг условных выражений необязательны, однако управляемые ими предложения должны быть заключены в фигурные скобки.

Предложение `let` служит для объявления локальной переменной, в нашей функции это переменная `t`. Указывать тип `t` необязательно при условии, что Rust может вывести его из того, как переменная используется. В нашем случае для `t` подходит только тип `u64`, соответствующий типам `m` и `n`. Rust выводит типы только внутри тел функций, типы параметров и возвращаемого значения функции необходимо указывать явно. Если бы мы все же захотели явно указать тип `t`, то могли бы написать:

```
let t: u64 = m; ...
```

В Rust имеется предложение `return`, но в функции `gcd` в нем нет необходимости. Если тело функции заканчивается выражением, за которым *не* следует точка с запятой, то значение выражения и будет значением функции. На самом деле любой блок, заключенный в фигурные скобки, может выступать в роли выражения. Например, следующее выражение печатает сообщение, а его значением является `x.cos()`:

```
{
    println!("evaluating cos x");
    x.cos()
}
```

В Rust такая форма возврата значения встречается очень часто, когда поток управления доходит до конца функции, а предложения `return` используются, только когда нужно явно вернуть управление из середины функции.

Написание и выполнение автономных тестов

В сам язык Rust встроены простые средства тестирования. Чтобы протестировать функцию `gcd`, можно написать:

```
#[test]
fn test_gcd() {
    assert_eq!(gcd(14, 15), 1);

    assert_eq!(gcd(2 * 3 * 5 * 11 * 17,
                   3 * 7 * 11 * 13 * 19),
               3 * 11);
}
```

Здесь определяется функция `test_gcd`, которая вызывает `gcd` и проверяет, что та возвращает правильное значение. Маркер `#[test]` перед определением помечает `test_gcd` как тестовую функцию; это значит, что при обычной компиляции она пропускается, но компилируется и автоматически вызывается, если программа запускается командой `cargo test`. Предположим, что определения функций `gcd` и `test_gcd` вписаны в пакет `hello`, созданный в начале этой главы. Если текущим является любой каталог внутри дерева пакета, то тесты можно запустить следующим образом:

```
$ cargo test
    Compiling hello v0.1.0 (file:///home/jimb/rust/hello)
     Finished dev [unoptimized + debuginfo] target(s) in 0.35 secs
      Running /home/jimb/rust/hello/target/debug/deps/hello-2375a82d9e9673d7

running 1 test
test test_gcd ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

$
```

Тестовые функции могут быть разбросаны по всему дереву исходного кода и находиться рядом с тестируемым кодом. Команда `cargo test` автоматически найдет их и запустит.

Маркер `#[test]` – пример *атрибута*. Атрибуты – это расширяемая система, позволяющая снабжать функции и другие объявления дополнительной информацией, подобно атрибутам в C++ и C# или аннотациям в Java. Они используются для управления предупреждениями компилятора, условного включения кода (подобно `#ifdef` в C и C++), инструктируют Rust о том, как взаимодействовать с кодом на других языках, и т. д. Мы встретим немало примеров атрибутов по мере изложения.

Обработка аргументов командной строки

Если мы хотим, чтобы наша программа принимала последовательность чисел в виде аргументов командной строки и печатала их наибольший общий делитель, то можем заменить функцию `main` такой:

```
use std::io::Write;
use std::str::FromStr;

fn main() {
    let mut numbers = Vec::new();

    for arg in std::env::args().skip(1) {
        numbers.push(u64::from_str(&arg)
                     .expect("error parsing argument"));
    }

    if numbers.len() == 0 {
        writeln!(std::io::stderr(), "Usage: gcd NUMBER ...").unwrap();
        std::process::exit(1);
    }

    let mut d = numbers[0];
    for m in &numbers[1..] {
        d = gcd(d, *m);
    }

    println!("The greatest common divisor of {:?} is {}",
             numbers, d);
}
```

Код довольно длинный, поэтому разберем его по частям.

```
use std::io::Write;
use std::str::FromStr;
```

Объявление `use` вводит в область видимости две характеристики, `Write` и `FromStr`. Характеристики мы будем рассматривать в главе 11, а пока достаточно считать, что характеристика – это набор методов, который могут реализовывать типы. И хотя имена `Write` и `FromStr` нигде в программе не используются, характеристика должна присутствовать в области видимости, чтобы мы могли обращаться к ее методам. В данном случае:

- любой тип, реализующий характеристику `Write`, обладает методом `write_fmt`, который записывает отформатированный текст в поток. Тип `std::io::Stderr` реализует `Write`, и для печати сообщений об ошибках мы употребляем макрос `writeln!`, который расширяется в код, содержащий вызов метода `write_fmt`;

- любой тип, реализующий характеристику `FromStr`, обладает методом `from_str`, который пытается получить значение этого типа путем разбора строки. Тип `u64` реализует `FromStr`, так что мы можем вызывать метод `u64::from_str` для разбора аргументов командной строки.

Переходим к функции `main`:

```
fn main() {
```

Наша функция `main` не возвращает значение, поэтому можно опустить стрелку `->` и тип, которые обычно находятся после списка параметров.

```
let mut numbers = Vec::new();
```

Мы объявляем изменяемую локальную переменную `numbers` и инициализируем ее пустым вектором. Тип `Vec` в Rust описывает растущий вектор, он аналогичен типу `std::vector` в C++, списку в Python и массиву в JavaScript. И хотя векторы могут динамически расти и уменьшаться, переменную все равно нужно пометить ключевым словом `mut`, чтобы Rust позволил добавлять числа в конец вектора.

Переменная `numbers` имеет тип `Vec<u64>` – вектор элементов типа `u64`, но, как и раньше, нет необходимости указывать тип явно. Rust выведет его, отчасти потому что мы добавляем в вектор значения типа `u64`, но также и потому, что передаем элементы вектора функции `gcd`, которая принимает только значения типа `u64`.

```
for arg in std::env::args().skip(1) {
```

Здесь для обработки аргументов командной строки используется цикл `for`, в котором переменной `arg` поочередно присваивается значение каждого аргумента.

Функция `std::env::args` возвращает *итератор* – объект, который по требованию возвращает значения аргументов и сообщает, когда их больше не останется. Итераторы встречаются в Rust повсеместно; стандартная библиотека содержит также итераторы, порождающие элементы вектора, строки файла, сообщения, получаемые из канала связи, да и вообще почти все, что имеет смысл обходить в цикле. Итераторы в Rust очень эффективны, обычно компилятор транслирует их в такой же код, как мы написали бы вручную. В главе 15 мы объясним, как они работают, и приведем примеры.

Помимо использования в циклах `for`, итераторы предоставляют ряд методов, которые можно вызывать непосредственно. Например, первое значение, возвращаемое итератором `std::env::args`, всегда совпадает с именем работающей программы. Поскольку мы хотим пропустить его, то вызываем метод итератора `skip`, который возвращает новый итератор, пропускающий первое значение.

```
numbers.push(u64::from_str(&arg)
             .expect("error parsing argument"));
```

Здесь вызывается метод `u64::from_str`, который пытается разобрать аргумент `arg` как 64-разрядное целое без знака. `u64::from_str` – это не метод значения типа `u64`, а функция, ассоциированная с типом `u64`, – как статический метод в C++ или Java. Функция `from_str` возвращает не само значение типа `u64`, а объект типа `Result`, показывающий, как завершился разбор: успешно или с ошибкой. Возможны два варианта `Result`:

- один обозначается `Ok(v)` и означает, что разбор завершился успешно и получено значение `v`;

- другой обозначается `Err(e)` и означает, что разбор завершился с ошибкой, а `e` – описание причины ошибки.

Все функции, занимающиеся вводом-выводом или еще как-то взаимодействующие с операционной системой, возвращают значения типа `Result`, причем вариант `Ok` содержит результаты в случае успешного выполнения – число переданных байтов, описатель открытого файла и т. д., а вариант `Err` – системный код ошибки. В отличие от большинства современных языков, в Rust нет исключений, а ошибки обрабатываются либо с помощью `Result`, либо посредством паники, как описано в главе 7.

Для проверки успешности разбора мы вызываем метод `expect` объекта `Result`. Если получен результат `Err(e)`, то `expect` печатает сообщение, включающее описание `e`, и немедленно завершает программу. Если же получен результат `Ok(v)`, то `expect` возвращает значение `v`, которое мы добавляем в конец вектора чисел.

```
if numbers.len() == 0 {
    writeln!(std::io::stderr(), "Usage: gcd NUMBER ...").unwrap();
    std::process::exit(1);
}
```

Если множество чисел пусто, то говорить о наибольшем общем делителе не приходится, поэтому мы проверяем, что вектор содержит хотя бы один элемент, и выходим, если это не так. Макрос `writeln!` выводит сообщение в стандартный поток ошибок, который мы получаем от функции `std::io::stderr()`. Вызов `.unwrap()` – это лаконичный способ проверить, что попытка печати сообщения об ошибке сама не завершается ошибкой; вызов `expect` тоже сгодился бы, но здесь он, пожалуй, излишен.

```
let mut d = numbers[0];
for m in &numbers[1..] {
    d = gcd(d, *m);
}
```

В этом цикле переменная `d` изменяется и всегда содержит наибольший общий делитель обработанных до сих пор чисел. Как и раньше, `d` объявлена изменяемой, чтобы ей можно было присваивать значения внутри цикла.

В этом цикле `for` есть два непонятных момента. Во-первых, что означает оператор `&` в конструкции `for m in &numbers[1..]`? Во-вторых, что означает `*` в выражении `gcd(d, *m)`? Эти моменты взаимосвязаны.

До сих пор в программе встречались только простые значения, например целые числа, которые помещаются в блок памяти фиксированного размера. Но сейчас мы собираемся обойти вектор, размер которого может быть произвольным и, вообще говоря, очень большим. Rust проявляет осторожность при обработке таких значений: он хочет оставить за программистом контроль над потреблением памяти, позволяя четко указать, как долго будет существовать каждое значение, и вместе с тем гарантировать, что память освобождается сразу, как только становится ненужной.

Поэтому в процессе обхода мы говорим Rust, что *владение* элементами вектора должно оставаться у `numbers`, а мы только *заимствуем* элементы для нужд цикла. Оператор `&` в выражении `&numbers[1..]` заимствует *ссылку* на элементы вектора, начиная со второго. В цикле `for` перебираются элементы, на которые указывает ссылка, так что `m` последовательно заимствует каждый элемент. Оператор `*` в выражении `*m` *разыменовывает* `m`, т. е. дает значение, на которое `m` ссылается, – в данном случае это следующее число типа `u64`, которое мы хотим передать `gcd`.

Наконец, поскольку переменная `numbers` владеет вектором, Rust автоматически освобождает его, когда `numbers` выходит из области видимости в конце `main`.

Правила, касающиеся владения и ссылок, – ключ к управлению памятью и безопасной конкурентности в Rust, мы подробно обсудим их в главах 4 и 5. Чтобы свободно программировать на Rust, вы должны твердо усвоить эти правила, но пока достаточно запомнить, что `&x` заимствует ссылку на `x`, а `*r` – значение, на которое ссылается ссылка `r`.

Продолжаем анализировать программу:

```
println!("The greatest common divisor of {:?} is {}",
         numbers, d);
```

Завершив обход элементов вектора `numbers`, программа печатает результат в стандартный поток вывода. Макрос `println!` принимает шаблонную строку, подставляет отформатированные аргументы вместо встречающихся в ней маркеров вида `{...}` и записывает результат в стандартный поток вывода.

В отличие от C и C++, которые требуют, чтобы в случае успешного завершения программы функция `main` возвращала 0, а в случае ошибки – ненулевое значение, Rust предполагает, что если `main` вообще вернула управление, то программа завершилась успешно. И лишь в случае явного вызова функций типа `expect` или `std::process::exit` программа может завершиться с кодом ошибки.

Команда `cargo run` позволяет передать программе аргументы, так что мы можем протестировать обработку командной строки:

```
$ cargo run 42 56
   Compiling hello v0.1.0 (file:///home/jimb/rust/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.38 secs
     Running `/home/jimb/rust/hello/target/debug/hello 42 56`
The greatest common divisor of [42, 56] is 14
$ cargo run 799459 28823 27347
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `/home/jimb/rust/hello/target/debug/hello 799459 28823 27347`
The greatest common divisor of [799459, 28823, 27347] is 41
$ cargo run 83
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `/home/jimb/rust/hello/target/debug/hello 83`
The greatest common divisor of [83] is 83
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `/home/jimb/rust/hello/target/debug/hello`
Usage: gcd NUMBER ...
$
```

В этом разделе мы рассмотрели несколько функций из стандартной библиотеки Rust. Если вам интересно, что еще в ней есть, рекомендуем обратиться к онлайновой документации. Реализован поиск по документации, который не только облегчает изучение, но и содержит ссылки на исходный код. Команда `rustup` автоматически устанавливает копию документации вместе с Rust. Для просмотра документации по стандартной библиотеке в браузере выполните команду

```
$ rustup doc -std
```

Документацию можно также посмотреть на сайте https://doc.rust-lang.org/.

Простой веб-сервер

Одна из сильных сторон Rust – набор библиотечных пакетов, написанных сообществом пользователей Rust и доступных любому желающему. Многие из них опубликованы на сайте https://crates.io. Команда `cargo` упрощает использование пакетов с этого сайта в ваших программах: она автоматически скачивает нужную версию, собирает ее и обновляет по мере необходимости. Пакет Rust, будь то библиотека или исполняемый файл, называется крейтом (*crate*); названия `cargo` и `crates.io` происходят от этого слова.

Чтобы показать, как все это работает, соберем простой веб-сервер, воспользовавшись веб-каркасом `iron`, HTTP-сервером `hyper` и многочисленными крейтами, от которых они зависят. Наш сайт будет запрашивать у пользователя два числа и вычислять их наибольший общий делитель.

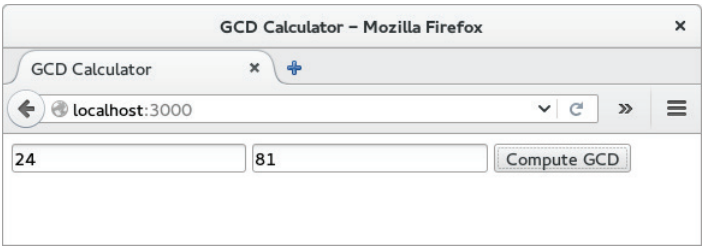


Рис. 2.1 ❖ Веб-страница, предлагающая вычислить НОД

Сначала попросим `cargo` создать новый пакет с именем `iron-gcd`:

```
$ cargo new --bin iron-gcd
     Created binary (application) `iron-gcd` project
$ cd iron-gcd
$
```

Затем отредактируем файл `Cargo.toml` в новом проекте, перечислив нем используемые пакеты:

```
[package]
name = "iron-gcd"
version = "0.1.0"
authors = ["You <you@example.com>"]

[dependencies]
iron = "0.5.1"
mime = "0.2.3"
router = "0.5.1"
urlencoded = "0.5.0"
```

В каждой строке раздела `[dependencies]` указываются имя одного крейта из имеющихся на сайте `crates.io` и нужная нам версия. Вполне возможно, что на сайте `crates.io` есть более поздние версии, чем указано выше, но, задавая те версии, с которыми тестировалась программа, мы можем быть уверены, что она будет компилироваться даже после выпуска новых версий. Подробно управление версиями обсуждается в главе 8.

Отметим, что перечислять нужно только те пакеты, которыми мы пользуемся непосредственно; обо всех остальных `cargo` позаботится самостоятельно.

Для первого раза сделаем веб-сервер совсем простым: он будет выдавать только одну страницу, которая запрашивает у пользователя два числа. Поместите в файл `iron-gcd/src/main.rs` такой текст:

```
extern crate iron;
#[macro_use] extern crate mime;

use iron::prelude::*;
use iron::status;

fn main() {
    println!("Serving on http://localhost:3000...");
    Iron::new(get_form).http("localhost:3000").unwrap();
}

fn get_form(_request: &mut Request) -> IronResult<Response> {
    let mut response = Response::new();

    response.set_mut(status::Ok);
    response.set_mut(mime!(Text/Html; Charset=Utf8));
    response.set_mut(r#"
        <title>GCD Calculator</title>
        <form action="/gcd" method="post">
          <input type="text" name="n"/>
          <input type="text" name="n"/>
          <button type="submit">Compute GCD</button>
        </form>
    "#);

    Ok(response)
}
```

В начале идут две директивы `extern crate`, которые предоставляют в распоряжение нашей программы крейты `iron` и `mime`, прописанные в файле `Cargo.toml`. Атрибут `#[macro_use]` перед директивой `extern crate mime` предупреждает Rust, что мы планируем использовать макросы, экспортируемые этим крейтом.

Далее следуют объявления `use`, делающие доступными некоторые открытые средства крейтов. Объявление `use iron::prelude::*` делает непосредственно видимыми программе все открытые имена в модуле `iron::prelude`. Вообще говоря, лучше бы полностью выписывать используемые имена, как мы поступили в случае `iron::status`, но по соглашению, если модуль называется `prelude`, значит, все экспортируемое им – средства общего характера, которые, скорее всего, понадобятся любому пользователю крейта. Так что в данном случае всеобъемлющая директива `use` более уместна.

Наша функция `main` проста: она печатает сообщение о том, как подключиться к серверу, вызывает функцию `Iron::new` для создания сервера, а затем назначает ему прослушиваемый TCP-порт 3000 на локальной машине. Мы передаем `Iron::new` функцию `get_form`, это означает, что сервер должен использовать эту функцию для обработки всех запросов. Ниже мы улучшим это решение.

Функция `get_form` принимает изменяемую ссылку (записывается в виде `&mut`) на значение типа `Request`, представляющее HTTP-запрос, для обработки которого

она вызвана. Данная конкретная функция не использует параметра `_request`, но ниже мы встретим функцию, где он используется. А пока скажем, что если имя параметра начинается знаком _, то Rust понимает, что этот параметр может не использоваться, и не выдает предупреждение.

В теле функции строится значение типа `Response`. Метод `set_mut` принимает решение о том, какую часть ответа формировать, исходя из типа аргумента, поэтому каждое обращение к `set_mut` определяет разные части `response`: передав `status::Ok`, мы задаем состояние HTTP; передав тип содержимого (с помощью макроса `mime!`, импортированного из крейта `mime`), мы формируем заголовок `Content-Type`, а передача строки задает тело ответа.

Поскольку в тексте ответа много двойных кавычек, мы воспользовались синтаксисом «простой строки»: буква `r`, нуль или более знаков `#`, двойная кавычка, затем содержимое строки, завершаемое еще одной двойной кавычкой, за которой следует такое же число символов `#`, как вначале. Внутри простой строки могут встречаться любые символы без экранирования, в т. ч. двойные кавычки; никакие управляющие последовательности, например `"`, не распознаются. Мы всегда можем гарантировать, что строка заканчивается именно там, где мы хотим, употребив вместе с кавычками больше знаков `#`, чем встречается в тексте.

Возвращаемый функцией тип `IronResult<Response>` – это еще один вариант типа `Result`, с которым мы уже сталкивались: `Ok(r)`, если значение `r` типа `Response` сформировано успешно, или `Err(e)` в случае ошибки `e`. Возвращаемое значение `Ok(response)` конструируется в конце функции с помощью синтаксиса «последнего выражения».

Написав функцию `main.rs`, мы можем воспользоваться командой `cargo run`, чтобы сделать все необходимое для запуска программы: скачать необходимые крейты, откомпилировать их, собрать нашу программу, скомпоновать все вместе и выполнить:

```
$ cargo run
    Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading iron v0.5.1
 Downloading urlencoded v0.5.0
 Downloading router v0.5.1
 Downloading hyper v0.10.8
 Downloading lazy_static v0.2.8
 Downloading bodyparser v0.5.0
...
Compiling conduit-mime-types v0.7.3
   Compiling iron v0.5.1
   Compiling router v0.5.1
   Compiling persistent v0.3.0
   Compiling bodyparser v0.5.0
   Compiling urlencoded v0.5.0
   Compiling iron-gcd v0.1.0 (file:///home/jimb/rust/iron-gcd)
     Running `target/debug/iron-gcd`
Serving on http://localhost:3000...
```

Теперь мы можем зайти на URL-адрес сервера в браузере и увидеть показанную выше страницу.

Увы, нажатие кнопки «Compute GCD» всего лишь отправляет браузер на адрес http://localhost:3000/gcd, и сервер отображает ту же самую страницу; так проис-

ходит при переходе на любой адрес, обслуживаемый нашим сервером. Исправим это, воспользовавшись типом `Router`, чтобы ассоциировать различные обработчики с разными путями.

Сначала подготовим все, чтобы можно было использовать имя `Router` без квалификации, для чего добавим следующие объявления в файл `iron-gcd/src/main.rs`:

```
extern crate router;
use router::Router;
```

Обычно при программировании на Rust все объявления `extern crate` и `use` собирают в начале файла, но это необязательно: Rust допускает объявления в любом порядке, лишь бы уровень вложенности был правильным. (Определения макросов и директивы `extern crate` с атрибутом `#[macro_use]` являются исключением из этого правила: они должны предшествовать использованию.)

Теперь модифицируем нашу функцию `main`:

```
fn main() {
    let mut router = Router::new();

    router.get("/", get_form, "root");
    router.post("/gcd", post_gcd, "gcd");

    println!("Serving on http://localhost:3000...");
    Iron::new(router).http("localhost:3000").unwrap();
}
```

Мы создаем объект `Router`, задаем функции-обработчики для двух путей и передаем этот объект в качестве обработчика запроса функции `Iron::new`. Тем самым мы получаем веб-сервер, который анализирует путь в URL-адресе, чтобы понять, какой обработчик вызвать.

Теперь все готово для написания функции `post_gcd`:

```
extern crate urlencoded;

use std::str::FromStr;
use urlencoded::UrlEncodedBody;

fn post_gcd(request: &mut Request) -> IronResult<Response> {
    let mut response = Response::new();

    let form_data = match request.get_ref::<UrlEncodedBody>() {
        Err(e) => {
            response.set_mut(status::BadRequest);
            response.set_mut(format!("Error parsing form data: {:?}\n", e));
            return Ok(response);
        }
        Ok(map) => map
    };

    let unparsed_numbers = match form_data.get("n") {
        None => {
            response.set_mut(status::BadRequest);
            response.set_mut(format!("form data has no 'n' parameter\n"));
            return Ok(response);
        }
        Some(nums) => nums
```

```
    };

    let mut numbers = Vec::new();
    for unparsed in unparsed_numbers {
        match u64::from_str(&unparsed) {
            Err(_) => {
                response.set_mut(status::BadRequest);
                response.set_mut(
                    format!("Value for 'n' parameter not a number: {:?}\n",
                            unparsed));
                return Ok(response);
            }
            Ok(n) => { numbers.push(n); }
        }
    }

    let mut d = numbers[0];
    for m in &numbers[1..] {
        d = gcd(d, *m);
    }

    response.set_mut(status::Ok);
    response.set_mut(mime!(Text/Html; Charset=Utf8));
    response.set_mut(
        format!("The greatest common divisor of the numbers {:?} is <b>{}</b>\n",
                numbers, d));
    Ok(response)
}
```

В основном эта функция состоит из последовательности выражений `match`, незнакомых программистам на C, C++, Java или JavaScript, но отлично знакомых работающим на Haskell или OCaml. Мы уже упоминали, что `Result` принимает либо значение `Ok(s)`, где `s` – значение в случае успеха, либо `Err(e)`, где `e` – значение в случае ошибки. Имея значение `Result res`, мы можем проверить, что в нем находится, и получить это значение с помощью выражения `match` вида:

```
match res {
    Ok(success) => { ... },
    Err(error)  => { ... }
}
```

Это условное выражение, как `if` или `switch` в языке C: если `res` равно `Ok(v)`, то выполняется первая ветвь, причем `success` получает значение `v`. Если же `res` равно `Err(e)`, то выполняется вторая ветвь, в которой `error` получает значение `e`. Переменные `success` и `error` локальны каждая в своей ветви. Значением всего выражения `match` является значение той ветви, которая выполнялась.

Изящество выражения `match` в том, что программа может получить значение, хранящееся в `Result`, только проверив предварительно, какого оно характера; мы никогда не сможем ошибочно принять ошибку за успешное завершение. В C и C++ часто забывают проверить, вернула ли функция код ошибки или нулевой указатель, но в Rust такие ошибки «отлавливаются» на этапе компиляции. Эта простая мера является важным улучшением в плане удобства работы с языком.

Rust позволяет определять собственные типы наподобие `Result` с вариантами, содержащими значения, и использовать выражения `match` для их анализа. В Rust такие типы называются перечислениями (enum), а в других языках они известны

под названием «алгебраические типы данных». Подробно перечисления описываются в главе 10.

Теперь, когда мы знаем, как читать выражения `match`, структура функции `post_gcd` становится понятной.

- Она вызывает функцию `request.get_ref::<UrlEncodedBody>()`, чтобы разобрать тело запроса и представить его в виде таблицы, отображающей имена параметров на массивы значений; в случае ошибки разбора функция сообщает об этом клиенту. Часть `::<UrlEncodedBody>` – это параметрический тип, показывающий, какую часть объекта `Request` должен извлекать метод `get_ref`. В данном случае тип `UrlEncodedBody` относится к телу, которое разбирается как URL-кодированная строка запроса. Мы еще вернемся к параметрическим типам в разделе «Конкурентность» этой главы.
- Функция ищет в этой таблице параметр с именем `"n"`, в котором хранятся числа, введенные в HTML-форме на веб-странице. Это не одна строка, а вектор строк, поскольку в форме может быть несколько параметров с одинаковым именем.
- Функция обходит вектор строк, пытаясь разобрать каждую как 64-разрядное число. В случае ошибки разбора возвращается страница с сообщением об ошибке.
- Наконец, функция вычисляет наибольший общий делитель, как и раньше, и конструирует ответ, содержащий результат. Макросу `format!` передается такая же шаблонная строка, как макросам `writeln!` и `println!`, но он возвращает строковое значение, а не выводит текст в поток.

Осталась только ранее написанная функция `gcd`. Теперь мы можем прервать серверы, которые еще работают, а затем пересобрать и перезапустить программу:

```
$ cargo run
   Compiling iron-gcd v0.1.0 (file:///home/jimb/rust/iron-gcd)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/iron-gcd`
Serving on http://localhost:3000...
```

Если теперь перейти по адресу `http://localhost:3000`, ввести два числа и нажать кнопку «Compute GCD», то будет показан результат:

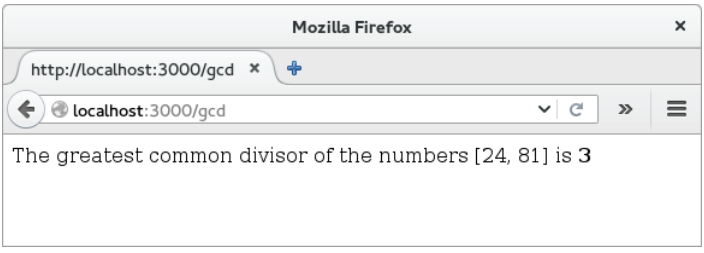


Рис. 2.2 ❖ Страница с результатом вычисления НОД

Конкурентность

Одна из сильных сторон Rust – поддержка конкурентного программирования. Те же правила, благодаря которым в Rust-программах отсутствуют ошибки при рабо-

те с памятью, гарантируют, что при доступе потоков к общей памяти невозможны гонки за данные. Например:

- если для координации потоков, вносящих изменения в разделяемую структуру данных, используются мьютексы, то Rust гарантирует, что доступ к данным возможен, только когда поток захватил мьютекс, и автоматически освобождает мьютекс, когда блокировка больше не нужна. В C и C++ связь между мьютексом и данными, которые он защищает, описывается только в комментариях;
- если вы хотите обращаться из нескольких потоков к общим данным только для чтения, то Rust гарантирует, что они не будут случайно модифицированы. В C и C++ в этом может помочь система типов, но при этом легко допустить ошибку;
- при передаче владения структурой данных от одного потока другому Rust гарантирует, что всякий доступ действительно прекращен. В C и C++ вы сами должны следить за тем, чтобы прежний владелец не обращался к данным. Если программа написана некорректно, то последствия могут зависеть от того, что находится в кэше процессора и сколько операций записи в память было произведено недавно. Что ж, некого винить, кроме самого себя.

В этом разделе мы разберем процедуру написания нашей второй многопоточной программы. Первую вы уже написали, сами того не сознавая: веб-каркас Iron, с помощью которого был реализован сервер нахождения НОД, использует пул потоков, в которых работают функции обработки запросов. Если сервер получает запросы одновременно, то функции `get_form` и `post_gcd` могут параллельно выполняться в нескольких потоках. Это удивительно, поскольку при разработке этих функций мы вообще не думали о конкурентности. Но Rust гарантирует безопасность, каким бы сложным ни был сервер: если программа компилируется, значит, в ней нет гонок за данные. Все функции в Rust потокобезопасны.

Программа из этого раздела рисует множество Мандельброта – фрактал, порожденный итеративным применением простой функции к комплексным числам. Изображение множества Мандельброта на графике часто называют «естественно параллельным» алгоритмом, поскольку характер взаимодействия между потоками очень прост. В главе 10 мы рассмотрим более сложные взаимодействия, но данная задача позволит продемонстрировать основные идеи.

Прежде всего создадим новый проект Rust:

```
$ cargo new --bin mandelbrot
    Created binary (application) `mandelbrot` project
```

Код будет находиться в файле `mandelbrot/src/main.rs`, а описание зависимостей – в файле `mandelbrot/Cargo.toml`.

Прежде чем приступать к конкурентной реализации множества Мандельброта, объясним, какое вычисление нам предстоит.

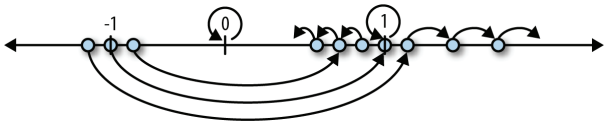
Что такое множество Мандельброта

Когда читаешь код, полезно понимать, что он пытается сделать, поэтому совершим небольшой экскурс в область чистой математики. Начнем с простого случая и будем постепенно усложнять его, пока не доберемся до существа множества Мандельброта.

Ниже представлен бесконечный цикл, написанный с использованием предложения `loop`, специально предназначенного в Rust для этой цели:

```
fn square_loop(mut x: f64) {
    loop {
        x = x * x;
    }
}
```

В действительности Rust может заметить, что переменная `x` нигде не используется, так что и ее значение можно не вычислять. Но пока предположим, что код работает именно так, как написан. Что происходит со значением `x`? При возведении в квадрат число, меньшее 1, уменьшается и при повторении операции стремится к нулю. А число, большее 1, увеличивается и, следовательно, стремится к бесконечности. Отрицательное число после возведения в квадрат становится положительным, после чего ведет себя, как в одном из предыдущих случаев.



Таким образом, в зависимости от значения, переданного функции `square_loop`, `x` либо стремится к 0, либо остается равным 1, либо стремится к бесконечности.

Теперь немного изменим цикл:

```
fn square_add_loop(c: f64) {
    let mut x = 0.;
    loop {
        x = x * x + c;
    }
}
```

На этот раз начальное значение `x` равно нулю, а на каждой итерации оно возводится в квадрат и к результату прибавляется `c`. Понять, как теперь изменяется `x`, сложнее, но после нескольких экспериментов мы приходим к выводу, что если `c` больше `0.25` или меньше `-2.0`, то `x` стремится к бесконечности, а иначе остается в окрестности нуля.

Следующий шаг: рассмотрим тот же цикл, только вместо чисел типа `f64` будем использовать комплексные числа. Крейт `num` на сайте `crates.io` содержит нужный нам тип комплексного числа, поэтому добавим его в раздел `[dependencies]` в нашем файле `Cargo.toml`. Ниже показано, как сейчас должен выглядеть этот файл (в дальнейшем мы его еще расширим):

```
[package]
name = "mandelbrot"
version = "0.1.0"
authors = ["You <you@example.com>"]

[dependencies]
num = "0.1.27"
```

Теперь можно написать предпоследний вариант цикла:

```
extern crate num;
use num::Complex;

#[allow(dead_code)]
fn complex_square_add_loop(c: Complex<f64>) {
    let mut z = Complex { re: 0.0, im: 0.0 };
    loop {
        z = z * z + c;
    }
}
```

По традиции комплексное число обозначается `z`, поэтому мы переименовали переменную цикла. Выражение `Complex { re: 0.0, im: 0.0 }` – это способ записи комплексного нуля в нотации крейта `num`. `Complex` – это тип структуры (или «struct») в Rust, определенный следующим образом:

```
struct Complex<T> {
    /// Вещественная часть комплексного числа
    re: T,

    /// Мнимая часть комплексного числа
    im: T
}
```

Здесь определена структура `Complex` с двумя полями: `re` и `im`. `Complex` – универсальная (generic) структура: часть `<T>` после имени типа можно прочитать как «для любого типа `T`». Таким образом, `Complex<f64>` – это тип комплексного числа, для которого поля `re` и `im` имеют тип `f64`, `Complex<f32>` – комплексное число, в котором вещественная и мнимая часть – 32-разрядные числа с плавающей точкой, и т. д. При таком определении выражение вида `Complex { re: R, im: I }` порождает комплексное число, в котором поле `re` инициализировано значением `R`, а поле `im` – значением `I`.

В крейте `num` определены арифметические операторы – `*`, `+` и другие – для значений типа `Complex`, поэтому оставшаяся часть функции работает так же, как и раньше, только для точек на комплексной плоскости, а не на вещественной прямой. В главе 11 мы объясним, как заставить операторы Rust работать с пользовательскими типами.

Вот мы и дошли до конечной точки путешествия в мир чистой математики. Множество Мандельброта определяется как множество комплексных чисел `c`, для которых `z` не устремляется в бесконечность. Первоначальный цикл возведения в квадрат вел себя вполне предсказуемо: любое число, большее 1 или меньшее –1, будет стремиться к бесконечности. Включение `+ c` на каждой итерации делает поведение менее очевидным: как было сказано, `z` устремляется в бесконечность, если `c` больше `0.25` или меньше `-2`. Распространение же этой игры на комплексные числа дает поистине странную и удивительно красивую картину, которую мы хотим нарисовать.

Будем рассматривать вещественную и мнимую часть комплексного числа `c`, `c.re` и `c.im` как координаты `x` и `y` точки на декартовой плоскости и будем рисовать точку черным цветом, если `c` принадлежит множеству Мандельброта, и более светлым в противном случае. Таким образом, для каждого пикселя изображения

мы должны выполнить показанный выше цикл на комплексной плоскости и посмотреть, устремляется точка к бесконечности или вечно обращается вокруг начала координат, и в зависимости от результата нарисовать пиксель тем или иным цветом.

Бесконечный цикл, конечно, работает не быстро, но для нетерпеливых есть две уловки. Во-первых, если ограничить количество итераций, то можно будет получить приемлемую аппроксимацию множества. От того, сколько итераций оставить, зависит точность построения его границы. Во-вторых, можно доказать, что если `z` хотя бы один раз покидает круг радиуса 2 с центром в начале координат, то в конечном итоге она отдалится от начала на сколь угодно большое расстояние.

Ниже приведен окончательный вариант цикла – сердце нашей программы.

```
extern crate num;
use num::Complex;

/// Пытается определить, принадлежит ли `c` множеству Мандельброта, ограничившись
/// `limit` итерациями.
///
/// Если `c` не принадлежит множеству, вернуть `Some(i)`, где `i` - число итераций,
/// понадобившееся для того, чтобы `c` покинула круг радиуса 2 с центром в начале
/// координат. Если `c` может принадлежать множеству (точнее, если после limit итераций
/// не удалось доказать, что `c` не является элементом множества), то вернуть `None`.
fn escape_time(c: Complex<f64>, limit: u32) -> Option<u32> {
    let mut z = Complex { re: 0.0, im: 0.0 };
    for i in 0..limit {
        z = z*z + c;
        if z.norm_sqr() > 4.0 {
            return Some(i);
        }
    }

    None
}
```

Эта функция принимает комплексное число `c`, которое мы хотим проверить на принадлежность множеству Мандельброта, и предельное число итераций, после которого мы готовы признать, что `c` может быть элементом этого множества.

Функция возвращает значение типа `Option<u32>`. В стандартной библиотеке Rust тип `Option` определен следующим образом:

```
enum Option<T> {
    None,
    Some(T),
}
```

`Option` – *перечислимый тип*, или просто «перечисление» (enum), поскольку в его определении перечислено несколько вариантов, которые могут принимать значения этого типа: для любого типа `T` значением типа `Option<T>` может быть либо `Some(v)`, где `v` – значение типа `T`, либо `None` – т. е. нет никакого значения типа `T`. Как и рассмотренный выше тип `Complex`, `Option` является универсальным типом – с помощью `Option<T>` можно представить факультативное значение произвольного типа `T`.

В нашем случае функция `escape_time` возвращает значение типа `Option<u32>`, показывающее, принадлежит ли `c` множеству Мандельброта, и если нет, то сколько итераций понадобилось, чтобы это установить. Если `c` не принадлежит множеству, то `escape_time` возвращает `Some(i)`, где `i` – номер итерации, на которой `z` вышла за пределы круга радиуса 2. В противном случае `c` теоретически может принадлежать множеству, и `escape_time` возвращает `None`.

```
for i in 0..limit {
```

В предыдущих примерах были показаны циклы `for` для обхода аргументов командной строки и элементов вектора. А в этом цикле `for` мы просто перебираем целые числа от `0` до `limit` (не включая последнее).

Метод `z.norm_sqr()` возвращает квадрат расстояния от `z` до начала координат. Чтобы решить, находится ли `z` вне круга радиуса 2, достаточно сравнить квадрат расстояния с `4.0`, а не вычислять квадратный корень.

Вы, вероятно, обратили внимание, что `///` обозначает строки комментария перед определением функции; комментарии перед членами структуры `Complex` тоже начинаются знаками `///`. Это *документирующие комментарии*; утилита `rustdoc` умеет разбирать их и описываемый ими код и порождает онлайновую документацию. Документация по стандартной библиотеке Rust составлена именно так. Подробно документирующие комментарии описаны в главе 8.

Оставшаяся часть программы решает, какую часть множества изображать при заданном разрешении, и распределяет работу между несколькими потоками для ускорения вычислений.

Разбор пары аргументов командной строки

Программе нужно несколько аргументов командной строки для управления разрешением создаваемого изображения и тем, какую часть множества Мандельброта изображать. Поскольку эти аргументы записываются в стандартном виде, мы написали функцию для их разбора.

```
use std::str::FromStr;

/// Разбирает строку `s`, содержащую пару координат, например: `"400x600"` или
/// `"1.0,0.5"`.
///
/// Точнее, `s` должна иметь вид <left><sep><right>, где <sep> - символ, заданный
/// в аргументе `separator`, а <left> и <right> - строки, допускающие разбор
/// методом `T::from_str`.
///
/// Если `s` удалось разобрать, то возвращает `Some<(x, y)>`, в противном случае
/// `None`.
fn parse_pair<T: FromStr>(s: &str, separator: char) -> Option<(T, T)> {
    match s.find(separator) {
        None => None,
        Some(index) => {
            match (T::from_str(&s[..index]), T::from_str(&s[index + 1..])) {
                (Ok(l), Ok(r)) => Some((l, r)),
                _ => None
            }
        }
```

```
    }
}

#[test]
fn test_parse_pair() {
    assert_eq!(parse_pair::<i32>("",        ','), None);
    assert_eq!(parse_pair::<i32>("10,",     ','), None);
    assert_eq!(parse_pair::<i32>(",10",     ','), None);
    assert_eq!(parse_pair::<i32>("10,20",   ','), Some((10, 20)));
    assert_eq!(parse_pair::<i32>("10,20xy", ','), None);
    assert_eq!(parse_pair::<f64>("0.5x",    'x'), None);
    assert_eq!(parse_pair::<f64>("0.5x1.5", 'x'), Some((0.5, 1.5)));
}
```

Определение `parse_pair` – пример универсальной функции:

```
fn parse_pair<T: FromStr>(s: &str, separator: char) -> Option<(T, T)> {
```

Часть `<T: FromStr>` можно прочитать так: «Для любого типа `T`, реализующего характеристику `FromStr` ...». По существу мы определяем целое семейство функций: `parse_pair::<i32>` разбирает пары значений типа `i32`, `parse_pair::<f64>` – пары значений с плавающей точкой и т. д. Это очень напоминает шаблон функции в C++. Программист на Rust назвал бы `T` *параметрическим типом* функции `parse_pair`. При использовании универсальной функции Rust зачастую способен вывести параметрические типы самостоятельно, так что указывать их явно, как в показанном выше коде, не нужно.

Функция возвращает значение типа `Option<(T, T)>`: либо `None`, либо `Some((v1, v2))`, где `(v1, v2)` – кортеж, содержащий два значения типа `T`. В `parse_pair` нет явного предложения `return`, т. е. возвращается значение последнего (и единственного) выражения в теле функции:

```
match s.find(separator) {
    None => None,
    Some(index) => {
        ...
    }
}
```

Метод `find` типа `String` ищет в строке символ `separator`. Если `find` возвращает `None`, значит, разделитель в строке не встречается, и тогда все выражение `match` принимает значение `None`, означающее, что разобрать строку не удалось. В противном случае разделитель встречается в позиции `index`.

```
match (T::from_str(&s[..index]), T::from_str(&s[index + 1..])) {
    (Ok(l), Ok(r)) => Some((l, r)),
    _ => None
}
```

Возможности выражения `match` начинают раскрываться в полной мере. Здесь аргументом `match` является кортеж:

```
(T::from_str(&s[..index]), T::from_str(&s[index + 1..]))
```

Выражения `&s[..index]` и `&s[index + 1..]` – это части строки до и после разделителя. Функция `from_str`, ассоциированная с параметрическим типом `T`, принимает

эти подстроки и пытается разобрать их как значения типа `T`. Если это получается, то возвращается кортеж, содержащий результаты. Именно с ним мы и производим сопоставление.

```
(Ok(l), Ok(r)) => Some((l, r)),
```

Сопоставление с этим образцом проходит, если оба элемента кортежа – варианты `Ok` значения типа `Result`, т. е. обе строки разобраны успешно. В таком случае выражение `match` принимает значение `Some((l, r))`, оно же и является значением функции.

```
_ => None
```

Всеобъемлющий образец _ сопоставляется со всем, чем угодно, значение при этом игнорируется. Если мы попали в эту точку, значит, функция `parse_pair` завершилась неудачно, поэтому ее значением будет `None`.

Имея функцию `parse_pair`, нетрудно написать функцию для разбора пары координат с плавающей точкой и вернуть их в виде значения типа `Complex<f64>`:

```
/// Разбирает пару чисел с плавающей точкой, разделенных запятой, и возвращает
/// ее в виде комплексного числа.
fn parse_complex(s: &str) -> Option<Complex<f64>> {
    match parse_pair(s, ',') {
        Some((re, im)) => Some(Complex { re, im }),
        None => None
    }
}

#[test]
fn test_parse_complex() {
    assert_eq!(parse_complex("1.25,-0.0625"),
               Some(Complex { re: 1.25, im: -0.0625 }));
    assert_eq!(parse_complex(",-0.0625"), None);
}
```

Функция `parse_complex` вызывает `parse_pair` и строит значение типа `Complex`, если координаты разобраны успешно, а в случае неудачи передает вызывающей стороне признак ошибки.

Внимательный читатель, вероятно, заметил, что для построения значения типа `Complex` мы воспользовались сокращенной нотацией. Очень часто поля структуры инициализируют одноименными переменными, поэтому Rust не настаивает на записи `Complex { re: re, im: im }`, а позволяет писать просто `Complex { re, im }`. За образец здесь взята аналогичная нотация в JavaScript и Haskell.

Отображение пикселей на комплексные числа

Наша программа должна работать с двумя взаимосвязанными пространствами координат: каждый пиксель выходного изображения соответствует точке на комплексной плоскости. Связь между этими пространствами зависит от того, какую часть множества Мандельброта мы собираемся нарисовать, а также от указанного в аргументах командной строки разрешения изображения. Следующая функция преобразует «пространство изображения» в «пространство комплексных чисел».

```
/// Зная строку и столбец пикселя выходного изображения, возвращает соответствующую
/// точку на комплексной плоскости.
///
/// `bounds` - пара, определяющая ширину и высоту изображения в пикселях.
/// `pixel` - пара (строка, столбец), определяющая конкретный пиксель изображения.
/// Параметры `upper_left` и `lower_right` - точки на комплексной плоскости,
/// описывающие область, покрываемую изображением.
fn pixel_to_point(bounds: (usize, usize),
                  pixel: (usize, usize),
                  upper_left: Complex<f64>,
                  lower_right: Complex<f64>)
    -> Complex<f64>
{
    let (width, height) = (lower_right.re - upper_left.re,
                           upper_left.im - lower_right.im);
    Complex {
        re: upper_left.re + pixel.0 as f64 * width  / bounds.0 as f64,
        im: upper_left.im - pixel.1 as f64 * height / bounds.1 as f64
        // Почему здесь вычитание? pixel.1 увеличивается при движении вниз,
        // тогда как мнимая часть увеличивается при движении вверх.
    }
}

#[test]
fn test_pixel_to_point() {
    assert_eq!(pixel_to_point((100, 100), (25, 75),
                              Complex { re: -1.0, im:  1.0 },
                              Complex { re:  1.0, im: -1.0 }),
               Complex { re: -0.5, im: -0.5 });
}
```

На рис. 2.3 иллюстрируется вычисление, выполняемое функцией `pixel_to_point`.

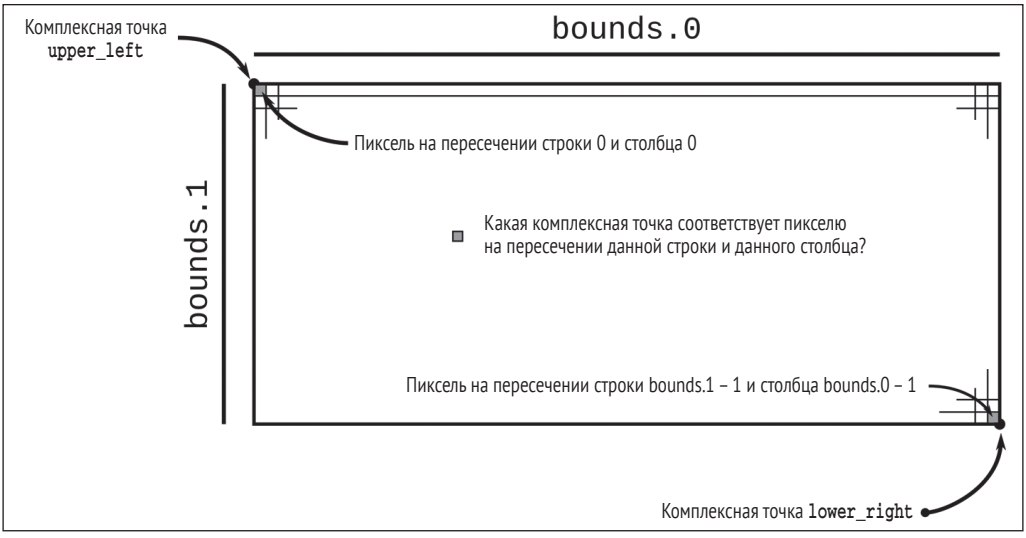


Рис. 2.3 ❖ Связь между комплексной плоскостью и пикселями изображения

Функция `pixel_to_point` производит простое вычисление, поэтому подробные объяснения мы опустим. Но есть несколько моментов, на которые стоит обратить внимание.

```
pixel.0
```

Выражения такого вида – это ссылки на элементы кортежа. В данном случае мы ссылаемся на первый элемент кортежа `pixel`.

```
pixel.0 as f64
```

Так в Rust обозначается преобразование типа: мы преобразуем `pixel.0` в значение типа `f64`. В отличие от C и C++, Rust обычно отказывается неявно преобразовывать числовые типы, вы должны указать преобразование явно. Это может показаться утомительным, но указывать, когда производить преобразование и какое именно, на удивление полезно. Неявные преобразования целых типов выглядят вполне невинно, но история учит, что они нередко бывали источниками ошибок и брешей в системе безопасности в реальных программах на C и C++.

Рисование множества

Чтобы нарисовать множество Мандельброта, мы для каждого пикселя изображения просто применяем функцию `escape_time` к соответствующей точке комплексной плоскости и рисуем пиксель цветом, зависящим от результата.

```
/// Рисует прямоугольную часть множества Мандельброта в буфере пикселей.
///
/// Аргумент `bounds` задает ширину и высоту буфера `pixels`, в котором каждый байт
/// представляет один полутоновый пиксель. Аргументы `upper_left` и `lower_right`
/// определяют точки на комплексной плоскости, соответствующие левому верхнему
/// и правому нижнему углам буфера пикселей.
fn render(pixels: &mut [u8],
          bounds: (usize, usize),
          upper_left: Complex<f64>,
          lower_right: Complex<f64>)
{
    assert!(pixels.len() == bounds.0 * bounds.1);

    for row in 0 .. bounds.1 {
        for column in 0 .. bounds.0 {
            let point = pixel_to_point(bounds, (column, row),
                                       upper_left, lower_right);
            pixels[row * bounds.0 + column] =
                match escape_time(point, 255) {
                    None => 0,
                    Some(count) => 255 - count as u8
                };
        }
    }
}
```

Тут ничего нового нет.

```
pixels[row * bounds.0 + column] =
    match escape_time(point, 255) {
```

```
        None => 0,
        Some(count) => 255 - count as u8
    };
```

Если `escape_time` сообщает, что точка `point` принадлежит множеству, то `render` рисует соответствующий пиксель черным цветом (`0`). В противном случае `render` назначает точке цвет тем темнее, чем больше времени ей понадобилось на выход из круга.

Запись файла изображения

Крейт `image` предоставляет функции для чтения и записи файлов в различных графических форматах, а также базовые функции для работы с изображениями. В частности, имеется кодировщик для формата PNG, в котором наша программа сохраняет результаты вычислений. Добавьте в раздел `[dependencies]` файла `Cargo.toml` такую строчку:

```
image = "0.13.0"
```

После этого можно написать такой код:

```
extern crate image;

use image::ColorType;
use image::png::PNGEncoder;
use std::fs::File;

/// Записывает буфер `pixels`, размеры которого заданы аргументом `bounds`, в файл
/// с именем `filename`.
fn write_image(filename: &str, pixels: &[u8], bounds: (usize, usize))
    -> Result<(), std::io::Error>
{
    let output = File::create(filename)?;

    let encoder = PNGEncoder::new(output);
    encoder.encode(&pixels,
                   bounds.0 as u32, bounds.1 as u32,
                   ColorType::Gray(8))?;

    Ok(())
}
```

Функция работает прямолинейно: открывает файл и пытается записать в него изображение. Мы передаем кодировщику данные пикселей из буфера `pixels`, его ширину и высоту из `bounds` и последний аргумент, который говорит, как интерпретировать байты в `pixels`: `ColorType::Gray(8)` означает, что каждый байт соответствует восьмиразрядному полутоновому пикселю.

Это все понятно. А интересно, как эта функция обрабатывает ошибки. Столкнувшись с ошибкой, мы должны сообщить об этом вызывающей функции. Как уже было сказано, функции, которые могут завершиться с ошибкой, должны возвращать значение типа `Result`, содержащее либо `Ok(s)` в случае успеха, где `s` – искомое значение, либо `Err(e)` в случае неудачи, где `e` – код ошибки. Так какие же типы обозначают успех и неудачу `write_image`?

Если все хорошо, то `write_image` нечего возвращать, она записала всю полезную информацию в файл. Поэтому в случае успеха возвращается «единичный» тип `()`,

который называется так, потому что существует единственное значение этого типа, обозначаемое также `()`. Единичный тип (unit type) аналогичен типу `void` в C и C++.

Ошибка может возникнуть либо потому, что `File::create` не смогла создать файл, либо потому, что `encoder.encode` не смогла записать в него изображение, т. е. операция ввода-вывода вернула код ошибки. Значение, возвращаемое функцией `File::create`, имеет тип `Result<std::fs::File, std::io::Error>`, а функцией `encoder.encode` – тип `Result<(), std::io::Error>`, т. е. тип ошибки в обоих случаях один и тот же, `std::io::Error`. В нашей функции `write_image` имеет смысл поступить так же.

Взгляните на обращение к `File::create`. Если эта функция возвращает `Ok(f)`, где `f` – успешно открытый файл типа `File`, то `write_image` может переходить к записи данных изображения в `f`. Если же `File::create` возвращает `Err(e)`, где `e` – некоторый код ошибки, то `write_image` должна сразу же вернуть `Err(e)` как собственное значение. Обращение к `encoder.encode` должно обрабатываться аналогично: ошибка приводит к немедленному возврату управления с передачей кода ошибки.

Оператор `?` специально введен для удобства таких проверок. Вместо того чтобы явно писать

```
let output = match File::create(filename) {
    Ok(f) => { f }
    Err(e) => { return Err(e); }
};
```

можно ограничиться гораздо более лаконичным эквивалентным кодом:

```
let output = File::create(filename)?;
```

Начинающие часто делают одну и ту же ошибку – употребляют оператор `?` в функции `main`. Но поскольку `main` не возвращает никакого значения, это работать не будет – нужно использовать метод `expect` объекта `Result`. Оператор `?` полезен только внутри функций, которые сами возвращают `Result`.

Мы воспользовались еще одним сокращением. Поскольку возвращаемый тип вида `Result<T, std::io::Error>` для некоторого типа `T` встречается очень часто – обычно он применяется в функциях ввода-вывода, – в стандартной библиотеке Rust определено сокращение для него. В модуле `std::io` имеются такие определения:

```
// Тип std::io::Error type.
struct Error { ... };

// Тип std::io::Result type, эквивалентен обычному типу `Result`, но специализирован таким
// образом, что типом ошибки является std::io::Error.
type Result<T> = std::result::Result<T, Error>
```

Если ввести это определение в область видимости с помощью объявления `use std::io::Result`, то можно будет записать тип, возвращаемый функцией `write_image`, более кратко – `Result<()>`. С такой формой вы часто будете встречаться, читая документацию по функциям из модулей `std::io`, `std::fs` и других.

Конкурентная программа рисования множества Мандельброта

Наконец, расставив все остальное по местам, мы можем продемонстрировать функцию `main`, в которой поставим себе на службу конкурентность. Сначала, для простоты, приведем неконкурентную версию:

```
use std::io::Write;

fn main() {
    let args: Vec<String> = std::env::args().collect();

    if args.len() != 5 {
        writeln!(std::io::stderr(),
                 "Порядок вызова: mandelbrot FILE PIXELS UPPERLEFT LOWERRIGHT")
            .unwrap();
        writeln!(std::io::stderr(),
                 "Пример: {} mandel.png 1000x750 -1.20,0.35 -1,0.20",
                 args[0])
            .unwrap();
        std::process::exit(1);
    }

    let bounds = parse_pair(&args[2], 'x')
        .expect("ошибка при разборе размеров изображения");
    let upper_left = parse_complex(&args[3])
        .expect("ошибка при разборе координат левого верхнего угла");
    let lower_right = parse_complex(&args[4])
        .expect("ошибка при разборе координат правого нижнего угла");

    let mut pixels = vec![0; bounds.0 * bounds.1];

    render(&mut pixels, bounds, upper_left, lower_right);

    write_image(&args[1], &pixels, bounds)
        .expect("ошибка при записи PNG-файла");
}
```

Собрав аргументы командной строки в вектор строк, мы разбираем каждый аргумент, а затем переходим к вычислениям.

```
let mut pixels = vec![0; bounds.0 * bounds.1];
```

Вызов макроса `vec![v; n]` создает вектор с `n` элементами, инициализированными значением `v`, так что этот код создает вектор нулей длиной `bounds.0 * bounds.1`, где `bounds` – разрешение изображения, заданное в командной строке. Этот вектор будет использоваться в качестве прямоугольного массива однобайтовых полутоновых пикселей, как показано на рис. 2.4.

Первая строчка, представляющая интерес:

```
render(&mut pixels, bounds, upper_left, lower_right);
```

Здесь вызывается функция `render`, вычисляющая изображение. Выражение `&mut pixels` заимствует изменяемую ссылку на наш буфер пикселей, давая `render` возможность заполнить его вычисленными полутоновыми значениями, при этом `pixels` остается владельцем вектора. В остальных аргументах передаются размеры изображения и прямоугольник на комплексной плоскости, который мы собираемся нарисовать.

```
write_image(&args[1], &pixels, bounds)
    .expect("error writing PNG file");
```

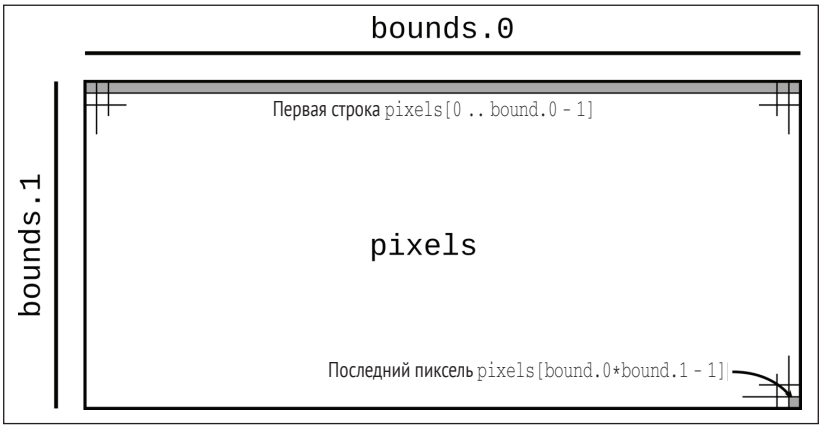


Рис. 2.4 ❖ Использование вектора как прямоугольного массива пикселей

Наконец, мы выводим буфер пикселей на диск в виде PNG-файла. В данном случае передается разделяемая (неизменяемая) ссылка на буфер, поскольку функция `write_image` не модифицирует содержимого буфера.

Чтобы распределить это вычисление между несколькими процессорами, проще всего разбить изображение на участки, по одному на процессор, и поручить каждому процессору раскраску назначенных ему пикселей. Для простоты разобьем изображение на горизонтальные полосы, как показано на рис. 2.5. Когда все процессоры закончат работу, мы запишем пиксели на диск.

Рис. 2.5 ❖ Разбиение буфера пикселей на полосы для параллельной отрисовки

Крейт `crossbeam` содержит ряд полезных средств конкурентности, в т. ч. *поток в области видимости* (scoped thread) – как раз то, что нам нужно. Чтобы воспользоваться этим крейтом, нужно добавить в файл `Cargo.toml` такую строчку:

```
crossbeam = "0.2.8"
```

Затем добавьте в начало файла `main.rs` строчку

```
extern crate crossbeam;
```

После этого строчку, в которой вызывается `render`, нужно заменить таким кодом:

```
let threads = 8;
let rows_per_band = bounds.1 / threads + 1;

{
    let bands: Vec<&mut [u8]> =
        pixels.chunks_mut(rows_per_band * bounds.0).collect();
    crossbeam::scope(|spawner| {
        for (i, band) in bands.into_iter().enumerate() {
            let top = rows_per_band * i;
            let height = band.len() / bounds.0;
            let band_bounds = (bounds.0, height);
            let band_upper_left =
                pixel_to_point(bounds, (0, top), upper_left, lower_right);
            let band_lower_right =
                pixel_to_point(bounds, (bounds.0, top + height), upper_left, lower_right);

            spawner.spawn(move || {
                render(band, band_bounds, band_upper_left, band_lower_right);
            });
        }
    });
}
```

Разберем этот код по частям.

```
let threads = 8;
let rows_per_band = bounds.1 / threads + 1;
```

Мы решили использовать восемь потоков[1]. Затем мы вычисляем число пикселей в каждой полосе. Поскольку высота полосы равна `rows_per_band`, а полная ширина изображения равна `bounds.0`, то площадь полосы в пикселях равна `rows_per_band * bounds.0`. Число строк округляется с избытком, чтобы полосы покрывали все изображение, даже если высота не делится нацело на `threads`.

```
let bands: Vec<&mut [u8]> = pixels.chunks_mut(rows_per_band * bounds.0).collect();
```

Делим буфер пикселей на полосы. Метод буфера `chunks_mut` возвращает итератор, порождающий изменяемые непересекающиеся участки буфера, каждый из которых охватывает `rows_per_band * bounds.0` пикселей, т. е. `rows_per_band` полных строк пикселей. Последний участок, порождаемый методом `chunks_mut`, может содержать меньше строк, но число пикселей во всех строках одно и то же. Наконец, метод итератора `collect` строит вектор, содержащий эти участки.

Теперь заставим поработать библиотеку `crossbeam`:

```
crossbeam::scope(|spawner| { ... });
```

Аргумент `|spawner| { ... }` – выражение *замыкания* в Rust. Замыкание – это нечто, что можно вызывать как функцию. В данном случае `|spawner|` – список аргументов, а `{ ... }` – тело функции. Заметим, что, в отличие от функций, объявленных с помощью ключевого слова `fn`, нет нужды объявлять типы аргументов замыкания, Rust сам выведет и их, и тип возвращаемого значения.

---

[1] В крейте `num_cpus` имеется функция, которая возвращает число доступных процессоров в системе.

Здесь `crossbeam::scope` вызывает замыкание, передавая в качестве аргумента `spawner` значение, которое замыкание сможет использовать для создания новых потоков. Функция `crossbeam::scope` ждет завершения всех потоков, а затем возвращает управление. При таком поведении Rust может гарантировать, что потоки не будут обращаться к своим участкам буфера `pixels` после того, как он покинет область видимости, а мы можем быть уверены, что после возврата из `crossbeam::scope` изображение полностью вычислено.

```
for (i, band) in bands.into_iter().enumerate() {
```

Здесь мы обходим полосы буфера пикселей. Итератор `into_iter()` делает каждую итерацию цикла исключительным владельцем одной полосы, гарантируя, что в каждый момент времени писать в нее может только один поток. Как это работает, мы подробно объясним в главе 5. Затем адаптер `enumerate` порождает кортежи, объединяющие элементы вектора с их индексами.

```
let top = rows_per_band * i;
let height = band.len() / bounds.0;
let band_bounds = (bounds.0, height);
let band_upper_left = pixel_to_point(bounds, (0, top), upper_left, lower_right);
let band_lower_right = pixel_to_point(bounds, (bounds.0, top + height),
                                      upper_left, lower_right);
```

Зная индекс и фактический размер полосы (напомним, что последняя полоса может быть короче остальных), мы можем построить ограничивающий прямоугольник – такой, который нужен функции `render`, но охватывающий только одну полосу буфера, а не все изображение. А функцию `pixel_to_point` будем использовать, чтобы найти на комплексной плоскости точки, соответствующие левому верхнему и правому нижнему углам полосы.

```
spawner.spawn(move || {
    render(band, band_bounds, band_upper_left, band_lower_right);
});
```

Наконец, создаем поток, исполняющий замыкание `move || { ... }`. Синтаксис выглядит немного странно: так обозначается замыкание без аргументов с телом `{ ... }`. Ключевое слово `move` в начале говорит, что это замыкание принимает владение используемыми в нем переменными; в частности, только замыкание может пользоваться изменяемым участком `band`.

Как уже было сказано, вызов `crossbeam::scope` гарантирует, что к моменту возврата из этой функции все потоки завершились, так что можно безопасно записать изображение в файл, чем мы и займемся в следующем разделе.

Выполнение программы рисования множества Мандельброта

В этой программе использовано несколько внешних крейтов: `num` для операций с комплексными числами, `image` для записи PNG-файлов и `crossbeam` – для создания потоков в области видимости. Вот как сейчас выглядит файл `Cargo.toml` с описанием всех зависимостей:

```
[package]
name = "mandelbrot"
version = "0.1.0"
authors = ["You <you@example.com>"]
```

```
[dependencies]
crossbeam = "0.2.8"
image = "0.6.1"
num = "0.1.27"
```

Теперь мы можем собрать и выполнить программу:

```
$ cargo build --release
    Updating registry `https://github.com/rust-lang/crates.io-index`
   Compiling bitflags v0.3.3
   ...
   Compiling png v0.4.3
   Compiling image v0.13.0
   Compiling mandelbrot v0.1.0 (file:///home/jimb/rust/mandelbrot)
    Finished release [optimized] target(s) in 42.64 secs
$ time target/release/mandelbrot mandel.png 4000x3000 -1.20,0.35 -1,0.20
real    0m1.750s
user    0m6.205s
sys     0m0.026s
$
```

Мы воспользовались программой Unix `time`, которая показывает, сколько времени работала программа; отметим, что хотя на вычисление изображения ушло более шести секунд, реально затрачено менее двух секунд. Закомментировав код сохранения изображения, вы легко убедитесь, что большая часть времени потрачена на запись файла. На ноутбуке, на котором тестировалась программа, конкурентная версия работала почти в четыре раза быстрее. В главе 19 мы покажем, как можно значительно улучшить этот результат.

Эта команда создает файл `mandel.png`, который можно открыть в системной программе просмотра или в браузере. Если все прошло нормально, то он будет выглядеть так:

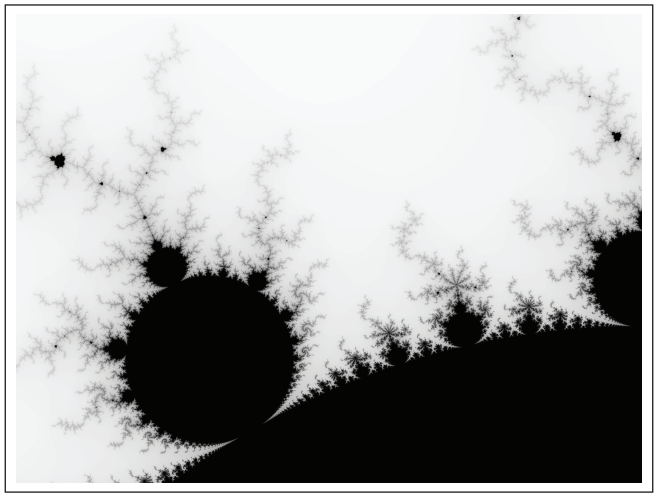


Рис. 2.6 ❖ Результат параллельного вычисления множества Мандельброта

Невидимая безопасность

Завершая главу, отметим, что написанная нами параллельная программа не так уж сильно отличается от того, что мы написали бы на любом другом языке: мы распределили участки буфера пикселей между процессорами, позволили им работать независимо, а по завершении работы представили результат. Так чем же тогда выделяется поддержка конкурентности в Rust?

За кадром остались программы, которые *невозможно* написать на Rust. Показанный выше код корректно распределяет буфер между потоками, но стоит немного изменить его, как возникнут гонки за данные, однако ни один из таких вариантов не пройдет статических проверок компилятора Rust. Компилятор C или C++ с радостью поможет вам в исследовании богатого множества программ с тонкими гонками за данные, но Rust сразу же предупредит об опасности.

В главах 4 и 5 мы опишем правила Rust, касающиеся безопасной работы с памятью. А в главе 19 объясним, как эти правила способствуют гигиене в части конкурентности. Но прежде нужно познакомиться с фундаментальными типами Rust. Это и будет темой следующей главы.

Глава 3

Базовые типы

> В мире много-много книг, и это хорошо, потому что разных людей тоже очень много, и каждый хочет читать что-то свое.
>
> — *Лемони Сникет*

В Rust типы служат нескольким целям:

- *безопасность*. Благодаря проверке типов компилятор Rust исключает целые классы типичных ошибок. За счет замены нулевых указателей и неконтролируемых объединений типобезопасными альтернативами Rust исключает даже ошибки, являющиеся частыми источниками аварийного завершения программы в других языках;
- *эффективность*. Программист может точно контролировать, как представляются значения в памяти, и выбирать типы, с которыми процессор заведомо работает эффективно. Программа не платит за излишнюю общность или гибкость;
- *лаконичность*. Всего вышеупомянутого Rust достигает без избыточного руководства со стороны программиста в виде явно указываемых типов. Rust-программы обычно не так загромождены типами, как аналогичные программы на C++.

В Rust нет ни интерпретатора, ни своевременной компиляции, вся программа транслируется в машинный код до начала выполнения. Типы помогают компилятору выбрать подходящее машинное представление значений, с которыми работает программа, – их производительность можно предсказать заранее, и они открывают доступ ко всем возможностям компьютера.

Rust – *статически типизированный* язык: не выполняя программу, компилятор проверяет, что на любом возможном пути выполнения значения используются только такими способами, которые совместимы с их типами. Поэтому многие ошибки программирования Rust отлавливает на ранней стадии, что очень важно для предоставляемых языком гарантий безопасности.

По сравнению с динамически типизированным языком, например JavaScript или Python, Rust требует более детального предварительного планирования: мы должны указывать типы параметров и возвращаемых значений функций, членов структур и еще нескольких конструкций. Однако благодаря двум особенностям Rust это не так страшно, как могло бы показаться.

- Зная явно указываемые вами типы, Rust *выводит* большую часть недосказанного. На практике часто бывает, что для некоторой переменной или вы-

ражения возможен лишь один тип, в таком случае Rust позволяет этот тип не указывать. Например, можно было бы явно указать все типы в функции:

```
fn build_vector() -> Vec<i16> {
    let mut v: Vec<i16> = Vec::<i16>::new();
    v.push(10i16);
    v.push(20i16);
    v
}
```

Но это громоздко и избыточно. Если мы знаем тип возвращаемого функцией значения, то очевидно, что `v` должно иметь тип `Vec<i16>` – вектор 16-разрядных целых со знаком, никакой другой тип не подойдет. А отсюда следует, что каждый элемент вектора должен иметь тип `i16`. Такого рода рассуждения лежат в основе механизма вывода типов в Rust, который позволяет записать ту же функцию короче:

```
fn build_vector() -> Vec<i16> {
    let mut v = Vec::new();
    v.push(10);
    v.push(20);
    v
}
```

Оба определения эквивалентны, в обоих случаях Rust генерирует один и тот же машинный код. Механизм вывода типов привносит удобочитаемость динамически типизированных языков, не жертвуя возможностью находить ошибки типизации на этапе компиляции.

- Функции могут быть *универсальными* (generic): если назначение и реализация функции достаточно общие, то можно определить ее так, что она будет работать для любого множества типов, удовлетворяющих необходимым условиям. Единственное определение применимо к неограниченному набору ситуаций. В языках Python и JavaScript все функции естественно работают таким образом: функция применима к любому значению, у которого имеются свойства и методы, необходимые функции для работы. (Этот подход часто называют «утиной типизацией»: если нечто квакает, как утка, то это и есть утка.) Но именно эта гибкость и усложняет раннее обнаружение ошибок типизации в таких языках; зачастую тестирование – единственный способ отловить подобные ошибки. Универсальные функции в Rust наделяют язык сходной степенью гибкости, оставляя возможность находить все ошибки типизации на этапе компиляции.

  Несмотря на гибкость, универсальные функции так же эффективны, как и обычные. Мы подробно остановимся на этой теме в главе 11.

Далее в этой главе рассматриваются типы Rust, начиная с простых машинных типов, целых чисел и чисел с плавающей точкой, и заканчивая составлением из них более сложных структур. Там, где это уместно, мы будем описывать представление значений типов в памяти и отмечать характеристики производительности.

Ниже перечислены типы, встречающиеся в Rust-программах. Показаны примитивные типы, наиболее употребительные типы из стандартной библиотеки и несколько примеров определенных пользователем типов.

Тип	Описание	Значения
`i8`, `i16`, `i32`, `i64`, `u8`, `u16`, `u32`, `u64`	Целые со знаком и без знака различной разрядности	`42`, `-5i8`, `0x400u16`, `0o100i16`, `20_922_789_888_000u64`, `b'*'` (байтовый литерал типа `u8`)
`isize`, `usize`	Целые со знаком и без знака того же размера, что машинный адрес (32 или 64 разряда)	`137`, `-0b0101_0010isize`, `0xffff_fc00usize`
`f32`, `f64`	Числа с плавающей точкой в формате IEEE, одинарной и двойной точности	`1.61803`, `3.14f32`, `6.0221e23f64`
`bool`	Булево значение	`true`, `false`
`char`	Символ Юникода шириной 32 разряда	`'*'`, `'\n'`, `'字'`, `'\x7f'`, `'\u{CA0}'`
`(char, u8, i32)`	Кортеж, допускаются различные типы	`('%', 0x7f, -1)`
`()`	«Единичный» (пустой) тип	`()`
`struct S { x: f32, y: f32 }`	Структура с именованными полями	`S { x: 120.0, y: 209.0 }`
`struct T(i32, char);`	Кортежеподобная структура	`T(120, 'X')`
`struct E;`	Структура, подобная единичному типу, не имеет полей	`E`
`enum Attend { OnTime, Late(u32) }`	Перечисление, алгебраический тип данных	`Attend::Late(5)`, `Attend::OnTime`
`Box<Attend>`	Бокс – владеет указателем на значение в куче	`Box::new(Late(15))`
`&i32`, `&mut i32`	Разделяемая и изменяемая ссылка – не владеющие памятью указатели, которые не должны существовать дольше, чем объекты, на которые указывают	`&s.y`, `&mut v`
`String`	Строка в кодировке UTF-8, размер динамически изменяется	`"ラーメン": ramen".to_string()`
`&str`	Ссылка на `str`, не владеющий памятью указатель на текст в кодировке UTF-8	`"そば": soba"`, `&s[0..12]`
`[f64; 4]`, `[u8; 256]`	Массив фиксированной длины, все элементы одного типа	`[1.0, 0.0, 0.0, 1.0]`, `[b' '; 256]`
`Vec<f64>`	Вектор переменной длины, все элементы одного типа	`vec![0.367, 2.718, 7.389]`
`&[u8]`, `&mut [u8]`	Ссылка на срезку: ссылка на участок массива, состоящая из указателя и длины	`&v[10..20]`, `&mut a[..]`
`&Any`, `&mut Read`	Объект характеристики, ссылка на любое значение, реализующее заданный набор методов	`value as &Any`, `&mut file as &mut Read`
`fn(&str, usize) -> isize`	Указатель на функцию	`i32::saturating_add`
(у замыкания нет явной формы)	Замыкание	`|a, b| a*a + b*b`

В этой главе рассматривается большинство типов за некоторыми исключениями:

- типам `struct` посвящена глава 9;
- перечислениям посвящена глава 10;
- характеристики описываются в главе 11;
- основная информация о типах `String` и `&str` приведена здесь, а подробности – в главе 17;
- типы функций и замыканий рассматриваются в главе 14.

Машинные типы

Фундамент системы типов в Rust составляет набор числовых типов фиксированной длины, подобранный так, что почти на всех современных процессорах они реализованы аппаратно, а также булев и символьный типы.

Имена числовых типов Rust строятся по общему образцу, в котором указаны разрядность и представление.

Таблица 3.1. Числовые типы Rust

Разрядность	Целое без знака	Целое со знаком	С плавающей точкой
8	u8	i8	
16	u16	i16	
32	u32	i32	f32
64	u64	i64	f64
машинное слово	usize	Isize	

Здесь под «машинным словом» понимается значение такого же размера, как размер адреса на машине, где исполняется код, обычно 32 или 64 разряда.

Целые типы

В случае типов целых чисел без знака весь доступный диапазон используется для представления положительных значений и нуля.

Тип	Диапазон
`u8`	от 0 до 2^8–1 (0 – 255)
`u16`	от 0 до 2^{16}–1 (0 – 65 535)
`u32`	от 0 до 2^{32}–1 (0 – 4 294 967 295)
`u64`	от 0 до 2^{64}–1 (0 – 18 446 744 073 709 551 615, или 18 квинтильонов)
`usize`	от 0 до 2^{32}–1 или 2^{64}–1

Для представления целых чисел со знаком используется дополнительный код, поэтому тем же комбинациям битов, что и в случае целых без знака, соответствует диапазон положительных и отрицательных чисел.

Тип	Диапазон
`i8`	от -2^7 до 2^7–1 (–128 – 127)
`i16`	от -2^{15} до 2^{15}–1 (–32 768 – 32 767)
`i32`	от -2^{31} до 2^{31}–1 (–2 147 483 648 – 2 147 483 647)
`i64`	от -2^{63} до 2^{63}–1 (–9 223 372 036 854 775 808 – 9 223 372 036 854 775 807)
`isize`	от -2^{31} до 2^{31}–1 или -2^{63} – 2^{63}–1

В Rust тип `u8` обычно применяется для представления «байтовых» значений. Например, чтение данных из файла или из сокета дает поток значений типа `u8`.

В отличие от C и C++, в Rust символьные и числовые типы различны, тип `char` не совпадает ни с `u8`, ни с `i8`. Тип `char` будет описан ниже в отдельном разделе.

Типы `usize` и `isize` аналогичны типам `size_t` и `ssize_t` в C и C++. Тип `usize` беззнаковый, а тип `isize` знаковый. Их длина зависит от размера адресного пространства

компьютера: в машинах с 32-разрядной архитектурой эти типы 32-разрядные, а в машинах с 64-разрядной архитектурой – 64-разрядные. Rust требует, чтобы индексы массива имели тип `usize`. Значения, представляющие размер массива или вектора либо число элементов в какой-то структуре данных, также обычно имеют тип `usize`.

В отладочных сборках Rust проверяет переполнение целочисленных типов в арифметических операциях.

```
let big_val = std::i32::MAX;
let x = big_val + 1;  // паника: переполнение в арифметической операции
```

В выпускных сборках такое сложение приведет к переходу через нуль, так что получится отрицательное число (в отличие от C++, в котором переполнение целого со знаком считается неопределенным поведением). Но если только вы не собираетесь навсегда отказываться от отладочных сборок, то мы не рекомендуем полагаться на эту особенность. Если вам нужна арифметика с переходом через нуль, то пользуйтесь таким методом:

```
let x = big_val.wrapping_add(1);  // ok
```

Целочисленные литералы в Rust могут снабжаться суффиксом, уточняющим тип: `42u8` – значение типа `u8`, а `1729isize` – типа `isize`. Суффикс можно опускать, и тогда Rust постарается вывести тип из контекста. Обычно тип при этом определяется однозначно, но иногда подходит несколько типов. В таком случае Rust принимает по умолчанию тип `i32`, если он входит в число альтернатив, а в противном случае сообщает о неоднозначности и считает это ошибкой.

Префиксы `0x`, `0o` и `0b` обозначают шестнадцатеричные, восьмеричные и двоичные литералы.

Чтобы было удобнее читать длинные числа, между цифрами можно вставлять знаки подчеркивания. Например, самое большое число типа `u32` можно записать в виде `4_294_967_295`. Где именно находятся подчерки, не важно, поэтому шестнадцатеричные или двоичные числа можно разбивать на группы по четыре цифры вместо трех, например `0xffff_ffff`, или отделять суффикс типа от цифр: `127_u8`.

Ниже приведено несколько примеров целочисленных литералов.

Литерал	Тип	Десятичное значение
`116i8`	`i8`	116
`0xcafeu32`	`u32`	51966
`0b0010_1010`	выведен	42
`0o106`	выведен	70

Хотя числовые типы и тип `char` различаются, в Rust тем не менее есть *байтовые литералы*, т. е. напоминающие символы литералы типа `u8`: `b'X'` – это код ASCII символа `X` в виде значения типа `u8`. Так, поскольку код ASCII буквы `A` равен 65, то литералы `b'A'` и `65u8` в точности эквивалентны. В байтовых литералах могут встречаться только символы, представленные в коде ASCII.

Существует несколько символов, которые не могут находиться после одиночной кавычки по причине синтаксической неоднозначности или трудности восприятия. Следующим символам должна предшествовать обратная косая черта:

Символ	Байтовый литерал	Числовой эквивалент
Одиночная кавычка '	`b'\''`	`39u8`
Обратная косая черта \	`b'\\'`	`92u8`
Новая строка	`b'\n'`	`10u8`
Возврат каретки	`b'\r'`	`13u8`
Табуляция	`b'\t'`	`9u8`

Если символ трудно записать или прочитать, то можно использовать его шестнадцатеричный код. Байтовый литерал вида `b'\xHH'`, где `HH` – две шестнадцатеричные цифры, представляет байт, значение которого равно `HH`. Например, управляющему символу «escape» с ASCII-кодом 37 (шестнадцатеричное 1B) соответствует байтовый литерал `b'\x1b'`. Поскольку байтовые литералы – просто альтернативная нотация для значений типа `u8`, выбирать стоит более простой вариант; записывать `b'\x1b'` вместо `27` имеет смысл, только чтобы подчеркнуть, что это ASCII-код.

Один целый тип можно преобразовать в другой с помощью оператора `as`. Преобразования рассматриваются в разделе «Приведение типов» главы 6, но вот несколько примеров:

```
assert_eq!(   10_i8  as u16,    10_u16); // принадлежит диапазону
assert_eq!( 2525_u16 as i16,  2525_i16); // принадлежит диапазону

assert_eq!(   -1_i16 as i32,    -1_i32); // с расширением знака
assert_eq!(65535_u16 as i32, 65535_i32); // с дополнением нулями

// Преобразования с выходом за границу конечного диапазона порождают
// значения, равные исходному по модулю 2^N, где N - разрядность
// конечного значения. Иногда эту операцию называют "усечением".
assert_eq!( 1000_i16 as  u8,   232_u8);
assert_eq!(65535_u32 as i16,    -1_i16);

assert_eq!(   -1_i8  as u8,   255_u8);
assert_eq!(  255_u8  as i8,    -1_i8);
```

Как и у любого значения, у целых чисел могут быть методы. Стандартная библиотека предлагает несколько базовых операций, о которых можно прочитать в документации. Отметим, что в документации имеются страницы как о самом типе (например, «`i32` (primitive type)»), так и о модуле, относящемся к этому типу («`std::u32`»).

```
assert_eq!(2u16.pow(4), 16);            // возведение в степень
assert_eq!((-4i32).abs(), 4);           // абсолютная величина
assert_eq!(0b101101u8.count_ones(), 4); // счетчик единиц
```

В этих примерах суффиксы типа обязательны: Rust не может найти методов значения, если не знает его типа. Но в реальном коде обычно имеется дополнительный контекст, достаточный для однозначного определения типа, так что суффиксы можно опускать.

Типы с плавающей точкой

Rust поддерживает типы с плавающей точкой с одинарной и двойной точностью в формате IEEE. Согласно спецификации IEEE 754-2008, эти типы включают положительную и отрицательную бесконечность, различные значения для представления положительного и отрицательного нуля, а также значение «не число».

Тип	Точность	Диапазон
f32	Одинарная точность IEEE (не менее 6 десятичных цифр)	Приблизительно от -3.4×10^{38} до $+3.4 \times 10^{38}$
f64	Двойная точность IEEE (не менее 15 десятичных цифр)	Приблизительно от -1.8×10^{308} до $+1.8 \times 10^{308}$

Типы Rust f32 и f64 соответствуют типам float и double в реализациях C и C++, поддерживающих спецификацию IEEE, и в Java, где эта спецификация всегда поддерживается.

В следующей таблице приведены примеры литералов с плавающей точкой.

Литерал	Тип	Значение
-1.5625	Выведен	-(1 9⁄16)
2.	Выведен	2
0.25	Выведен	1/4
1e4	Выведен	10000
40f32	f32	40
9.109_383_56e-31f64	f64	Приблизительно 9.10938356 × 10-31

В модулях стандартной библиотеки std::f32 и std::f64 определены константы для специальных значений: INFINITY, NEG_INFINITY (отрицательная бесконечность), NAN (не число), MIN и MAX (наименьшее и наибольшее представимые конечные значения).

В модулях std::f32::consts и std::f64::consts определены различные математические константы, в т. ч. E, PI и квадратный корень из двух. Типы f32 и f64 предоставляют полный набор методов для математических вычислений, например 2f64.sqrt() дает квадратный корень из двух с двойной точностью. В документации по стандартной библиотеке эти методы описаны в разделах «f32 (primitive type)» и «f64 (primitive type)». Вот несколько примеров:

```
assert_eq!(5f32.sqrt() * 5f32.sqrt(), 5.); // точно 5.0, в соответствии с IEEE
assert_eq!(-1.01f64.floor(), -1.0);
assert!((-1. / std::f32::INFINITY).is_sign_negative());
```

В реальном коде суффиксы обычно опускаются, т. к. тип определяется по контексту. Если же это невозможно, то сообщение об ошибке может выглядеть странно. Например, такой код не компилируется:

```
println!("{}", (2.0).sqrt());
```

И компилятор Rust при этом ругается:

```
error: no method named `sqrt` found for type `{float}` in the current scope
```

Это может показаться удивительным: где же еще нужно искать метод sqrt, как не в типе с плавающей точкой? Для исправления ошибки нужно явно указать тип – тем или иным способом:

```
println!("{}", (2.0_f64).sqrt());
println!("{}", f64::sqrt(2.0));
```

В отличие от C и C++, Rust не производит почти никаких числовых преобразований неявно. Если функция ожидает аргумента типа f64, то передача аргумента типа i32 считается ошибкой. Rust даже отказывается неявно преобразовывать

значения типа i16 в тип i32, хотя любое значение типа i16 является в то же время значением типа i32. Но ключевое слово здесь «неявно» – явное преобразование всегда можно выразить с помощью оператора as: i as f64 или x as i32. Из-за отсутствия неявных преобразований программа на Rust иногда оказывается более пространной, чем аналогичная программа на C или C++. Но у неявных преобразований целых чисел печальная история ошибок и брешей в системе безопасности. Наш опыт показывает, что благодаря явному указанию числовых преобразований в Rust мы стали обращать внимание на проблемы, которые в противном случае остались бы незамеченными. Преобразованиям посвящен раздел «Приведение типов» главы 6.

Тип bool

Булев тип в Rust, bool, принимает два обычных для этого типа значения: true и false. Операторы сравнения, например == и <, порождают результаты типа bool: значение выражения 2 < 5 равно true.

Многие языки снисходительно относятся к использованию значений других типов в контексте, где требуется булево значение. C и C++ неявно преобразуют символы, целые числа, числа с плавающей точкой и указатели в булев тип, когда они встречаются в предложении if или while. Python допускает строки, списки, словари и даже множества в булевых контекстах, считая, что значение равно true, если соответствующий объект не пуст. Но Rust в этом отношении ведет себя очень строго: в управляющих конструкциях типа if и while условием должно быть булево выражение, как и в закороченных логических операторах && и ||. Необходимо писать if x != 0 { ... }, а не просто if x { ... }.

Оператор as может преобразовать значение типа bool в целое:

```
assert_eq!(false as i32, 0);
assert_eq!(true  as i32, 1);
```

Однако преобразование в обратном направлении, из числового типа в bool, невозможно. Необходимо записывать сравнение явно, например x != 0. Хотя для представления типа bool нужен только один бит, в Rust для булева значения отводится целый байт в памяти, так что на него можно создать указатель.

Символы

Тип char в Rust представляет один символ Юникода в виде 32-разрядного значения.

Тип char используется для одиночных символов, а для представления строк и потоков текста применяется кодировка UTF-8. Таким образом, тип String представляет текст в виде последовательности байтов в кодировке UTF-8, а не в виде массива символов.

Символьные литералы – это символы, заключенные в одиночные кавычки, например '8' или '!'. Допускаются любые символы Юникода, так, символьный литерал '錆' представляет японский иероглиф «sabi» (ржавчина, rust).

Как и в случае байтовых литералов, некоторым символам должен предшествовать знак \:

Символ	Символьный литерал
Одиночная кавычка '	`'\''`
Обратная косая черта \	`'\\'`
Новая строка	`'\n'`
Возврат каретки	`'\r'`
Табуляция	`'\t'`

Если вам так больше нравится, можете записывать кодовую позицию Юникода в шестнадцатеричном виде.

- Если кодовая позиция находится в диапазоне от U+0000 до U+007F (т. е. принадлежит набору символов ASCII), то символ можно записывать в виде `'\xHH'`, где `HH` – две шестнадцатеричные цифры. Например, символьные литералы `'*'` и `'\x2A'` эквивалентны, поскольку кодовая позиция символа `*` равна 42, или 2A в шестнадцатеричном виде.
- Любой символ Юникода можно записать в виде `'\u{HHHHHH}'`, где `HHHHHH` – шестнадцатеричное число, содержащее от одной до шести цифр. Например, символьный литерал `'\u{CA0}'` представляет символ «ಠ» языка каннада, используемый для обозначения неодобрения в Юникоде: «ಠ_ಠ». Тот же самый литерал можно записать просто как `'ಠ'`.

В типе `char` всегда хранится кодовая позиция Юникода из диапазона 0x0000—0xD7FF или 0xE000—0x10FFFF. В нем никогда не хранится половина суррогатной пары (т. е. кодовая позиция из диапазона 0xD800—0xDFFF) или значение, выходящее за пределы кодового пространства Юникода (большее 0x10FFFF). Для контроля допустимости значений типа `char` в Rust используется как система типов, так и динамические проверки.

Rust никогда не производит неявных преобразований между типом `char` и другими типами. Для преобразования `char` в целый тип можно использовать оператор `as`; если разрядность конечного типа меньше 32, то старшие биты отбрасываются:

```
assert_eq!('*' as i32, 42);
assert_eq!('ಠ' as u16, 0xca0);
assert_eq!('ಠ' as i8, -0x60); // U+0CA0 усечено до восьми бит, со знаком
```

Обратное преобразование в `char` допускает только тип `u8`; оператор `as` в Rust выполняет только простые преобразования, которые не могут завершиться ошибкой, однако любой целый тип, кроме `u8`, содержит значения, не являющиеся допустимыми кодовыми позициями Юникода, поэтому для совершения преобразования потребовалась бы проверка во время выполнения. Но в стандартной библиотеке есть функция `std::char::from_u32`, которая принимает произвольное значение типа `u32` и возвращает значение типа `Option<char>`: если аргумент не является допустимой кодовой позицией Юникода, то `from_u32` возвращает `None`, иначе `Some(c)`, где `c` – соответствующее значение типа `char`.

Стандартная библиотека предоставляет ряд полезных методов для работы с символами, о которых можно прочитать в документации, поискав раздел «char (primitive type)» и модуль «`std::char`». Например:

```
assert_eq!('*'.is_alphabetic(), false);
assert_eq!('β'.is_alphabetic(), true);
assert_eq!('8'.to_digit(10), Some(8));
assert_eq!('♉'.len_utf8(), 3);
assert_eq!(std::char::from_digit(2, 10), Some('2'));
```

Естественно, одиночные символы не так интересны, как строки и потоки текста. Стандартный тип `String` и общие вопросы работы с текстом будут рассмотрены ниже в этой главе.

Кортежи

Кортежем называется пара, тройка, четверка и т. д. значений различных типов. Кортеж можно записать в виде последовательности элементов, разделенных запятыми и заключенных в круглые скобки. Например, `("Brazil", 1985)` – кортеж, первым элементом которого является статически распределенная строка, а вторым – целое число, он имеет тип `(&str, i32)` (на втором месте может быть другой целый тип, выведенный Rust для литерала `1985`). Имея значение кортежа `t`, к его элементам можно обращаться так: `t.0`, `t.1` и т. д.

Кортежи мало похожи на массивы. Во-первых, элементы кортежа могут иметь разные типы, тогда как в массиве все элементы должны быть одного типа. Во-вторых, индексами кортежа могут быть только константы, например `t.4`. Нельзя написать `t.i` или `t[i]`, чтобы получить `i`-й элемент.

В Rust кортежи часто используются для возврата нескольких значений из функции. Например, метод `split_at`, который делит строку пополам и возвращает обе половины, объявлен следующим образом:

```
fn split_at(&self, mid: usize) -> (&str, &str);
```

Тип возвращаемого значения `(&str, &str)` – кортеж, состоящий из двух подстрок. Каждый элемент возвращенного значения можно присвоить отдельной переменной:

```
let text = "I see the eigenvalue in thine eye";
let (head, tail) = text.split_at(21);
assert_eq!(head, "I see the eigenvalue ");
assert_eq!(tail, "in thine eye");
```

Это короче и понятнее, чем эквивалентный код:

```
let text = "I see the eigenvalue in thine eye";
let temp = text.split_at(21);
let head = temp.0;
let tail = temp.1;
assert_eq!(head, "I see the eigenvalue ");
assert_eq!(tail, "in thine eye");
```

Кортежи часто выступают в роли минималистского структурного типа. Например, в программе рисования множества Мандельброта из главы 2 нужно было передавать ширину и высоту изображения функциям рисования и записи на диск. Можно было бы объявить структуру с членами `width` и `height`, но это слишком громоздко для столь тривиальной задачи, поэтому мы воспользовались кортежем:

```
/// Записывает буфер `pixels`, размеры которого заданы аргументом `bounds`, в файл
/// с именем `filename`.
fn write_image(filename: &str, pixels: &[u8], bounds: (usize, usize))
    -> Result<(), std::io::Error>
{ ... }
```

Тип параметра `bounds` – `(usize, usize)`, т. е. кортеж из двух значений типа `usize`. Конечно, можно было явно выделить параметры `width` и `height`, и машинный код при этом был бы почти такой же. Вопрос только в понятности. Мы считаем, что размер – это одно значение, а не два, и использование кортежа позволяет выразить такой взгляд.

Как это ни удивительно, еще одно распространенное применение кортежа – нуль-кортеж `()`. Его принято называть «единичным типом», потому что существует всего одно значение этого типа, которое тоже записывается `()`. В Rust единичный тип используется тогда, когда никакого осмысленного значения нет, но контекст требует наличия какого-то типа.

Например, функция, не возвращающая значения, имеет тип `()`. Функция `std::mem::swap` из стандартной библиотеки ничего не возвращает, она просто обменивает местами два своих аргумента. Ее объявление выглядит так:

```
fn swap<T>(x: &mut T, y: &mut T);
```

Здесь `<T>` означает, что функция `swap` универсальная: ее можно применять к ссылкам на значения произвольного типа `T`. Но тип возвращаемого значения в сигнатуре `swap` вообще не указан, это сокращенный способ сказать, что она возвращает единичный тип:

```
fn swap<T>(x: &mut T, y: &mut T) -> ();
```

Аналогично написанная нами выше функция `write_image` возвращает значение типа `Result<(), std::io::Error>`; это означает, что в случае ошибки возвращается `std::io::Error`, а в случае успеха ничего.

Допускается запятая после последнего элемента кортежа: типы `(&str, i32,)` и `(&str, i32)` эквивалентны, как и выражения `("Brazil", 1985,)` и `("Brazil", 1985)`. Rust допускает необязательную запятую в конце всюду, где используются запятые: в аргументах функций, в массивах, в определениях структур и перечислений и т. д. Человеку это может показаться странным, но упрощает сопровождение в случае, когда элементы могут добавляться в список и удаляться из него.

Для единообразия допускаются даже кортежи, содержащие всего одно значение. Литерал `("lonely hearts",)` – кортеж, содержащий единственную строку, его тип – `(&str,)`. Здесь запятая после значения обязательна, чтобы отличить кортеж с одним элементом от простого выражения в скобках.

Указательные типы

В Rust есть несколько типов, представляющих адреса в памяти.

Между Rust и большинством языков существует большая разница в части сборки мусора. В Java, если класс `Tree` содержит поле `Tree left;`, то `left` является ссылкой на другой, отдельно созданный объект типа `Tree`. В Java объекты никогда не содержат других объектов целиком.

В Rust дело обстоит иначе. При проектировании языка ставилась задача свести к минимуму динамическое выделение памяти. По умолчанию значения вкладываются друг в друга. Значение `((0, 0), (1440, 900))` хранится как четыре соседних числа. Если сохранить его в локальной переменной, то ее длина будет равна суммарной длине четырех целых чисел. Память из кучи не выделяется.

Это замечательно с точки зрения эффективности использования памяти, но в результате, когда Rust-программе нужен указатель на другое значение, мы должны явно пользоваться указательными типами. Зато на указательные типы в безопасном коде на Rust накладываются ограничения, исключающие неопределенное поведение, поэтому использовать их безопасно гораздо проще, чем в C++.

Мы обсудим три указательных типа: ссылки, боксы и небезопасные указатели.

Ссылки

Значение типа `&String` – ссылка на значение типа `String`, `&i32` – ссылка на `i32` и т. д.

Для начала ссылки проще всего представлять себе как базовый указательный тип в Rust. Ссылка может указывать на любое значение в любом месте: в стеке или в куче. Выражение `&x` порождает ссылку на `x`; в Rust говорят, что оно «заимствует ссылку на `x`». Если имеется ссылка `r`, то выражение `*r` дает значение, на которое указывает `r`. Все это очень похоже на операторы `&` и `*` в C и C++. И точно так же, как указатель в C, при выходе ссылки из области видимости ресурсы автоматически не освобождаются.

Но, в отличие от указателей в C, ссылки в Rust никогда не принимают значения `null`: в безопасном Rust попросту не существует способа создать нулевую ссылку. Кроме того, ссылки в Rust по умолчанию неизменяемы:

- `&T` – неизменяемая ссылка, как `const T*` в C;
- `&mut T` – изменяемая ссылка, как `T*` в C.

Еще одно важное отличие состоит в том, что Rust отслеживает владельца и время жизни значений, поэтому ошибки вроде висячих указателей, двойного освобождения памяти и недействительных указателей обнаруживаются на этапе компиляции. Правила безопасности для ссылок рассматриваются в главе 5.

Боксы

Самый простой способ выделить память для значения из кучи – воспользоваться функцией `Box::new`.

```
let t = (12, "eggs");
let b = Box::new(t);  // выделить память для кортежа из кучи
```

Переменная `t` имеет тип `(i32, &str)`, поэтому тип `b` – `Box<(i32, &str)>`. Функция `Box::new()` выделяет память, достаточную для хранения кортежа в куче. Когда `b` покидает область видимости, память сразу же освобождается, если только `b` не *передана* – например, посредством возврата из функции. Передача (move) – это основа механизма управления памятью, выделенной из кучи, в Rust; мы подробно рассмотрим его в главе 4.

Простые указатели

В Rust имеются также простые указательные типы `*mut T` и `*const T`. Они работают так же, как указатели в C++. Использование простых указателей небезопасно, по-

скольку Rust не отслеживает, на что они указывают. Так, простой указатель может быть нулевым или указывать на область памяти, которая уже освобождена или содержит значение другого типа. Тут открывается простор для всех классических ошибок работы с указателями в C++.

Однако разыменовывать простой указатель можно только внутри `unsafe`-блока. `Unsafe`-блоки – механизм Rust, позволяющий использовать продвинутые возможности языка на страх и риск программиста. Если в программе нет `unsafe`- блоков (или те, что есть, написаны корректно), то все гарантии безопасности, описанные в этой книге, действуют. Детали см. в главе 21.

Массивы, векторы и срезки

В Rust есть три типа для представления последовательностей значений в памяти:

- тип `[T; N]` представляет массив `N` значений типа `T`. Размер массива фиксирован, задается на этапе компиляции и является частью типа. Увеличить или уменьшить массив невозможно;
- тип `Vec<T>`, называемый «вектором `T`», предназначен для динамически выделяемых растущих последовательностей значений типа `T`. Элементы вектора находятся в куче, так что его размер можно изменять: добавлять в конец отдельные элементы или целые векторы, удалять элементы и т. д.;
- типы `&[T]` и `&mut [T]`, называемые «разделяемыми срезками `T`» или «изменяемыми срезками `T`», – это ссылки на ряд соседних элементов, являющийся частью какой-то другой последовательности, например массива или вектора. Можно считать, что срезка – это указатель на первый элемент плюс счетчик доступных элементов после него. Изменяемая срезка `&mut [T]` позволяет читать и изменять элементы, но не может быть общей; разделяемая срезка `&[T]` позволяет обращаться к элементам сразу нескольким читателям, но запрещает модифицировать элементы.

Если `v` – значение любого из этих трех типов, то выражение `v.len()` дает количество элементов в `v`, а `v[i]` ссылается на `i`-й элемент `v`. Первый элемент `v` равен `v[0]`, последний – `v[v.len() - 1]`. Rust проверяет, что `i` находится в этом диапазоне; если это не так, выражение паникует. Длина `v` может быть равна нулю, в этом случае любая попытка обратиться к его элементу по индексу приводит к панике. Индекс `i` должен быть значением типа `usize`, никакие другие целые типы не допускаются.

Массивы

Есть несколько способов записать значение массива. Самый простой – перечислить значения в квадратных скобках:

```
let lazy_caterer: [u32; 6] = [1, 2, 4, 7, 11, 16];
let taxonomy = ["Animalia", "Arthropoda", "Insecta"];

assert_eq!(lazy_caterer[3], 7);
assert_eq!(taxonomy.len(), 3);
```

В типичном случае, когда длинный массив заполнен каким-то одним значением, можно написать `[V; N]`, где `V` – значение каждого элемента, а `N` – длина массива. Например, `[true; 10000]` – массив из 10 000 элементов типа `bool`, равных `true`:

```
let mut sieve = [true; 10000];
for i in 2..100 {
    if sieve[i] {
        let mut j = i * i;
        while j < 10000 {
            sieve[j] = false;
            j += i;
        }
    }
}

assert!(sieve[211]);
assert!(!sieve[9876]);
```

Такой синтаксис часто встречается, когда нужно определить буфер фиксированной длины: `[0u8; 1024]` – буфер размером 1 килобайт, заполненный нулевыми байтами. В Rust нет нотации для неинициализированных массивов (вообще, Rust гарантирует, что код ни при каких обстоятельствах не будет работать с неинициализированными значениями).

Длина массива является частью его определения и фиксируется на этапе компиляции. Если `n` – переменная, то нельзя написать `[true; n]`, чтобы получить массив с `n` элементами. Если необходим массив, длина которого может изменяться во время выполнения, то пользуйтесь вектором.

Все полезные методы, применяемые к массивам, – итерирование, поиск, сортировка, заполнение, фильтрация и т. п. – определены как методы срезок, а не массивов. Но поскольку Rust неявно преобразует ссылку на массив в срезку, когда ищет методы, то любой метод срезки можно применять прямо к массиву:

```
let mut chaos = [3, 5, 4, 1, 2];
chaos.sort();
assert_eq!(chaos, [1, 2, 3, 4, 5]);
```

Здесь метод `sort` определен для срезок, но поскольку `sort` принимает операнд по ссылке, мы можем применить его непосредственно к массиву `chaos`, при этом неявно создается срезка `&mut[i32]`, ссылающаяся на весь массив. Метод `len`, который мы уже упоминали, также является методом срезки.

Подробнее срезки будут рассмотрены ниже в этой главе.

Вектор

Вектор `Vec<T>` – это массив переменной длины, содержащий элементы типа `T`, память для которого выделена из кучи.

Вектор можно создать несколькими способами. Самое простое – воспользоваться макросом `vec!`, который синтаксически очень похож на создание литерального массива:

```
let mut v = vec![2, 3, 5, 7];
assert_eq!(v.iter().fold(1, |a, b| a * b), 210);
```

Но, конечно, это не массив, а вектор, так что в него можно добавлять элементы динамически:

```
v.push(11);
v.push(13);
assert_eq!(v.iter().fold(1, |a, b| a * b), 30030);
```

Можно также построить вектор, повторяя некоторое значение заданное число раз. Этот способ также похож на инициализацию литеральных массивов:

```
fn new_pixel_buffer(rows: usize, cols: usize) -> Vec<u8> {
    vec![0; rows * cols]
}
```

Макрос `vec!` эквивалентен вызову функции `Vec::new`, которая создает пустой вектор, с последующим добавлением элементов:

```
let mut v = Vec::new();
v.push("step");
v.push("on");
v.push("no");
v.push("pets");
assert_eq!(v, vec!["step", "on", "no", "pets"]);
```

Вектор можно также построить из значений, порождаемых итератором:

```
let v: Vec<i32> = (0..5).collect();
assert_eq!(v, [0, 1, 2, 3, 4]);
```

Как правило, при использовании метода `collect` нужно указывать тип, как в этом примере, потому что `collect` умеет строить разные коллекции, а не только векторы. Явно задав тип `v`, мы недвусмысленно сообщили, какая коллекция нам нужна.

Для векторов, как и для массивов, можно использовать методы срезок:

```
// Палиндром!
let mut v = vec!["a man", "a plan", "a canal", "panama"];
v.reverse();
// Разумный, но разочаровывающий:
assert_eq!(v, vec!["panama", "a canal", "a plan", "a man"]);
```

Здесь метод `reverse` в действительности определен для срезок, но в момент вызова у вектора неявно заимствуется ссылка `&mut [&str]`, и для нее-то и вызывается `reverse`.

Без типа `Vec` в Rust не обойтись – он используется почти всюду, где нужен список динамического размера, – поэтому есть еще много методов, которые конструируют новые векторы или расширяют существующие. Мы рассмотрим их в главе 16.

Тип `Vec<T>` состоит из трех компонентов: указатель на выделенный в куче буфер для хранения элементов, максимальное число элементов в буфере (емкость) и число элементов, находящихся в буфере в данный момент (длина). Если буфер заполнен, то при добавлении еще одного элемента выделяется память для большего буфера, содержимое старого буфера копируется в новый, обновляется указатель на буфер и емкость, после чего старый буфер освобождается.

Если заранее известно, сколько элементов будет храниться в векторе, то вместо функции `Vec::new` можно вызвать функцию `Vec::with_capacity`, которая с самого начала создаст вектор с достаточно большим буфером, тогда элементы можно будет добавлять по одному, не опасаясь перераспределения памяти. Макрос `vec!` примерно

так и поступает, потому что он знает, сколько элементов будет в векторе. Отметим, что таким образом устанавливается только начальный размер вектора; если впоследствии он окажется недостаточным, то буфер вектора увеличится, как обычно.

Во многих библиотечных функциях при любой возможности используется функция `Vec::with_capacity`, а не `Vec::new`. Так, в примере функции `collect` выше итератор `0..5` заранее знает, что отдаст пять значений, и `collect` пользуется этим, чтобы с самого начала выделить вектору память достаточного объема. Как это работает, мы расскажем в главе 15.

Если метод вектора `len` возвращает число элементов, хранящихся в нем в данный момент, то метод `capacity` сообщает, сколько элементов может поместиться в векторе без перераспределения памяти:

```
let mut v = Vec::with_capacity(2);
assert_eq!(v.len(), 0);
assert_eq!(v.capacity(), 2);

v.push(1);
v.push(2);
assert_eq!(v.len(), 2);
assert_eq!(v.capacity(), 2);

v.push(3);
assert_eq!(v.len(), 3);
assert_eq!(v.capacity(), 4);
```

В вашей программе емкости могут отличаться от показанных выше. Тип `Vec` и системный распределитель кучи могут округлять запрошенный объем памяти, даже в случае функции `with_capacity`.

Элементы можно вставлять в любое место вектора и удалять из любого места, хотя при этом все элементы после точки вставки или удаления сдвигаются вперед или назад, так что для длинного вектора операция может работать долго:

```
let mut v = vec![10, 20, 30, 40, 50];

// Вставить в позицию 3 элемент 35.
v.insert(3, 35);
assert_eq!(v, [10, 20, 30, 35, 40, 50]);

// Удалить элемент в позиции 1.
v.remove(1);
assert_eq!(v, [10, 30, 35, 40, 50]);
```

Метод `pop` удаляет последний элемент вектора и возвращает его. Точнее, удаление из вектора `Vec<T>` возвращает значение типа `Option<T>`: `None`, если вектор уже был пуст, и `Some(v)`, если последним элементом был `v`.

```
let mut v = vec!["carmen", "miranda"];
assert_eq!(v.pop(), Some("miranda"));
assert_eq!(v.pop(), Some("carmen"));
assert_eq!(v.pop(), None);
```

Вектор можно обойти в цикле `for`:

```
// Получить аргументы командной строке в виде вектора строк.
let languages: Vec<String> = std::env::args().skip(1).collect();
for l in languages {
```

```
    println!("{}: {}", l,
             if l.len() % 2 == 0 {
                 "функциональный"
             } else {
                 "императивный"
             });
}
```

Смысл этой программы выясняется, если передать ей список языков программирования:

```
$ cargo run Lisp Scheme C C++ Fortran
   Compiling fragments v0.1.0 (file:///home/jimb/rust/book/fragments)
    Running `.../target/debug/fragments Lisp Scheme C C++ Fortran`
Lisp: функциональный
Scheme: функциональный
C: императивный
C++: императивный
Fortran: императивный
$
```

Вот наконец-то удовлетворительное определение термина «функциональный язык».

Несмотря на свою фундаментальную роль, `Vec` – обычный тип Rust, не встроенный в сам язык. О том, как определяются такие типы, мы поговорим в главе 21.

Поэлементное построение векторов

Строить векторы по одному элементу не так уж плохо, как кажется. Всякий раз, как емкость вектора оказывается исчерпанной, выделяется память для нового буфера в два раза больше прежнего. Допустим, что вначале буфер вектора мог содержать всего один элемент. По мере роста размер буфера будет принимать значения 1, 2, 4, 8, … 2^n. Вспомнив формулу суммы геометрической прогрессии, мы увидим, что суммарный размер всех предыдущих буферов равен 2^n-1, очень близко к размеру последнего буфера. Поскольку в среднем число элементов в буфере не меньше его емкости, вектор выполняет не более одного копирования на каждый элемент!

А это означает, что применение `Vec::with_capacity` вместо `Vec::new` дает лишь увеличение скорости в постоянное число раз, а не улучшение алгоритма. Впрочем, для небольших векторов возможность избежать нескольких вызовов распределителя кучи может дать ощутимую разницу в производительности.

Срезки

Срезка, записываемая в виде `[T]` без указания длины, – это участок массива или вектора. Поскольку длина срезки может быть любой, они не хранятся в переменных и не передаются в виде аргументов функции. Срезки всегда передаются по ссылке.

Ссылка на срезку является *толстым указателем*, она занимает два слова: указатель на первый элемент срезки и количество элементов в ней.

Рассмотрим следующий код:

```
let v: Vec<f64> = vec![0.0,  0.707,  1.0,  0.707];
let a: [f64; 4] =     [0.0, -0.707, -1.0, -0.707];

let sv: &[f64] = &v;
let sa: &[f64] = &a;
```

В последних двух строках Rust автоматически преобразует ссылки `&Vec<f64>` и `&[f64; 4]` в ссылки на срезки, которые указывают непосредственно на данные.

В конце память выглядит следующим образом:

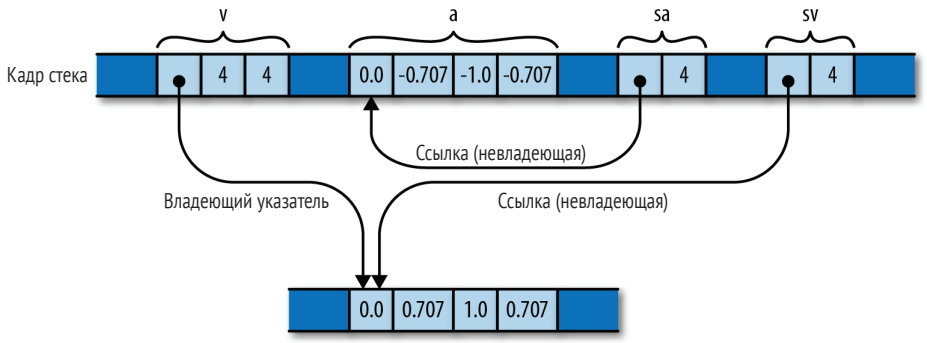


Рис. 3.1 ❖ Вектор v и массив a в памяти; на них ссылаются срезки sv и sa

Если обычная ссылка – это невладеющий указатель на одно значение, то ссылка на срезку – невладеющий указатель на несколько значений. Поэтому ссылка на срезку – удачный выбор, когда нужно написать функцию, работающую с последовательностью однородных данных, будь то массив или вектор, в стеке или в куче. Вот, например, функция, которая печатает срезку чисел, по одному в строке:

```
fn print(n: &[f64]) {
    for elt in n {
        println!("{}", elt);
    }
}

print(&v);  // работает с векторами
print(&a);  // работает с массивами
```

Поскольку эта функция принимает ссылку на срезку, ее можно применить как к вектору, так и к массиву. Вообще, многие методы, которые на первый взгляд кажутся методами векторов или массивов, на самом деле определены для срезок. Например, методы `sort` и `reverse`, которые сортируют и обращают последовательность элементов на месте, в действительности являются методами типа срезки `[T]`.

Для получения ссылки на срезку массива, вектора или уже существующей срезки следует указать диапазон в качестве индекса:

```
print(&v[0..2]);   // напечатать первые два элемента v
print(&a[2..]);    // напечатать элементы a, начиная с a[2]
print(&sv[1..3]);  // напечатать v[1] и v[2]
```

Как и при обычном доступе к массиву по индексу, Rust проверяет корректность индексов. Попытка заимствовать срезку, выходящую за конец данных, приводит к панике.

Мы часто употребляем слово «срезка» для ссылочных типов вида `&[T]` или `&str`, но надо понимать, что это сокращение: правильно называть их ссылками на срез-

ки. Поскольку срезки почти всегда скрываются за ссылками, мы употребляем сокращенное название более общего понятия.

Строковые типы

Программисты, знакомые с C++, знают, что в этом языке есть два строковых типа. Строковые литералы имеют указательный тип `const char *`. Но в стандартной библиотеке имеется также класс `std::string` для динамического создания строк во время выполнения.

Rust в этом отношении устроен похоже. В этом разделе мы покажем все способы записи строковых литералов, а затем познакомимся с двумя строковыми типами. Подробнее строки и работа с текстом рассмотрены в главе 17.

Строковые литералы

Строковые литералы заключены в двойные кавычки. В них используются те же управляющие последовательности, начинающиеся знаком `\`, что и в символьных литералах:

```
let speech = "\"Ouch!\" said the well.\n";
```

В строковых литералах, в отличие от символьных, одиночные кавычки не нуждаются в экранировании, а двойные нуждаются.

Строка может занимать несколько строчек в тексте программы:

```
println!("In the room the women come and go,
    Singing of Mount Abora");
```

Символ новой строки в этом строковом литерале включен в строку, поэтому будет напечатан. То же самое относится к пробелам в начале второй строчки.

Если какая-то строчка заканчивается символом `\`, то следующий за ним символ новой строки и пробелы в начале следующей строчки опускаются:

```
println!("It was a bright, cold day in April, and \
    there were four of us—\
    more or less.");
```

В данном случае будет напечатана одна строчка текста. Строка содержит только один пробел между словами «and» и «there» – тот, который предшествует обратной косой черте.

Иногда необходимость удваивать все знаки `\` в строке вызывает неудобства (классический пример – регулярные выражения и пути в Windows). Для таких случаев в Rust предусмотрены *простые строки* (raw string). Простая строка начинается буквой `r`. Все знаки обратной косой черты и пробельные символы включаются в простую строку без какой-либо обработки. Никакие управляющие последовательности не распознаются.

```
let default_win_install_path = r"C:\Program Files\Gorillas";
let pattern = Regex::new(r"\d+(\.\d+)*");
```

Включить двойную кавычку в простую строку, просто поставив перед ней знак `\`, невозможно – напомним, не распознаются *никакие* управляющие последовательности. Но и на этот случай есть лекарство – начало и конец простой строки можно обозначить знаками решетки:

```
println!(r###"
    Эта простая строка начинается последовательностью 'r###"'.
    Поэтому она заканчивается, только когда встретится кавычка ('"'),
    за которой следуют три знака решетки ('###'):
"###);
```

Количество знаков решетки может быть произвольным – столько, сколько понадобится, чтобы однозначно определить конец строки.

Байтовые строки

Строковый литерал с префиксом `b` называется байтовой строкой. Она представляет собой срезку значений типа `u8` (т. е. байтов), а не Юникод-текст.

```
let method = b"GET";
assert_eq!(method, &[b'G', b'E', b'T']);
```

Байтовые строки согласуются с описанными ранее вариантами синтаксиса строк: они могут занимать несколько строчек, содержать управляющие последовательности и соединяться с помощью обратной косой черты. Простые байтовые строки начинаются символами `br"`.

Байтовые строки не могут содержать произвольных символов Юникода, а должны обходиться символами кода ASCII и управляющими последовательностями вида `\xHH`.

Переменная `method` выше имеет вид `&[u8; 3]`: это ссылка на массив из 3 байтов. Она не располагает методами строки, которые мы обсудим ниже. Со строками ее роднит разве что синтаксис записи.

Строки в памяти

В Rust строки являются последовательностями символов Юникода, но в памяти они хранятся не как массивы символов типа `char`, а в кодировке переменной ширины UTF-8. Каждый символ ASCII в строке занимает один байт, а все прочие символы – несколько байтов.

На рис. 3.2 показаны значения типа `String` и `&str`, созданные следующим кодом:

```
let noodles = "noodles".to_string();
let oodles = &noodles[1..];
let poodles = "ಠ_ಠ";
```

С типом `String` связан буфер переменного размера, в котором хранится текст в кодировке UTF-8. Буфер выделяется из кучи, поэтому его размер может увеличиваться или уменьшаться по мере необходимости. В примере выше переменная `noodles` имеет тип `String` и владеет буфером длиной 8 байтов, из которых используются семь. Можно считать, что тип `String` – это вектор `Vec<u8>`, который гарантированно содержит корректно сформированную строку UTF-8; именно так в действительности и реализован тип `String`.

Тип `&str` (произносится «стир» или «срезка строки») – это ссылка на отрезок текста в кодировке UTF-8, которым владеет кто-то еще: этот текст «заимствуется». В примере выше переменная `oodles` – это ссылка `&str` на последние 6 байтов буфера, принадлежащего переменной `noodles`, т. е. она представляет текст «oodles». Как и любая другая ссылка на срезку, `&str` является толстым указателем, содержащим

адрес данных и их длину. Можно считать, что `&str` – это просто тип `&[u8]`, который гарантированно содержит корректно сформированный текст в кодировке UTF-8.

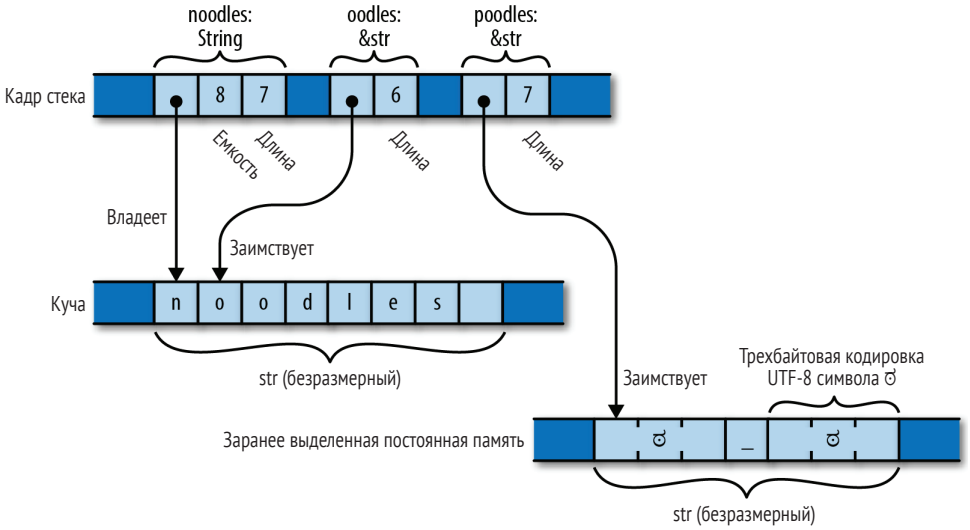


Рис. 3.2 ❖ Значения типа `String`, `&str` и `str`

Строковый литерал – это ссылка `&str` на текст в предварительно выделенной постоянной памяти, расположенной в той же области, что и машинный код программы. В примере выше `poodles` – строковый литерал, указывающий на область длиной семь байтов, созданную в начале выполнения программы и существующую до момента ее завершения.

Метод `.len()` типа `String` или `&str` возвращает длину строки, измеряемую в байтах, а не в символах:

```
assert_eq!("ơ_ơ".len(), 7);
assert_eq!("ơ_ơ".chars().count(), 3);
```

Модифицировать `&str` невозможно:

```
let mut s = "hello";
s[0] = 'c';    // ошибка: к типу `str` невозможн изменяющий доступ по индексу
s.push('\n');  // ошибка: в типе `&str` нет метода `push`
```

Для создания новых строк во время выполнения пользуйтесь типом `String`.

Тип `&mut str` существует, но не особенно полезен, потому что почти все операции с текстом в кодировке UTF-8 могут изменять длину в байтах, а срезка не способна перераспределить память для объекта, на который ссылается. Фактически для типа `&mut str` доступны только операции `make_ascii_uppercase` и `make_ascii_lowercase`, которые по определению модифицируют текст на месте и затрагивают только однобайтовые символы.

Тип String

Тип `&str` очень похож на `&[T]` – толстый указатель на данные. А тип `String` аналогичен `Vec<T>`.

	Vec<T>	String
Автоматически освобождает буфер	Да	Да
Может расти	Да	Да
Имеет статические методы `::new()` и `::with_capacity()`	Да	Да
Имеет методы `.reserve()` и `.capacity()`	Да	Да
Имеет методы `.push()` и `.pop()`	Да	Да
Синтаксис диапазона `v[start..stop]`	Да, возвращает `&[T]`	Да, возвращает `&str`
Автоматическое преобразование	`&Vec<T>` в `&[T]`	`&String` в `&str`
Наследует методы	От `&[T]`	От `&str`

Как и `Vec`, каждая строка типа `String` обладает собственным выделенным из кучи буфером, который не разделяется с другими строками. Когда переменная типа `String` покидает область видимости, ее буфер автоматически освобождается, если только переменная не была передана.

Создать строку можно несколькими способами.

- Метод `.to_string()` преобразует `&str` в `String`. Строка при этом копируется.

```
let error_message = "too many pets".to_string();
```

- Макрос `format!()` работает как `println!()`, но вместо вывода в `stdout` возвращает новую строку, при этом в конец не добавляется знак новой строки.

```
assert_eq!(format!("{}°{:02}′{:02}″N", 24, 5, 23),
           "24°05′23″N".to_string());
```

- У массивов, срезок и векторов есть два метода, `.concat()` и `.join(sep)`, которые образуют новый объект `String` из нескольких строк:

```
let bits = vec!["veni", "vidi", "vici"];
assert_eq!(bits.concat(), "venividivici");
assert_eq!(bits.join(", "), "veni, vidi, vici");
```

Иногда возникает вопрос, какой тип лучше использовать: `&str` или `String`. В главе 4 этот вопрос рассматривается подробно, а пока просто скажем, что `&str` может ссылаться на любую срезку любой строки, будь то строковый литерал (хранящийся вместе с исполняемым кодом) или объект `String` (динамически выделяемый и освобождаемый во время выполнения). Следовательно, `&str` больше подходит для аргументов функций, когда требуется, чтобы вызывающая сторона могла передавать строку любого вида.

Использование строк

Строки поддерживают операторы `==` и `!=`. Две строки равны, если они содержат одни и те же символы в одном и том же порядке (при этом они могут указывать на разные области памяти).

```
assert!("ONE".to_lowercase() == "one");
```

Строки также поддерживают операторы `<`, `<=`, `>` и `>=` и многочисленные полезные методы и функции, прочитать о которых можно в документации (раздел «`str` (primitive type)» и модуль `std::str`). А можете вместо этого обратиться к главе 17.

```
assert!("peanut".contains("nut"));
assert_eq!("☉_☉".replace("☉", "■"), "■_■");
```

```
assert_eq!("    clean\n".trim(), "clean");

for word in "veni, vidi, vici".split(", ") {
    assert!(word.starts_with("v"));
}
```

Имейте в виду, что в силу самой природы Юникода простое посимвольное сравнение не всегда дает ожидаемый результат. Например, в Rust обе строки `"th\u{e9}"` и `"the\u{301}"` являются допустимыми Юникод-представлениями французского слова *thé* (чай). Согласно стандарту Юникода, они должны отображаться и обрабатываться одинаково, но в Rust считаются разными строками. Аналогично в операторах сравнения, например `<`, используется простой лексикографический порядок, основанный на значении кодовой позиции символа. Это упорядочение лишь иногда совпадает с упорядочением, применяемым в языке пользователя в соответствии с культурными традициями. Более подробно эти вопросы обсуждаются в главе 17.

Другие типы, похожие на строки

Rust гарантирует, что строки всегда имеют корректное представление в UTF-8. Но иногда программа вынуждена иметь дело со строками, которые *некорректны* с точки зрения правил Юникода. Такое обычно происходит, когда Rust-программа взаимодействует с другой системой, не поддерживающей эти правила. Например, в большинстве операционных систем легко создать файл с именем, не являющимся корректной строкой Юникода. И что делать Rust-программе, столкнувшейся с таким именем файла?

Для подобных случаев Rust предлагает несколько типов, похожих на строки. При работе с Юникод-текстом применяйте только типы `String` и `&str`, но:

- при работе с именами файлов пользуйтесь типами `std::path::PathBuf` и `&Path`;
- при работе с двоичными данными пользуйтесь типами `Vec<u8>` и `&[u8]`;
- при работе с именами переменных среды и аргументами командной строки в формате операционной системы пользуйтесь типами `OsStr` и `OsString`;
- при взаимодействии с библиотеками, написанными на C, в которых строки заканчиваются нулем, пользуйтесь типами `std::ffi::CString` и `&CStr`.

Более сложные типы

Типы – сердцевина Rust. Мы будем говорить о них и знакомиться с новыми типами на протяжении всей книги.

В частности, большая часть возможностей Rust проистекает из пользовательских типов, потому что именно в них определены методы. Существуют три разновидности пользовательских типов, которые будут рассмотрены в последующих главах: структуры (глава 9), перечисления (глава 10) и характеристики (глава 11).

У функций и замыканий собственные типы, они рассматриваются в главе 14. А типы, определенные в стандартной библиотеке, будут вводиться по ходу изложения. Так, в главе 16 представлены типы стандартных коллекций.

Но со всем этим придется подождать. Прежде рассмотрим концепции, лежащие в основе правил безопасности в Rust.

Глава 4

Владение

> Я обнаружил, что Rust заставляет меня переосмыслить многое из того, что я привык считать «обычной практикой» в C/C++, чтобы программа хотя бы откомпилировалась… Хочу подчеркнуть, что Rust не из тех языков, которые можно выучить за пару дней, отложив трудные технические вопросы на потом. Вам придется сразу же освоить строгую безопасность, и поначалу вы будете чувствовать себя некомфортно. Но зато лично я теперь снова чувствую, что компиляция кода – это не пустое дело.
>
> — *Митчелл Нордайн*

Rust обещает две вещи, абсолютно необходимые в безопасном языке системного программирования:

- программист определяет время жизни каждого значения в программе. Rust быстро освобождает память и другие ресурсы, принадлежащие значению, в точке, контролируемой программистом;
- программа никогда не сможет воспользоваться указателем на освобожденный объект. Обращение по «висячему указателю» – типичная ошибка в C и C++; если вам повезет, то программа просто аварийно завершится, а если не повезет, то получите брешь в системе безопасности. Rust обнаруживает такие ошибки на этапе компиляции.

C и C++ выполняют первое обещание: мы можем вызвать `free` или `delete` для любого объекта, динамически выделенного из кучи, в любой момент времени. Но второе обещание остается невыполненным: программист должен сам следить за тем, чтобы указатель на освобожденное значение не использовался. Обширный накопленный опыт свидетельствует о том, что этому требованию трудно удовлетворить, – неправильная работа с указателями сплошь и рядом оказывается причиной ошибок, зарегистрированных в открытых базах данных, посвященных проблемам безопасности.

Многие языки выполняют второе обещание с помощью сборки мусора, в ходе которой автоматически освобождаются объекты, на которые не осталось активных указателей. Но в обмен мы теряем контроль над тем, когда именно сборщик мусора освобождает объект. В общем случае поведение сборщиков мусора слабо предсказуемо, бывает нелегко понять, почему память не освободилась, когда вы этого ждали. А когда работаешь с объектами, представляющими файлы, сетевые соединения и другие ресурсы операционной системы, очень важно, чтобы эти объекты, а вместе с ними и сопутствующие ресурсы, освобождались вовремя.

Ни один из этих компромиссов для Rust не годится: программист должен контролировать время жизни значений и язык должен быть безопасным. Но эта область проектирования языков программирования неплохо изучена. Не внеся фундаментальных изменений, в ней невозможно добиться существенного улучшения.

Rust разрубает этот узел неожиданным способом: введя ограничения на использование указателей. Эта и следующая глава как раз и посвящены объяснению данных ограничений. Пока достаточно знать, что некоторые привычные конструкции не отвечают правилам и для них придется подыскать альтернативы. Но в целом ограничения привносят достаточно порядка в хаос, чтобы компилятор Rust мог проверить отсутствие ошибок, связанных с небезопасной работой с памятью: висячих указателей, двойного освобождения, использования неинициализированной памяти и т. д. На этапе выполнения указатели – это просто адреса в памяти, как в C и C++. Разница в том, что программа работает с ними безопасно, и это можно доказать.

Те же правила лежат в основе поддержки безопасного конкурентного программирования в Rust. Благодаря тщательно спроектированным потоковым примитивам правила, гарантирующие корректность работы с памятью, позволяют утверждать, что программа не содержит гонок за данные. Никакая ошибка в Rust-программе не может привести к тому, что один поток повредит данные другого и тем самым приведет к трудно воспроизводимым ошибкам в других частях программы. Недетерминированное поведение, присущее многопоточному коду, инкапсулировано в тех конструкциях, которые предназначены для работы с ним, – мьютексах, каналах сообщений, атомарных значениях и т. д., а не в обычных ссылках на области памяти. Многопоточный код в C и C++ заслуженно пользуется дурной репутацией, но Rust реабилитировал его.

Бесшабашная ставка Rust – заявление, которое должно стать залогом его успеха и которое формирует внутреннюю основу языка, – заключается в том, что, несмотря на все ограничения, язык обладает более чем достаточной гибкостью для решения почти любой задачи и что преимущества – устранение широкого класса ошибок, связанных с управлением памятью и конкурентностью, – оправдывают необходимость изменить стиль программирования. Авторы этой книги с оптимизмом смотрят в будущее Rust именно потому, что обладают солидным опытом работы с C и C++. Для нас выгоды перехода на Rust совершенно очевидны.

Правила Rust, скорее всего, покажутся вам непохожими на все, что вы видели в других языках программирования. На наш взгляд, научиться работать с ними и применять их себе во благо – самое трудное в изучении Rust. В этой главе мы сначала обоснуем правила Rust, показав, как те же проблемы проявляются в других языках. Затем мы объясним эти правила во всех деталях. И наконец, поговорим об исключениях и почти исключениях.

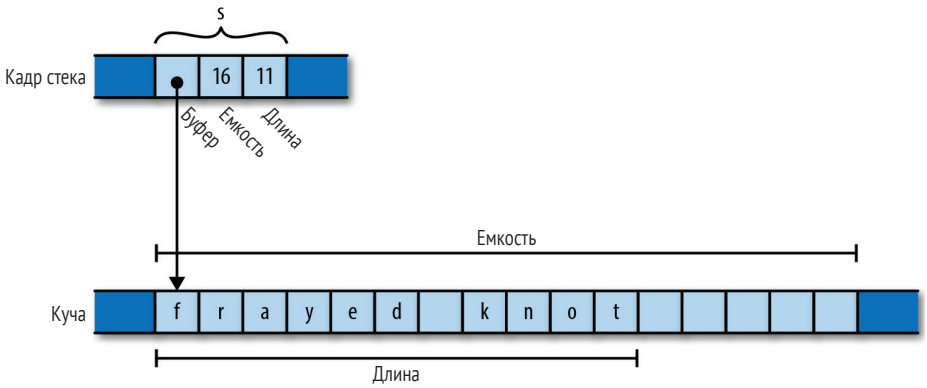
Владение

Если вы читали много кода, написанного на C или C++, то наверняка встречали комментарии, в которых говорилось, что экземпляр некоторого класса «владеет» объектом, на который указывает. Обычно это означает, что объект-владелец решает, когда освободить принадлежащий ему объект: когда владелец умирает, он забирает в могилу все ему принадлежащее.

Пусть, например, мы пишем такой код на C++:

```
std::string s = "frayed knot";
```

Строка `s` обычно представлена в памяти следующим образом:



Здесь сам объект `std::string` всегда занимает три слова: указатель на выделенный из кучи буфер, емкость буфера (сколько текста в него поместится, прежде чем нужно будет перераспределять память) и длина текста, хранящегося в буфере сейчас. Эти три поля являются закрытыми членами класса `std::string` и недоступны его пользователям.

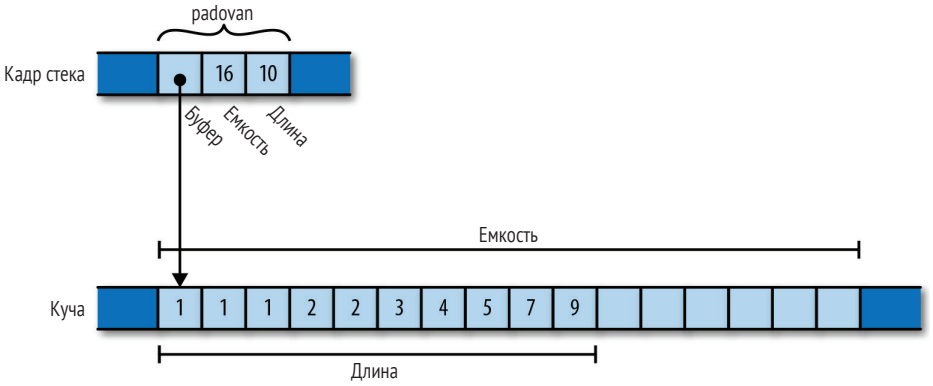
Класс `std::string` владеет своим буфером: когда программа уничтожает строку, ее деструктор освобождает буфер. В прошлом в некоторых библиотеках C++ один буфер мог разделяться между несколькими объектами `std::string`, а для того чтобы решить, когда освобождать буфер, использовался счетчик ссылок. В последних вариантах спецификации C++ такое явно запрещено; во всех современных библиотеках применяется описанный выше подход. В подобных ситуациях принято считать, что хотя другому коду разрешено создавать временные указатели на память, принадлежащую данному объекту, этот код несет ответственность за то, чтобы все такие указатели были уничтожены к моменту, когда владелец решит уничтожить принадлежащий ему объект. Вы можете создать указатель на символ внутри буфера `std::string`, но когда строка будет уничтожена, ваш указатель станет недействительным, и именно вы отвечаете за то, чтобы он больше не использовался. Владелец определяет время жизни принадлежащих ему объектов, а все остальные должны уважать его решения.

Rust выносит этот принцип из комментариев и явно включает его в язык. В Rust у каждого значения имеется единственный владелец, который определяет время его жизни. Когда владелец освобождается – в терминологии Rust «уничтожается» (dropped), – принадлежащее ему значение уничтожается тоже. Смысл этих правил – в том, чтобы время жизни любого значения можно было определить, просто читая код, и предоставить программисту контроль над временем жизни, поддерживаемый языком.

Переменная владеет собственным значением. Когда управление покидает блок, в котором переменная объявлена, эта переменная уничтожается, и вместе с ней уничтожается ее значение, например:

```
fn print_padovan() {
    let mut padovan = vec![1,1,1];  // здесь выделена
    for i in 3..10 {
        let next = padovan[i-3] + padovan[i-2];
        padovan.push(next);
    }
    println!("P(1..10) = {:?}", padovan);
}                                   // а здесь уничтожена
```

Переменная `padovan` имеет тип `std::vec::Vec<i32>`, т. е. является вектором 32-разрядных целых. В памяти конечное значение `padovan` выглядит примерно так:



Это очень похоже на описанный выше объект класса `std::string` в C++, только элементами буфера являются 32-разрядные целые, а не символы. Отметим, что слова, в которых хранятся указатель, емкость и длина буфера, связанные с переменной `padovan`, находятся в кадре стека функции `print_padovan`, и лишь сам буфер вектора выделен из кучи.

Как строка `s` ранее, вектор владеет буфером, в котором хранятся его элементы. Когда переменная `padovan` покидает область видимости в конце функции, программа уничтожает вектор. А поскольку вектор владеет своим буфером, буфер уничтожается вместе с ним.

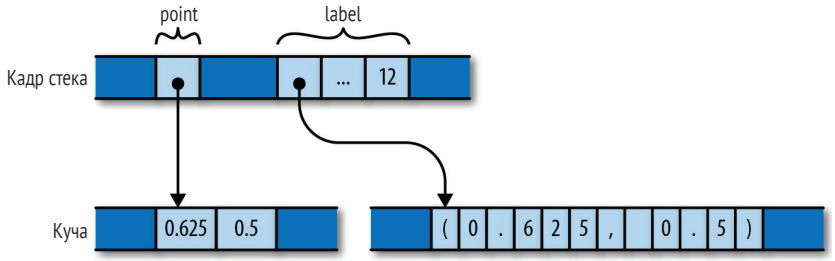
Тип `Box` в Rust служит еще одним примером владения. `Box<T>` – это указатель на значение типа `T`, размещенное в куче. При вызове `Box::new(v)` выделяется место в куче, значение `v` перемещается в него и возвращается объект `Box`, указывающий на выделенную область. Поскольку `Box` владеет областью, на которую указывает, при уничтожении `Box` эта область освобождается.

Например, для размещения кортежа в куче можно поступить следующим образом:

```
{
    let point = Box::new((0.625, 0.5));  // здесь выделена память для point
    let label = format!("{:?}", point);  // здесь выделена память для label
    assert_eq!(label, "(0.625, 0.5)");
}                                        // здесь обе переменные уничтожены
```

Когда программа вызывает `Box::new`, в куче выделяется место для кортежа из двух значений типа `f64`, аргумент `(0.625, 0.5)` перемещается в эту область и воз-

вращается указатель на нее. К моменту, когда управление доходит до макроса `assert_eq!`, кадр стека выглядит так:

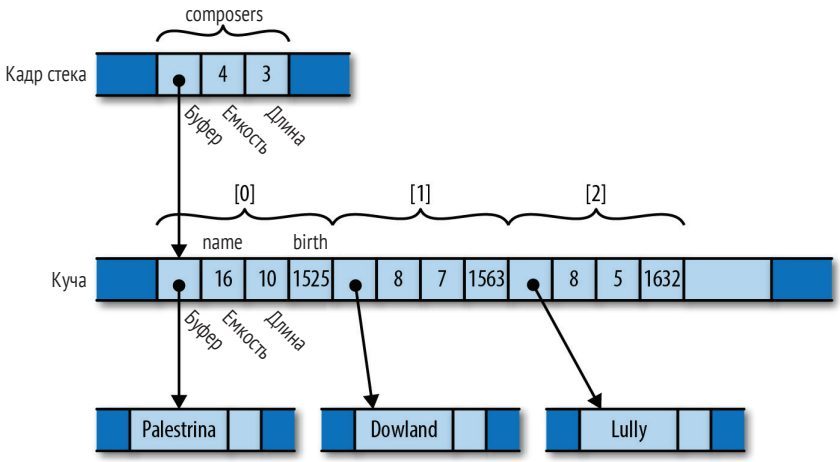


В самом кадре стека хранятся переменные `point` и `label`, каждая из которых ссылается на принадлежащую ей область кучи. Когда эти переменные уничтожаются, принадлежащие им области кучи также освобождаются.

Как переменная владеет своим значением, так структура владеет своими членами, а кортежи, массивы и векторы владеют своими элементами.

```
struct Person { name: String, birth: i32 }

let mut composers = Vec::new();
composers.push(Person { name: "Palestrina".to_string(),
                        birth: 1525 });
composers.push(Person { name: "Dowland".to_string(),
                        birth: 1563 });
composers.push(Person { name: "Lully".to_string(),
                        birth: 1632 });
for composer in &composers {
    println!("{}, born {}", composer.name, composer.birth);
}
```

Здесь переменная `composers` имеет тип `Vec<Person>` – вектор структур, в каждой из которых хранятся строка и число. В памяти конечное значение `composers` выглядит так:



В этом примере много отношений владения, но все они довольно просты: `composers` владеет вектором, вектор владеет своими элементами – структурами типа `Person`, каждая структура владеет своими полями, а строковое поле владеет своим текстом. Когда управление покидает область видимости, в которой объявлена переменная `composers`, программа уничтожает ее значение, а вместе с ним уходит в небытие и вся эта иерархия. Если бы вместо вектора использовались другие коллекции – `HashMap` или, скажем, `BTreeSet`, – картина была бы точно такой же.

Теперь отступим на шаг назад и рассмотрим описанные выше отношения владения более пристально. У каждого значения есть ровно один владелец, так что решить, когда это значение должно быть уничтожено, просто. Но одно значение может владеть многими другими, например вектор `composers` владеет всеми своими элементами. А эти значения могут, в свою очередь, владеть какими-то значениями: каждый элемент `composers` владеет строкой, которая владеет текстом.

Отсюда следует, что владельцы и принадлежащие им значения образуют *деревья*: владелец данного значения является его родителем, а значения, принадлежащие данному значению, – его детьми. А корнем каждого дерева является переменная; когда эта переменная выходит из области видимости, вместе с ней уходит все дерево. Такое дерево владения можно видеть на диаграмме переменной `composers`: это не «дерево» в смысле структуры дерева поиска или HTML-документа, составленного из элементов DOM. Мы имеем тут дерево, построенное из различных типов, причем правило единственного владельца запрещает перекрестные связи, которые могли бы превратить структуру данных в нечто более сложное, чем дерево. Каждое значение в Rust-программе является членом некоторого дерева с корнем в какой-то переменной.

В Rust-программах значения обычно не уничтожаются явно, как это делается в программах на C и C++ с помощью `free` и `delete`. Чтобы уничтожить значение в Rust, нужно каким-то образом исключить его из дерева владения: вывести переменную из области действия, удалить элемент вектора или еще что-то в этом роде. В этот момент Rust гарантирует корректное уничтожение значения вместе со всем ему принадлежащим.

В каком-то смысле Rust обладает меньшей мощью, чем другие языки: любой другой практический язык позволяет строить произвольные графы объектов, указывающих друг на друга, как мы того пожелаем. Но именно благодаря меньшей мощи языка Rust анализ программы, выполняемый языком, оказывается более содержательным. Гарантии безопасности, которые дает Rust, возможны только потому, что связи между объектами программы проще проследить. Это часть той самой «бесшабашной ставки», которую мы уже упоминали: на практике Rust утверждает, что его гибкости обычно более чем достаточно, чтобы найти хотя бы несколько отличных решений, не выходящих за рамки языковых ограничений.

При всем при том сказанное выше все же слишком ограничительно, чтобы быть полезным на практике. Rust добавляет к картине еще несколько штрихов:

- значения можно передавать от одного владельца другому. Это позволяет строить, реорганизовывать и уничтожать дерево;
- в стандартной библиотеке имеются указательные типы `Rc` и `Arc` с подсчетом ссылок, благодаря которым у значения может быть несколько владельцев при соблюдении некоторых ограничений;
- разрешается «заимствовать ссылку» на значение; ссылки – это невладеющие указатели с ограниченным временем жизни.

Все это увеличивает гибкость модели владения, не нарушая обещаний Rust. Мы рассмотрим их по очереди, отложив разговор о ссылках до следующей главы.

Передача владения

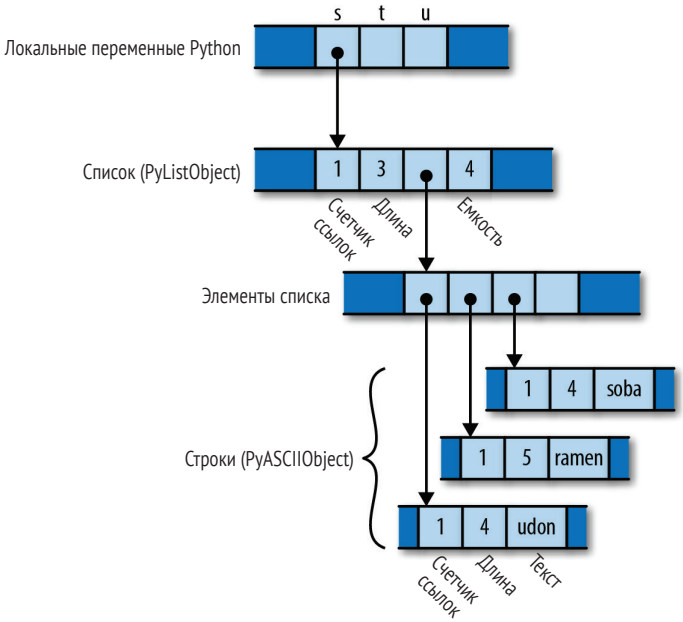
Для большинства типов в Rust операции типа присваивания значения переменной, передачи функции или возврата из функции не приводят к копированию значения, вместо этого значение *передается* (move). Исходный владелец передает право владения конечному, а сам становится неинициализированным; начиная с этого момента время жизни значения контролируется новым владельцем. В Rust-программе сложные структуры строятся и разбираются по одному значению за раз, по одной передаче за раз.

Возможно, вы удивлены тем, что Rust изменяет семантику таких фундаментальных операций, ведь нет сомнений, что смысл присваивания на данном историческом этапе прочно зафиксирован. Но если внимательно присмотреться к тому, как в разных языках обрабатывается присваивание, то мы заметим значительные вариации. Подобное сравнение заодно прояснит смысл и последствия выбора, сделанного в Rust.

Рассмотрим такой код на Python:

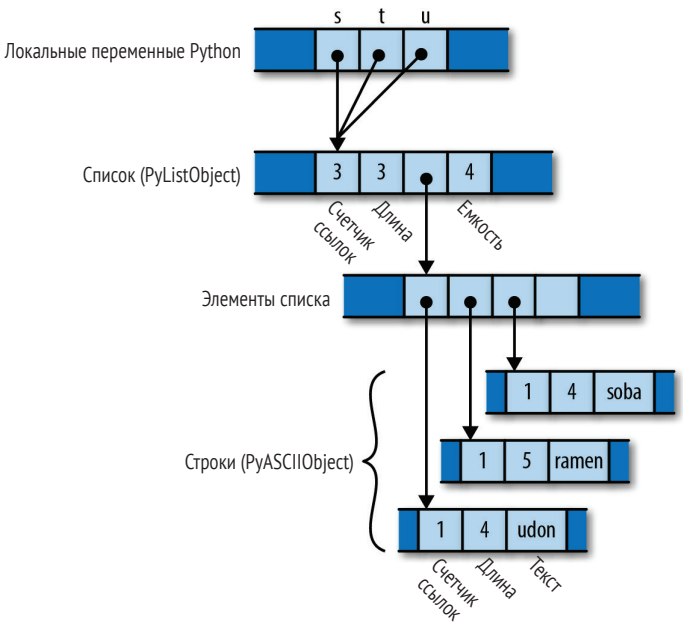
```
s = ['udon', 'ramen', 'soba']
t = s
u = s
```

С каждым объектом в Python связан счетчик ссылок, который равен количеству значений, указывающих на этот объект. Поэтому после присваивания `s` состояние программы выглядит так (некоторые поля опущены):



Поскольку на список указывает только `s`, счетчик ссылок списка равен 1, а поскольку список – единственный объект, указывающий на строки, то счетчик ссылок в каждой строке тоже равен 1.

Что происходит, когда программа выполняет присваивания переменным `t` и `u`? В Python присваивание реализовано просто: конечная переменная начинает указывать на тот же объект, что исходная, а счетчик ссылок в объекте увеличивается на 1. Поэтому конечное состояние программы выглядит так:

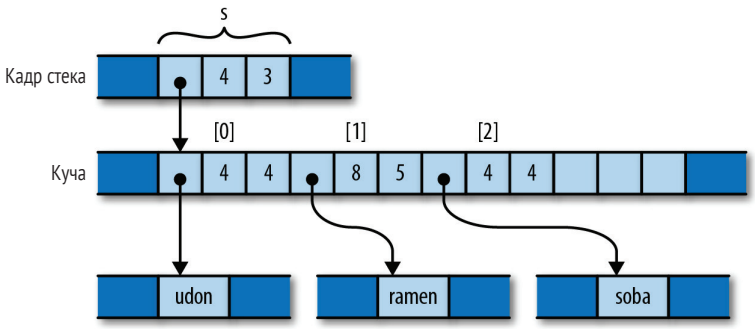


Python скопировал указатель из `s` в `t` и `u` и сделал счетчик ссылок списка равным 3. Присваивание в Python обходится дешево, но поскольку создается новая ссылка на объект, мы должны хранить счетчики ссылок, чтобы знать, когда можно будет освободить значение.

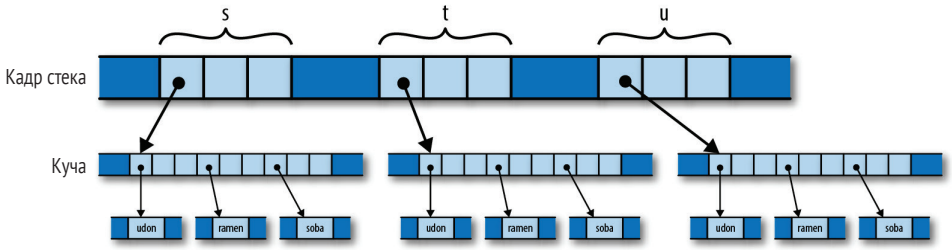
Теперь рассмотрим аналогичный код на C++:

```
using namespace std;
vector<string> s = { "udon", "ramen", "soba" };
vector<string> t = s;
vector<string> u = s;
```

Первоначальное значение `s` размещено в памяти так:



Что происходит, когда программа присваивает значение `s` переменным `t` и `u`? В C++ присваивание `std::vector` приводит к созданию копии вектора, и так же ведет себя `std::string`. Следовательно, к моменту выхода из этого кода программа создаст три вектора и девять строк:



В зависимости от конкретных значений присваивание в C++ может потреблять неограниченный объем памяти и процессорного времени. Но преимущество в том, что программе легко решить, когда освободить всю память: как только переменные выходят из области видимости, вся выделенная в этом фрагменте память автоматически освобождается.

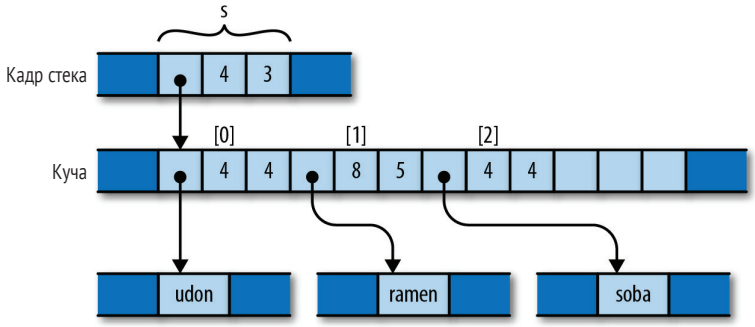
В некотором смысле C++ и Python выбрали противоположные решения: в Python присваивание дешево, но ценой подсчета ссылок (и в общем случае сборки мусора). В C++ очевидно, кто владеет памятью, но расплачиваться за это приходится глубоким копированием объектов в момент присваивания. Программисты на C++ зачастую вовсе не рады этому выбору: глубокое копирование может обходиться дорого, и обычно есть более практичные альтернативы.

Так как же выглядит аналогичная программа на Rust? А вот как:

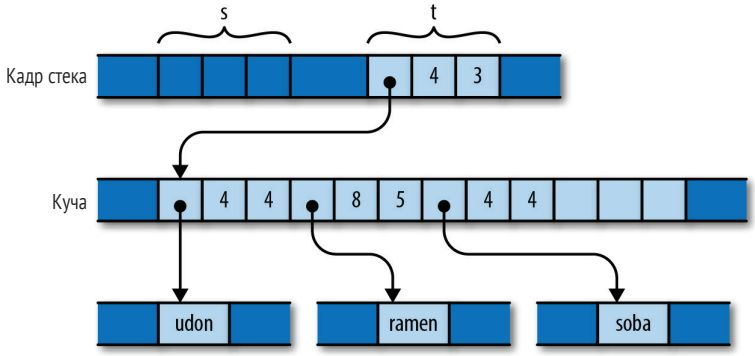
```
let s = vec!["udon".to_string(), "ramen".to_string(), "soba".to_string()];
let t = s;
let u = s;
```

Подобно C и C++, Rust размещает простые строковые литералы вида `"udon"` в постоянной памяти, поэтому для чистоты сравнения с примерами на C++ и Python мы вызвали метод `to_string`, чтобы получить значения типа `String` в куче.

Поскольку в Rust и в C++ представления векторов и строк похожи, то после инициализации `s` ситуация выглядит так же, как в C++:



Напомним, однако, что в Rust для большинства типов присваивание *передает* значение от исходной переменной к конечной, оставляя исходную неинициализированной. Поэтому после инициализации `t` память программы выглядит так:



Что здесь произошло? В ходе инициализации `let t = s;` все три поля заголовка вектора были перемещены из `s` в `t`; теперь вектором владеет `t`. Элементы вектора остались там же, где и были, и со строками тоже ничего не случилось. У каждого значения по-прежнему единственный владелец, хотя и другой. Не пришлось изменять никаких счетчиков ссылок. А компилятор теперь считает, что `s` не инициализирована.

А что произойдет, когда мы дойдем до предложения `let u = s;`? Здесь производится попытка присвоить `u` неинициализированное значение `s`. Rust предусмотрительно запрещает использовать неинициализированные значения, поэтому компилятор на такой код ругается:

```
error[E0382]: use of moved value: `s`
--> ownership_double_move.rs:9:9
  |
8 | let t = s;
  |     - value moved here
9 | let u = s;
  |     ^ value used here after move
  |
```

Рассмотрим последствия передачи в этом примере. Как и в Python, присваивание стоит дешево: программа просто перемещает три слова заголовка вектора из одного места в другое. Но, как и в C++, всегда ясно, кто владелец: программе не нужно подсчитывать ссылки или заниматься сборкой мусора, чтобы знать, когда освобождать память, занятую элементами вектора и содержимым строк.

Цена за это – явный запрос копирования в тех случаях, когда это нужно. Если вы хотите, чтобы в конце работы программа оказалась в таком же состоянии, как в случае C++, когда каждая переменная хранит независимую копию структуры, то должны будете вызвать метод вектора `clone`, который производит глубокое копирование вектора и его элементов:

```
let s = vec!["udon".to_string(), "ramen".to_string(), "soba".to_string()];
let t = s.clone();
let u = s.clone();
```

Можно воспроизвести также поведение Python, если воспользоваться указательными типами Rust с подсчетом ссылок; мы обсудим этот вопрос чуть ниже.

Другие операции с передачей

До сих пор мы приводили примеры инициализации, когда переменная получает значение в предложении `let` одновременно с объявлением. Присваивание переменной несколько отличается, поскольку, когда значение передается уже инициализированной переменной, старое значение этой переменной уничтожается. Например:

```
let mut s = "Govinda".to_string();
s = "Siddhartha".to_string(); // здесь значение "Govinda" уничтожено
```

В этом случае, когда мы присваиваем переменной `s` строку `"Siddhartha"`, сначала уничтожается ее прежнее значение `"Govinda"`. Однако рассмотрим такой фрагмент:

```
let mut s = "Govinda".to_string();
let t = s;
s = "Siddhartha".to_string(); // здесь ничего не уничтожается
```

На этот раз `t` переняла владение исходной строкой от `s`, так что к моменту присваивания `s` та уже не инициализирована. В такой ситуации никакая строка не уничтожается.

Мы рассматривали инициализацию и присваивание просто потому, что это простые операции. Но Rust применяет семантику передачи почти во всех случаях, где используются значения. При передаче аргументов функции передается владение ее параметрами, при возврате значения из функции владение переходит вызывающей стороне. При построении кортежа владение значениями переходит к кортежу. И так далее.

Вероятно, вы теперь лучше понимаете, что происходит в примерах из предыдущего раздела. Так, при построении вектора композиторов мы писали:

```
struct Person { name: String, birth: i32 }

let mut composers = Vec::new();
composers.push(Person { name: "Palestrina".to_string(),
                        birth: 1525 });
```

В этом коде есть несколько мест, в которых происходит передача владения, помимо инициализации и присваивания:

- *возврат значения из функции.* Вызов `Vec::new()` конструирует новый вектор и возвращает не указатель на вектор, а сам вектор: владение им передается от `Vec::new` переменной `composers`. Аналогично вызов `to_string` возвращает новый экземпляр `String`;
- *конструирование новых значений.* Поле `name` новой структуры `Person` инициализировано значением, возвращенным методом `to_string`. Структура принимает владение строкой;
- *передача значения функции.* Методу вектора `push` передается вся структура `Person`, а не указатель на нее, в результате чего структура помещается в конец вектора. Вектор принимает владение структурой `Person` и, как следствие, становится косвенным владельцем строки в поле `name`.

Передача значений может показаться неэффективной, но следует помнить две вещи. Во-первых, передача всегда относится к самому значению, а не к области кучи, в которой оно хранится. В случае векторов и строк «само значение» – это всего лишь три слова заголовка, тогда как потенциально большие массивы элементов и текстовые буферы остаются там, где и были, – в куче. Во-вторых, кодогенератор Rust отлично умеет «проницать» все эти передачи, и на практике значение часто сохраняется именно в том месте, где должно быть.

Передача владения и поток управления

Во всех предыдущих примерах поток управления был очень простым, но как передача владения взаимодействует с более сложным кодом? Общий принцип таков: если существует возможность, что значение переменной было передано, после чего она не получила нового значения, то переменная считается неинициализированной. Например, если у переменной все еще имеется значение после вычисления условия в выражении `if`, то ее можно использовать в обеих ветвях:

```
let x = vec![10, 20, 30];
if c {
    f(x); // ... здесь разрешено передать x
} else {
    g(x); // ... здесь тоже разрешено передать x
}
h(x) // ошибка: здесь x не инициализирована после выполнения на любом пути
```

По тем же причинам передача владения переменной в цикле запрещена:

```
let x = vec![10, 20, 30];
while f() {
    g(x); // ошибка: x передана на первой итерации,
          // а на второй уже не инициализирована
}
```

Если, конечно, мы не присваиваем `x` новое значение на каждой итерации:

```
let mut x = vec![10, 20, 30];
while f() {
    g(x);          // владение x передано
    x = h();       // x получает новое значение
}
e(x);
```

Передача владения и индексированное содержимое

Мы уже отмечали, что в результате передачи исходная сущность становится неинициализированной, а конечная принимает владение значением. Но не каждый владелец значения готов стать неинициализированным. Рассмотрим такой код:

```
// Построить вектор строк "101", "102", ... "105"
let mut v = Vec::new();
for i in 101 .. 106 {
    v.push(i.to_string());
}

// Выбрать случайные элементы вектора.
let third = v[2];
let fifth = v[4];
```

Чтобы этот код работал, Rust должен каким-то образом запомнить, что третий и пятый элементы вектора стали неинициализированными, и хранить эту информацию до момента уничтожения вектора. В общем случае векторам пришлось бы нести дополнительную информацию о том, какие элементы еще живы, а какие стали неинициализированными. Очевидно, что такое поведение языка системного программирования не назовешь правильным: вектор должен быть вектором – и больше ничем. На самом деле при компиляции показанного выше кода Rust выдает ошибку:

```
error[E0507]: cannot move out of indexed content
  --> ownership_move_out_of_vector.rs:14:17
   |
14 |     let third = v[2];
   |                 ^^^^
   |                 |
   |                 help: consider using a reference instead `&v[2]`
   |                 cannot move out of indexed content
```

К аналогичной ошибке приводит передача значения переменной `fifth`. В сообщении об ошибке Rust предлагает воспользоваться ссылкой, если вам нужен доступ к элементу без передачи владения им. Часто это именно то, что требуется. Но что, если действительно нужно переместить элемент из вектора? Тогда нужно отыскать способ сделать это, не нарушая ограничений типа. Вот три возможности:

```
// Построить вектор строк "101", "102", ... "105"
let mut v = Vec::new();
for i in 101 .. 106 {
    v.push(i.to_string());
}

// 1. Взять последнее значение:
let fifth = v.pop().unwrap();
assert_eq!(fifth, "105");

// 2. Взять значение из середины вектора и переместить на его место
// последний элемент:
let second = v.swap_remove(1);
assert_eq!(second, "102");
```

```
// 3. Подставить другое значение вместо изъятого:
let third = std::mem::replace(&mut v[2], "substitute".to_string());
assert_eq!(third, "103");

// Посмотрим, что осталось от вектора.
assert_eq!(v, vec!["101", "104", "substitute"]);
```

В каждом случае элемент перемещается из вектора, но так, что вектор остается заполненным, хотя его размер, возможно, уменьшается.

Типы коллекций наподобие `Vec` обычно содержат методы, позволяющие потребить все элементы в цикле:

```
let v = vec!["liberté".to_string(),
             "égalité".to_string(),
             "fraternité".to_string()];

for mut
    s in v { s.push('!');
    println!("{}", s);
}
```

Когда вектор обрабатывается в цикле вида `for ... in v`, владение вектором *передается* от `v`, так что переменная `v` оказывается неинициализированной. Внутренний механизм цикла `for` принимает владение вектором и разлагает его на элементы. На каждой итерации цикл передает очередной элемент переменной `s`. Поскольку теперь `s` владеет строкой, мы можем модифицировать ее в теле цикла перед печатью. А поскольку сам вектор больше не виден программе, никакой код не может наблюдать его в середине цикла в частично опустошенном состоянии.

Если вам все же понадобится передать значение от владельца, состояния которого компилятор не может отследить, подумайте о том, чтобы изменить тип владельца на такой, для которого можно динамически сказать, имеет он значение или нет. Вот вариация на тему предыдущего примера:

```
struct Person { name: Option<String>, birth: i32 }

let mut composers = Vec::new();
composers.push(Person { name: Some("Palestrina".to_string()),
                        birth: 1525 });
```

Следующее предложение недопустимо:

```
let first_name = composers[0].name;
```

Будет выдано то же сообщение «cannot move out of indexed content», что и раньше. Но поскольку мы изменили тип поля `name` с `String` на `Option<String>`, `None` теперь является допустимым значением этого поля, так что следующий фрагмент работает:

```
let first_name = std::mem::replace(&mut composers[0].name, None);
assert_eq!(first_name, Some("Palestrina".to_string()));
assert_eq!(composers[0].name, None);
```

Вызов `replace` перемещает значение из поля `composers[0].name`, оставляя вместо него `None` и передавая владение исходным значением вызывающей программе. На самом деле подобное использование `Option` настолько распространено, что в дан-

ном типе имеется метод `take` специально для этой цели. Показанный выше вызов `replace` можно заменить эквивалентным предложением:

```
let first_name = composers[0].name.take();
```

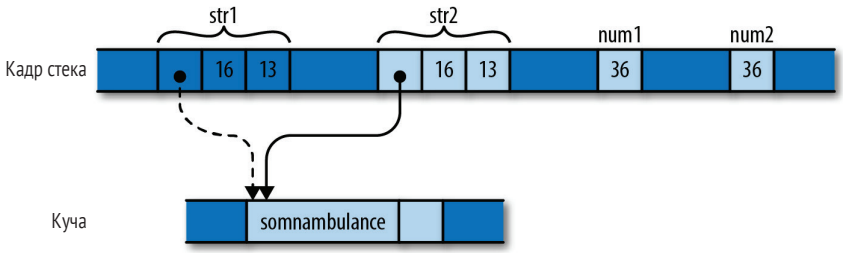
Копируемые типы: исключения из правила передачи владения

В рассмотренных выше примерах передачи значений фигурировали векторы, строки и другие типы, которые потенциально занимают много памяти, так что копирование обходится дорого. Благодаря передаче значения присваивание оказывается дешевым и понятно, кто является владельцем. Но для более простых типов, например целых чисел и символов, такая аккуратная обработка – излишество.

Сравним состояние памяти при присваивании значений типа `String` и `i32`:

```
let str1 = "somnambulance".to_string();
let str2 = str1;

let num1: i32 = 36;
let num2 = num1;
```

После выполнения этого кода память выглядит так:



Как и в случае векторов, присваивание *передает* владение от `str1` к `str2`, так что в итоге мы не получаем двух строк, отвечающих за освобождение одного и того же буфера. Однако с `num1` и `num2` ситуация иная. Число типа `i32` – это просто комбинация битов в памяти, оно не владеет никакими ресурсами в куче и ни от чего не зависит. После перемещения битов `num1` в `num2` мы просто получили совершенно независимую копию `num1`.

Передача значения оставляет исходную переменную неинициализированной. Но если в случае `str1` мы добились цели – лишить переменную значения, то в отношении `num1` эта цель не имеет смысла; никакого вреда от того, что мы продолжим использовать `num1`, нет. Передача в данном случае не дает никаких преимуществ, а только создает неудобства.

Выше мы аккуратно сказали, что передача владения осуществляется для *большинства* типов, теперь поговорим об исключениях, которые в Rust называются «копируемыми типами» (Copy type). Присваивание значения копируемому типу приводит к копированию, а не к передаче значения. Исходная переменная остается инициализированной, сохраняет прежнее значение и может использовать-

ся дальше. Точно так же копируемые типы ведут себя при передаче аргументов функциям и конструкторам.

К стандартным копируемым типам относятся все машинные числовые типы, целые и с плавающей точкой, типы `char` и `bool` и некоторые другие. Кортеж или массив фиксированного размера, содержащий копируемые типы, сам является копируемым типом.

Копируемыми могут быть только типы, для которых достаточно простого побитового копирования. Как мы уже объясняли, `String` не является копируемым типом, потому что владеет буфером, выделенным из кучи. По тем же причинам не является копируемым тип `Box<T>`, он владеет находящимся в куче объектом ссылки. Тип `File`, представляющий описатель файла операционной системы, некопируемый: дублирование такого значения приводит к запросу еще одного описателя у операционной системы. Аналогично не является копируемым тип `MutexGuard`, представляющий захваченный мьютекс: его копирование вообще лишено смысла, поскольку в каждый момент времени удерживать мьютекс может только один поток.

Эвристическое правило таково: если при уничтожении значения некоторого типа нужно произвести какие-то специальные действия, то этот тип не может быть копируемым. Объект `Vec` должен освободить свои элементы, объект `File` – закрыть описатель файла, объект `MutexGuard` – освободить мьютекс. При побитовом дублировании таких типов остается непонятным, какое из двух значений несет ответственность за ресурсы, которым владел исходный объект.

А что сказать о типах, определенных пользователем? По умолчанию типы `struct` и `enum` некопируемые:

```
struct Label { number: u32 }

fn print(l: Label) { println!("STAMP: {}", l.number); }

let l = Label { number: 3 };
print(l);
println!("Номер моей метки: {}", l.number);
```

Этот код не компилируется, Rust выдает сообщение:

```
error[E0382]: use of moved value: `l.number`
  --> ownership_struct.rs:12:40
   |
11 |     print(l);
   |           - value moved here
12 |     println!("Номер моей метки: {}", l.number);
   |                                      ^^^^^^^^ value used here after move
   |
   = note: move occurs because `l` has type `main::Label`, which does not
           implement the `Copy` trait
```

Поскольку тип `Label` некопируемый, после передачи его функции `print` владельцем значения стала эта функция, которая уничтожила значение перед возвратом. Но это же глупо, ведь `Label` – всего лишь тип `i32` с некоторыми синтаксическими украшениями. Совершенно незачем передавать владение при передаче `l` функции `print`.

Однако определенные пользователем типы являются некопируемыми только по умолчанию. Если все поля структуры имеют копируемый тип, то и саму структуру можно сделать копируемой, поместив перед ее определением атрибут `#[derive(Copy, Clone)]`:

```
#[derive(Copy, Clone)]
struct Label { number: u32 }
```

После этого изменения код нормально компилируется. Но если попытаться проделать такой трюк для структуры, не все поля которой имеют копируемый тип, то ничего не выйдет. При компиляции следующего кода

```
#[derive(Copy, Clone)]
struct StringLabel { name: String }
```

выдается ошибка:

```
error[E0204]: the trait `Copy` may not be implemented for this type
 --> ownership_string_label.rs:7:10
  |
7 | #[derive(Copy, Clone)]
  |          ^^^^
8 | struct StringLabel { name: String }
  |                      ------------ this field does not implement `Copy`
```

Почему же пользовательские типы автоматически не признаются копируемыми в случае, когда это возможно? От того, является тип копируемым или нет, сильно зависит, как его можно использовать: копируемые типы более гибкие, поскольку присваивание и родственные операции не оставляют исходную переменную неинициализированной. Но для автора типа ситуация выглядит прямо противоположно: копируемые типы крайне ограничены в отношении того, какие типы они могут содержать, тогда как некопируемые могут выделять память из кучи и пользоваться другими ресурсами. Поэтому объявление типа копируемым влечет серьезные обязательства со стороны его автора: если впоследствии тип понадобится сделать некопируемым, то код, в котором он использовался, возможно, придется модифицировать.

В C++ разрешено перегружать оператор присваивания и определять специальные копирующие и перемещающие конструкторы, но в Rust таких механизмов нет. В Rust любая передача значения – это побайтовое поверхностное копирование, оставляющее источник неинициализированным. Копирование производится точно так же, только источник остается инициализированным. Это означает, что классы C++ могут предоставлять удобные интерфейсы, когда за обманчиво простым кодом скрываются подсчет ссылок, откладывание накладного копирования на потом или другие хитрые трюки на уровне реализации. Типы Rust ничего подобного делать не умеют.

Но за эту гибкость C++ расплачивается тем, что такие базовые операции, как присваивание, передача параметров и возврат значений из функций, оказываются менее предсказуемыми. Например, выше в данной главе мы показали, что присваивание одной переменной другой в C++ может потреблять произвольный

объем памяти и процессорного времени. Один из принципов Rust – все накладные расходы должны быть понятны программисту. Базовые операции должны оставаться простыми. Потенциально дорогостоящие операции должны быть явными, как, например, обращение к функции `clone` в примере выше для глубокого копирования векторов и хранящихся в них строк.

В этом разделе копируемость и клонируемость рассматривались как некие свойства – `Copy` и `Clone`, – которыми может обладать тип. На самом деле это примеры *характеристик* (trait) – принятого в Rust расширяемого механизма для классификации типов по тому, что с ними можно делать. В общем виде характеристики рассматриваются в главе 11, а конкретно характеристики `Copy` и `Clone` – в главе 13.

Rc и Arc: совместное владение

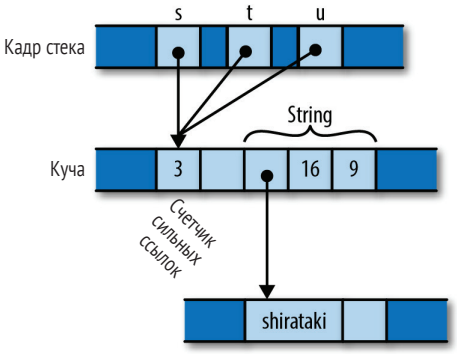
Хотя в типичной Rust-программе у большинства значений владелец только один, бывают случаи, когда трудно найти для каждого значения единственного владельца с подходящим временем жизни; нужно, чтобы значение просто существовало до тех пор, пока кто-то им пользуется. Для таких случаев Rust предлагает указательные типы с подсчетом ссылок, `Rc` и `Arc`. Как и следовало ожидать, эти типы абсолютно безопасны: вы не можете забыть о модификации счетчика ссылок, создать другие указатели на значение, так чтобы Rust этого не заметил, или столкнуться с другими проблемами, преследующими указательные типы с подсчетом ссылок в C++.

Типы `Rc` и `Arc` очень похожи, единственное различие между ними в том, что `Arc` можно безопасно разделять между потоками без каких-либо церемоний (`Arc` означает «Atomic Reference Count» – «атомарный счетчик ссылок»), тогда как в типе `Rc` для обновления счетчика ссылок применяется более быстрый, но не потокобезопасный код. Если разделять указатели между потоками не нужно, то и не платите за менее производительный тип `Arc`, а пользуйтесь `Rc`; Rust не даст случайно передать значение такого типа через границы потоков. Во всех остальных отношениях оба типа эквивалентны, поэтому мы будем рассматривать только `Rc`.

Ранее в этой главе мы показали, как счетчики ссылок используются в Python для управления временем жизни значений. В Rust аналогичного результата можно достичь с помощью типа `Rc`. Рассмотрим такой код:

```
use std::rc::Rc;

// Rust может вывести вс эти типы, но для ясности они указаны явно
let s: Rc<String> = Rc::new("shirataki".to_string());
let t: Rc<String> = s.clone();
let u: Rc<String> = s.clone();
```

Для любого типа `T` значение типа `Rc<T>` – указатель на выделенное из кучи значение типа `T`, к которому присоединен счетчик ссылок. Клонирование значения типа `Rc<T>` не приводит к копированию `T`, вместо этого на него создается еще один указатель, и счетчик ссылок увеличивается на 1. После выполнения приведенного выше кода в памяти складывается такая ситуация:



Каждый из трех указателей типа `Rc<String>` ссылается на одну и ту же область памяти, в которой хранится счетчик ссылок и есть место для строки. К самим указателям типа `Rc` применимы обычные правила владения, и, когда последний существующий `Rc`-указатель уничтожается, Rust уничтожает саму строку.

Все обычные методы `String` можно вызывать и для `Rc<String>`.

```
assert!(s.contains("shira"));
assert_eq!(t.find("taki"), Some(5));
println!("{} are quite chewy, almost bouncy, but lack flavor", u);
```

Значение, которым владеет указатель типа `Rc`, неизменяемо. Попытка добавить текст в конец строки:

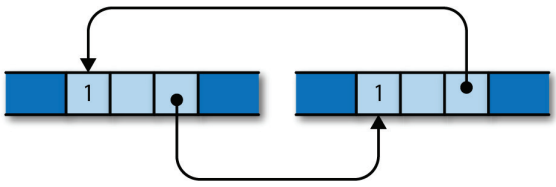
```
s.push_str(" noodles");
```

будет отвергнута Rust:

```
error: cannot borrow immutable borrowed content as mutable
  --> ownership_rc_mutability.rs:12:5
   |
12 |     s.push_str(" noodles");
   |     ^ cannot borrow as mutable
```

Гарантии безопасности при работе с памятью и потоками в Rust основаны на том, что никакое значение не может одновременно иметь несколько владельцев и быть изменяемым. Rust предполагает, что значение, на которое указывает `Rc`-указатель, в общем случае может иметь несколько владельцев, а значит, оно должно быть неизменяемым. Почему это ограничение так важно, мы объясним в главе 5.

У механизма подсчета ссылок для управления памятью есть одна хорошо известная проблема: если два значения с подсчитываемыми ссылками указывают друг на друга, то каждое из них не дает счетчику ссылок на другое значение обратиться в нуль, поэтому ни то, ни другое никогда не будут освобождены:



Из-за этого в Rust возможна утечка значения, но такие ситуации возникают редко. Невозможно создать цикл без того, чтобы в какой-то точке более старое значение указывало на более новое. Очевидно, для этого необходимо, чтобы более старое значение было изменяемым. Поскольку `Rc`-указатели гарантируют неизменяемость значений, на которые указывают, то при обычных обстоятельствах создать цикл невозможно. Однако в Rust есть способы создать неизменяемые значения с изменяемыми частями; это явление, называемое *внутренней изменяемостью*, мы рассмотрим в главе 9. Если такая техника используется в сочетании с `Rc`-указателями, то образование цикла и утечка памяти становятся возможны.

Иногда образования циклов `Rc`-указателей можно избежать, воспользовавшись «слабыми указателями» типа `std::rc::Weak` для некоторых связей. Но в этой книге мы их рассматривать не будем, обратитесь к документации.

Передача владения и указатели с подсчетом ссылок – два способа ослабить ограничения на дерево владения. В следующей главе мы рассмотрим третий способ: заимствование ссылок на значения. Освоив владение и заимствование, вы минуете самую крутую часть кривой изучения Rust и будете готовы к применению уникальных возможностей этого языка.

Глава 5

Ссылки

От библиотек худа не бывает.
— Марк Миллер

Все встречавшиеся до сих пор указательные типы – непритязательный указатель на кучу `Box<T>` и указатели, скрытые внутри значений типа `String` и `Vec`, – являются владеющими: когда указатель уничтожается, вместе с ним уничтожается и значение, на которое он указывает. В Rust имеются также *ссылки* – невладеющие указательные типы, которые никак не влияют на время жизни объектов, на которые ссылаются.

На самом деле верно прямо противоположное: ссылка не должна существовать дольше, чем ее объект. Из кода должно с очевидностью следовать, что ссылка не живет дольше объекта, на который указывает. Чтобы подчеркнуть это, в Rust создание ссылки на значение называется «заимствованием» этого значения, а взятое в долг нужно будет в конечном итоге вернуть владельцу.

Если фраза «должно с очевидностью следовать» вызывает у вас скептическую реакцию, то вы не одиноки. В ссылках как таковых нет ничего особенного, на внутреннем уровне это просто адреса. Но вот правила, благодаря которым они безопасны, – новый вклад Rust; ни в каких языках, кроме экспериментальных, ничего подобного до сих пор не встречалось. И хотя эти правила – та часть Rust, для овладения которой требуется больше всего усилий, разнообразие классических, повседневных ошибок, которых с их помощью удается избежать, поистине впечатляет, а на многопоточное программирование они оказывают раскрепощающий эффект. И это тоже бесшабашная ставка Rust.

В качестве примера рассмотрим построение таблицы тех деятелей искусства эпохи Возрождения, которые совершили убийство, и прославивших их работ. В стандартной библиотеке Rust имеется тип хеш-таблицы, воспользуемся им для определения собственного типа:

```
use std::collections::HashMap;
type Table = HashMap<String, Vec<String>>;
```

Иными словами, эта хеш-таблица, отображающая значения типа `String` на значения типа `Vec<String>`, сопоставляет имени автора список названий его произведений. Элементы `HashMap` можно обойти в цикле `for`, так что мы можем написать отладочную функцию, которая распечатывает таблицу:

```
fn show(table: Table) {
    for (artist, works) in table {
```

```
        println!("работы {}:", artist);
        for work in works {
            println!("  {}", work);
        }
    }
}
```

Построение и печать таблицы не вызывают затруднений:

```
fn main() {
    let mut table = Table::new();
    table.insert("Джезуальдо".to_string(),
                 vec!["мадригалы".to_string(),
                      "Tenebrae Responsoria".to_string()]);
    table.insert("Караваджо".to_string(),
                 vec!["Музыканты".to_string(),
                      "Призвание апостола Матфея".to_string()]);
    table.insert("Челлини".to_string(),
                 vec!["Персей с головой Медузы".to_string(),
                      "Сальера".to_string()]);

    show(table);
}
```

И все отлично работает.

```
$ cargo run
     Running `/home/jimb/rust/book/fragments/target/debug/fragments`
работы Джезуальдо:
  Tenebrae Responsoria
  мадригалы
работы Челлини:
  Персей с головой Медузы
  Сальера
работы Караваджо:
  Музыканты
  Призвание апостола Матфея
$
```

Но если вы прочитали раздел предыдущей главы, посвященный передаче владения, то определение функции `show` должно вызвать вопросы. В частности, тип `HashMap` не является копируемым – да и не может быть таковым, поскольку владеет динамически выделенной таблицей. Поэтому, когда программа вызывает `show(table)`, вся структура передается функции, а переменная `table` оказывается неинициализированной. Если вызывающая сторона попробует использовать `table` снова, то возникнет проблема:

```
show(table);
assert_eq!(table["Джезуальдо"][0], "мадригалы");
```

Rust сообщает, что переменная `table` стала недоступной:

```
error[E0382]: use of moved value: `table`
  --> references_show_moves_table.rs:29:16
   |
28 |     show(table);
```

```
   |          ----- value moved here
29 |     assert_eq!(table["Джезуальдо"][0], "мадригалы");
   |                ^^^^^ value used here after move
   |
   = note: move occurs because `table` has type `HashMap<String, Vec<String>>`,
           which does not implement the `Copy` trait
```

Заглянув внутрь определения `show`, мы увидим, что внешний цикл `for` принимает владение хеш-таблицей и полностью потребляет ее, а внутренний цикл делает то же самое с каждым вектором (с таким поведением мы уже встречались выше в примере «Liberté, égalité, fraternité»). Из-за семантики передачи владения мы полностью уничтожили всю структуру, всего лишь распечатав ее. Вот спасибо, Rust!

По-хорошему здесь нужно было использовать ссылки. Ссылка дает доступ к значению, не покушаясь на владение. Бывают ссылки двух видов:

- *разделяемая ссылка* позволяет читать значение, но не позволяет изменять его. Зато в каждый момент времени можно иметь сколь угодно много разделяемых ссылок на одно значение. Выражение `&e` дает разделяемую ссылку на значение `e`; если `e` имеет тип `T`, то `&e` имеет тип `&T` (читается «ссылка на `T`»). Разделяемые ссылки – копируемые типы;
- *изменяемая ссылка* позволяет читать и модифицировать значение. Но вместе с ней не должно существовать никаких других ссылок на то же значение. Выражение `&mut e` дает изменяемую ссылку на значение `e`, ее тип записывается в виде `&mut T`. Изменяемые ссылки – некопируемые типы.

Различие между разделяемыми и изменяемыми ссылками можно трактовать как проверку соблюдения правила «много читателей или один писатель» на этапе компиляции. На самом деле это правило применимо не только к ссылкам, но и к владельцу заимствованного значения. Пока существуют разделяемые ссылки на значение, даже его владелец не вправе изменить его – значение заблокировано. Никто не сможет модифицировать таблицу `table`, пока с ней работает функция `show`. Аналогично, если существует изменяемая ссылка на значение, то она обладает исключительным доступом к этому значению; с владельцем нельзя производить никаких действий, пока изменяемая ссылка не пропадет. Не смешивать разделение с изменяемостью – ключ к безопасной работе с памятью, а о причинах мы поговорим ниже в этой главе.

Функции печати в данном примере не нужно модифицировать таблицу, достаточно прочитать ее. Поэтому вызывающая сторона может передать ссылку на таблицу:

```
show(&table);
```

Ссылки – это невладеющие указатели, поэтому переменная `table` остается владельцем всей структуры, а `show` только на время позаимствовала ее. Разумеется, определение `show` необходимо подправить, но чтобы увидеть разницу, надо смотреть очень пристально:

```
fn show(table: &Table) {
    for (artist, works) in table {
        println!("работы {}:", artist);
        for work in works {
            println!("  {}", work);
        }
    }
}
```

Тип параметра `table` изменился с `Table` на `&Table`: вместо передачи таблицы по значению (сопровождаемой переходом владения к функции) мы передаем разделяемую ссылку. Всего-то небольшое изменение текста. Но как оно проявляется в теле функции?

Если в первоначальной версии внешний цикл `for` принимал владение хеш-таблицей `HashMap` и потреблял ее, то в новой версии он получает разделяемую ссылку на `HashMap`. Согласно определению, обход `HashMap` по разделяемой ссылке порождает разделяемые ссылки на пары (ключ, значение): переменная `artist` теперь имеет тип `&String`, а не `String`, а переменная `works` – `&Vec<String>`, а не `Vec<String>`.

Внутренний цикл претерпел аналогичные изменения. По определению, обход вектора по разделяемой ссылке порождает разделяемые ссылки на его элементы, потому `work` теперь имеет тип `&String`. Нигде в функции владение не передается, передаются лишь невладеющие ссылки.

Но если мы захотим написать функцию, которая сортирует работы каждого автора по алфавиту, то разделяемой ссылки будет недостаточно, поскольку такие ссылки не позволяют модифицировать объект. Функция сортировки должна получать изменяемую ссылку на таблицу:

```
fn sort_works(table: &mut Table) {
    for (_artist, works) in table {
        works.sort();
    }
}
```

А мы должны передать такую ссылку:

```
sort_works(&mut table);
```

Это изменяемое заимствование позволяет `sort_works` читать и модифицировать нашу структуру, чего и требует метод вектора `sort`.

Передавая функции значение способом, который передает и владение им, мы говорим, что передача произведена «по значению». Если же функции передается ссылка на значение, то говорят о передаче параметра «по ссылке». Мы исправили код функции `show`, заменив передачу таблицы по значению передачей по ссылке. Это различие существует во многих языках, но в Rust оно особенно важно, потому что напрямую отражается на владении.

Ссылки как значения

В примере выше мы видели довольно типичное использование ссылок: наделение функции возможностью обращаться к структуре или манипулировать ей, не принимая на себя владения. Но гибкость ссылок этим не ограничивается, и ниже мы на примерах детально рассмотрим, что происходит.

Сравнение ссылок в Rust и в C++

Ссылки в C++ действительно имеют общие черты со ссылками в Rust. Самое главное – те и другие на машинном уровне являются просто адресами. Но на практике ссылки Rust используются совсем иначе.

В C++ ссылки создаются неявно в процессе преобразования и разыменовываются тоже неявно:

```
// Код написан на C++!
int x = 10;
int &r = x;           // при инициализации неявно создается ссылка
assert(r == 10);      // неявное разыменование r для получения значения x
r = 20;               // значение 20 сохраняется в x, а r по-прежнему указывает на x
```

В Rust ссылки создаются явно оператором `&` и явно разыменовываются оператором `*`:

```
// Здесь и далее снова код Rust.
let x = 10;
let r = &x;           // &x - разделяемая ссылка на x
assert!(*r == 10);    // явное разыменование r
```

Для создания изменяемой ссылки служит оператор `&mut`:

```
let mut y = 32;
let m = &mut y;       // &mut y - изменяемая ссылка на y
*m += 32;             // явно разыменовываем m, чтобы изменить значение y
assert!(*m == 64);    // и проверяем новое значение y
```

Но ведь в исправленной версии функции `show`, в которую таблица авторов передавалась по ссылке, а не по значению, оператор `*` нигде не использовался. Почему так?

Поскольку ссылки широко распространены в Rust, оператор `.` неявно разыменовывает свой левый операнд, если это необходимо:

```
struct Anime { name: &'static str, bechdel_pass: bool };
let aria = Anime { name: "Aria: The Animation", bechdel_pass: true };
let anime_ref = &aria;
assert_eq!(anime_ref.name, "Aria: The Animation");

// Эквивалентно, но разыменование явное:
assert_eq!((*anime_ref).name, "Aria: The Animation");
```

Макрос `println!` в функции `show` расширяется в код, содержащий оператор `.`, поэтому он тоже выигрывает от неявного разыменования.

Оператор `.` может также неявно заимствовать ссылку на свой левый операнд, если это необходимо для вызова метода. Например, метод `sort` типа `Vec` принимает изменяемую ссылку на метод, поэтому следующие два вызова эквивалентны:

```
let mut v = vec![1973, 1968];
v.sort();             // неявно заимствует изменяемую ссылку на v
(&mut v).sort();      // эквивалентно, но уж очень некрасиво
```

Короче говоря, если в C++ преобразования между ссылками и l-значениями (выражениями, ссылающимися на области памяти) неявные и могут производиться всюду, где необходимо, то в Rust для создания и разыменования ссылки служат операторы `&` и `*`, а единственным исключением является оператор `.`, который осуществляет заимствование и разыменование неявно.

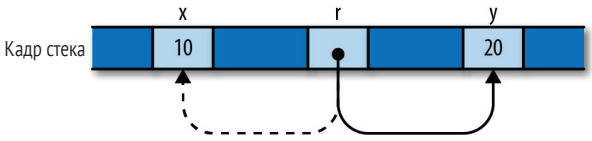
Присваивание ссылкам

В результате присваивания ссылке та начинает указывать на новое значение:

```
let x = 10;
let y = 20;
```

```
let mut r = &x;
if b { r = &y; }
assert!(*r == 10 || *r == 20);
```

Первоначально ссылка `r` указывает на `x`. Но если `b` равно `true`, то программа перенаправляет ее на `y`:



В C++ дело обстоит совершенно по-другому – там присваивание ссылке приводит к записи нового значения в объект ссылки. В C++ невозможно направить ранее инициализированную ссылку на другое значение.

Ссылки на ссылки

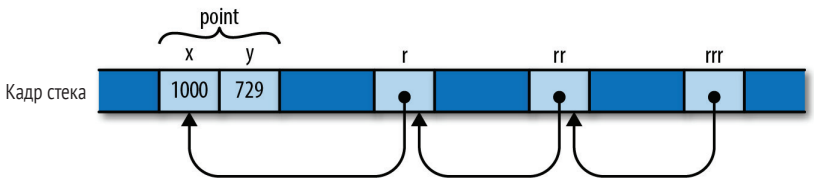
В Rust разрешены ссылки на ссылки:

```
struct Point { x: i32, y: i32 }
let point = Point { x: 1000, y: 729 };
let r: &Point = &point;
let rr: &&Point = &r;
let rrr: &&&Point = &rr;
```

(Для ясности мы явно выписали ссылочные типы, но их можно и опустить, в этом коде нет ничего, что Rust не смог бы вывести самостоятельно.) Оператор `.` проследует по ссылке столько раз, сколько необходимо для доступа к значению:

```
assert_eq!(rrr.y, 729);
```

В памяти эти ссылки выглядят так:



Таким образом, выражение `rrr.y` – в полном соответствии с типом `rrr` – производит троекратный переход по ссылке для нахождения структуры, а затем уже выделяет из нее поле `y`.

Сравнение ссылок

Подобно оператору `.`, операторы сравнения в Rust «видят сквозь» любое число ссылок при условии, что типы обоих операндов одинаковы:

```
let x = 10;
let y = 10;
```

```
let rx = &x;
let ry = &y;

let rrx = &rx;
let rry = &ry;

assert!(rrx <= rry);
assert!(rrx == rry);
```

Последнее утверждение выполняется, несмотря на то что `rrx` и `rry` указывают на разные значения (`rx` и `ry`), т. к. оператор `==` проследует по всем ссылкам и сравнивает целевые объекты в конце цепочек, `x` и `y`. Почти всегда это именно то, что нам нужно, особенно при написании универсальных функций. Если все же нужно знать, указывают ли две ссылки на один и тот же адрес в памяти, то можно воспользоваться функцией `std::ptr::eq`, которая сравнивает ссылки как адреса:

```
assert!(rx == ry);              // объекты ссылки равны
assert!(!std::ptr::eq(rx, ry)); // но расположены по разным адресам
```

Ссылки не бывают нулевыми

Ссылки никогда не бывают нулевыми. Не существует аналогий с указателем `NULL` в C или `nullptr` в C++; у ссылки нет начального значения по умолчанию (никакую переменную, вне зависимости от типа, нельзя использовать, пока она не инициализирована); и Rust не умеет преобразовывать целое число в ссылку (разве что в коде, помеченном `unsafe`), поэтому преобразовать 0 в ссылку тоже не получится.

В C и C++ нулевой указатель часто используется, чтобы показать отсутствие значения. Например, функция `malloc` возвращает либо указатель на новую область памяти, либо `nullptr`, если памяти недостаточно. В Rust, когда требуется значение, которое либо является ссылкой на что-то, либо не является, следует использовать тип `Option<&T>`. На машинном уровне `None` представляется в виде нулевого указателя, а `Some(r)`, где `r` – значение типа `&T` – в виде ненулевого адреса, так что тип `Option<&T>` по эффективности не уступает указателям в C и C++, способным принимать нулевые значения, но безопаснее: тип устроен так, что вы просто не можете не проверить, равно значение `None` или нет, перед тем как его использовать.

Заимствование ссылок на произвольные выражения

Если в C и C++ оператор `&` разрешается применять только к выражениям определенного вида, то Rust позволяет заимствовать ссылку на значение выражения любого вида:

```
fn factorial(n: usize) -> usize {
    (1..n+1).fold(1, |a, b| a * b)
}
let r = &factorial(6);
assert_eq!(r + &1009, 1729);
```

В подобных ситуациях Rust просто создает анонимную переменную для хранения значения выражения и направляет на нее ссылку. Время жизни анонимной переменной зависит от того, что вы делаете со ссылкой:

- если ссылка сразу же присваивается переменной в предложении `let` (или делается частью какой-то структуры или массива, который сразу же присваи-

вается переменной), то Rust обеспечивает анонимной переменной такую же долгую жизнь, как у переменной, которая инициализирована в предложении `let`. В примере выше так происходит для объекта ссылки `r`;

- в противном случае анонимная переменная существует до конца объемлющего предложения. В нашем примере анонимная переменная, созданная для хранения значения `1009`, существует только до конца предложения `assert_eq!`.

Для тех, кто привык к C или C++, это может показаться потенциальным источником ошибок. Но напомним, что Rust никогда не позволит написать код, в котором присутствует висячая ссылка. Если бы ссылка могла использоваться, когда анонимная переменная уже прекратила существование, то Rust сообщил бы об ошибке еще на этапе компиляции. И тогда мы могли бы исправить код, поместив объект ссылки в именованную переменную с подходящим временем жизни.

Ссылки на срезки и объекты характеристик

Все ссылки, встречавшиеся до сих пор, были простыми адресами. Но в Rust есть также два вида «толстых указателей». Так называются значения, занимающие два слова, в которых хранятся адрес некоторого значения и дополнительная информация, необходимая для его использования.

Ссылка на срезку является толстым указателем, в котором хранятся начальный адрес и длина срезки. Срезки подробно рассматривались в главе 3.

Еще один вид толстого указателя – «объект характеристики», ссылка на значение, реализующее некоторую характеристику. В объекте характеристики хранятся адрес значения и указатель на реализацию характеристики, соответствующую этому значению, чтобы можно было вызывать методы характеристики. Подробно объекты характеристик будут рассмотрены в главе 11.

Если отвлечься от дополнительных данных, то ссылки на срезки и объекты характеристик ведут себя точно так же, как все остальные виды ссылок: они не владеют значениями, на которые указывают; они не могут жить дольше этих значений; они бывают разделяемыми и изменяемыми и т. д.

Безопасность ссылок

Судя по всему, сказанному выше, ссылки очень похожи на обычные указатели в C или C++. Но ведь те небезопасны, как же Rust умудряется держать ссылки под контролем? Быть может, лучший способ понаблюдать за правилами в действии – попробовать нарушить их? Начнем с простейшего примера, а затем добавим интересные усложнения и объясним, как все работает.

Заимствование локальной переменной

Это довольно очевидный случай. Нельзя заимствовать ссылку на локальную переменную и вынести ее за пределы области видимости самой переменной:

```
{
    let r;
    {
        let x = 1;
        r = &x;
```

```
    }
    assert_eq!(*r, 1);  // ошибка: читается память, которую когда-то занимала `x`
}
```

Компилятор Rust отказывается принимать такую программу и выдает сообщение

```
error: `x` does not live long enough
  --> references_dangling.rs:8:5
   |
7  |         r = &x;
   |             - borrow occurs here
8  |     }
   |     ^ `x` dropped here while still borrowed
9  |     assert_eq!(*r, 1); // bad: reads memory `x` used to occupy
10 | }
   | - borrowed value needs to live until here
```

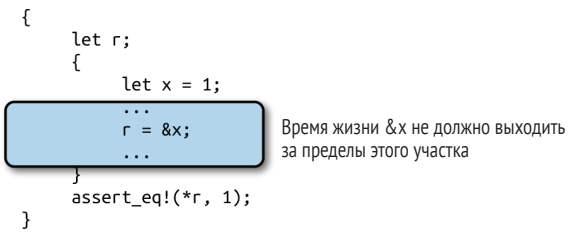
Rust ругается, что `x` существует только до конца внутреннего блока, тогда как ссылка на нее остается жива до конца внешнего блока, в результате чего образуется висячий указатель, а это запрещено.

Конечно, человеку понятно, что эта программа некорректна, но интересно понять, как к этому выводу пришел компилятор. Даже на этом простом примере видны логические средства, которые Rust применяет для проверки гораздо более сложного кода.

Rust пытается сопоставить каждому ссылочному типу в программе *время жизни*, удовлетворяющее ограничениям, налагаемым способом его применения. Время жизни – это некоторый участок программы, в котором использовать ссылку безопасно: лексический блок, предложение, выражение, область видимости некоторой переменной и т. п. Время жизни – по сути своей плод воображения компилятора. На этапе выполнения ссылка – не более чем адрес; время ее жизни – это часть ее типа, не имеющая никакого представления во время выполнения.

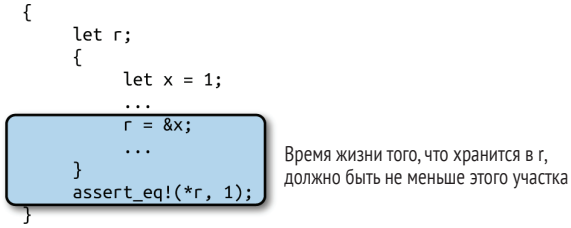
В данном примере есть три времени жизни, связь между которыми следует прояснить. У каждой из переменных `r` и `x` время жизни простирается от точки инициализации до точки выхода из области видимости. Третье время жизни определено для ссылочного типа: типа ссылки, заимствованной в виде `&x` и сохраненной в `r`.

Одно ограничение довольно очевидно: ссылка на переменную `x` не должна существовать дольше самой переменной.



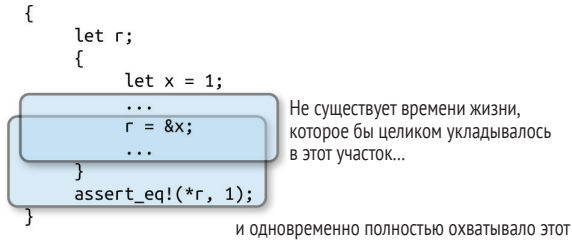
Стоит выйти за точку, где `x` покидает область видимости, как ссылка становится висячим указателем. Мы говорим, что время жизни переменной должно «охватывать» или «содержать» время жизни заимствованной у нее ссылки.

Но есть и другое ограничение: если ссылка сохранена в переменной `r`, то тип ссылки должен оставаться хорошим на протяжении всего времени жизни переменной, от точки инициализации до точки выхода из области видимости.

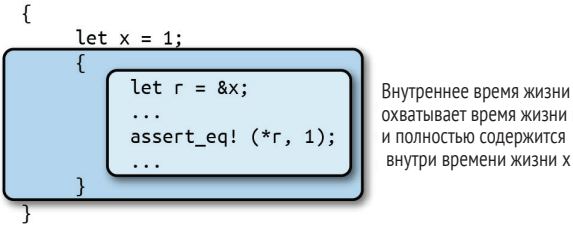


Если ссылка не может прожить столько же, сколько переменная, то в какой-то точке `r` станет висячим указателем. Мы говорим, что время жизни ссылки должно «охватывать» или «содержать» время жизни переменной.

Первое ограничение говорит, насколько большим может быть время жизни ссылки, а второе – насколько малым оно может быть. Rust пытается подобрать для каждой ссылки время жизни, удовлетворяющее всем ограничениям. Но в нашем примере такого времени жизни не существует.



Приведем другой пример, где все складывается ко всеобщему удовольствию. Ограничения те же: время жизни ссылки должно содержаться внутри времени жизни `x` и полностью охватываться временем жизни `r`. Но поскольку теперь время жизни `r` меньше, удается найти время жизни, удовлетворяющее обоим ограничениям:



Эти правила естественно применяются и в случае, когда мы заимствуем ссылку на какую-то часть большей структуры данных, например на элемент вектора:

```
let v = vec![1, 2, 3];
let r = &v[1];
```

Поскольку `v` владеет вектором, который владеет своими элементами, время жизни `v` должно охватывать время жизни ссылочного типа `&v[1]`. Аналогично, если ссылка сохраняется в какой-то структуре данных, то ее время жизни должно охватывать время жизни этой структуры. Если, к примеру, мы создаем вектор ссылок, то все они должны иметь время жизни, охватывающее время жизни переменной, владеющей вектором.

Это суть процедуры, которую Rust применяет ко всему коду. Включая в рассмотрение другие языковые средства – скажем, структуры данных и вызовы функций, – мы вводим дополнительные ограничения, но принцип не изменяется: сначала определить, какие ограничения вытекают из способа использования ссылок в программе, а затем подобрать времена жизни, удовлетворяющие всем ограничениям. Это мало чем отличается от процедуры, которую программисты на C и C++ выполняют по собственной инициативе, разница только в том, что Rust знает правила и настаивает на их соблюдении.

Получение ссылок в качестве параметров

Как Rust гарантирует, что функция безопасно использует переданную ей ссылку? Пусть имеется функция `f`, которая принимает ссылку и сохраняет ее в глобальной переменной. Нам придется несколько раз переделывать этот код, но начнем с такой попытки:

```
// В этом коде несколько ошибок, он не компилируется.
static mut STASH: &i32;
fn f(p: &i32) { STASH = p; }
```

Эквивалент глобальной переменной в Rust – *статическая* переменная: это значение, которое создается в момент запуска программы и существует до момента ее завершения. (Как и для любого другого объявления, видимость статических переменных контролируется системой модулей Rust, так что «глобально» только время их жизни, а не видимость.) Статические переменные рассматриваются в главе 8, а пока просто перечислим правила, которым этот код не удовлетворяет:

- каждая статическая переменная должна быть инициализирована;
- изменяемые статические переменные принципиально не являются потокобезопасными (ведь любой поток может получить доступ к статической переменной в любое время), и даже в однопоточных программах они являются источником различных ошибок, связанных с реентерабельностью. Поэтому доступ к изменяемой статической переменной возможен только внутри блока с пометкой `unsafe`. В данном примере эти проблемы нас не очень волнуют, поэтому просто добавим `unsafe`-блок и пойдем дальше.

После исправления имеем:

```
static mut STASH: &i32 = &128;
fn f(p: &i32) { // этого еще не достаточно
    unsafe {
        STASH = p;
    }
}
```

Почти все сделано. Чтобы увидеть, в чем состоит оставшаяся проблема, выпишем явно некоторые части, которые Rust любезно позволяет опускать. Сигнатура функции `f` – на самом деле сокращенный вариант такой записи:

```
fn f<'a>(p: &'a i32) { ... }
```

Здесь время жизни `'a` (произносится «апостроф А») – это *параметрическое время жизни* `f`. Конструкцию `<'a>` можно прочитать как «для любого времени жизни `'a`», так что, записывая `fn f<'a>(p: &'a i32)`, мы определяем функцию, которая принимает ссылку на значение типа `i32` с любым заданным временем жизни `'a`.

Поскольку мы обязаны обеспечить работу при любом времени жизни `'a`, то код должен работать и тогда, когда это наименьшее возможное время жизни: в точности охватывающее вызов `f`. Тогда камнем преткновения становится следующее присваивание:

```
STASH = p;
```

Поскольку `STASH` существует в течение всего времени выполнения программы, хранящийся в ней ссылочный тип должен иметь время жизни той же протяженности; в Rust такое время жизни обозначается `'static`. Но время жизни ссылки `p` – какое-то `'a`, которое может быть любым, лишь бы оно охватывало вызов `f`. Поэтому Rust отвергает наш код:

```
error[E0312]: lifetime of reference outlives lifetime of borrowed content...
 --> references_static.rs:6:17
  |
6 |          STASH = p;
  |                  ^
  |
  = note: ...the reference is valid for the static lifetime...
note: ...but the borrowed content is only valid for the anonymous lifetime #1
      defined on the function body at 4:0
 --> references_static.rs:4:1
  |
4 | / fn f(p: &i32) { // still not good enough
5 | |     unsafe {
6 | |         STASH = p;
7 | |     }
8 | | }
| |_^
```

В этот момент становится ясно, что наша функция не может принять в качестве аргумента любую ссылку. Но у нее есть возможность принять ссылку со временем жизни `'static`: сохранение такой ссылки в переменной `STASH` не может привести к образованию висячего указателя. И действительно, следующий код компилируется:

```
static mut STASH: &i32 = &10;

fn f(p: &'static i32) {
    unsafe {
        STASH = p;
    }
}
```

На этот раз из сигнатуры f следует, что p должна быть ссылкой со временем жизни 'static, так что проблем с ее сохранением в STASH не возникает. Функцию f можно применять только к другим статическим переменным, но только при таком условии можно гарантировать, что в STASH не останется висячий указатель. Итак, можно написать такой код:

```
static WORTH_POINTING_AT: i32 = 1000;
f(&WORTH_POINTING_AT);
```

Поскольку переменная WORTH_POINTING_AT статическая, типом &WORTH_POINTING_AT является &'static i32, а его передавать f безопасно.

Но давайте отступим на шаг и посмотрим, что произошло с сигнатурой f по пути к корректности программы: первоначальная сигнатура f(p: &i32) превратилась в f(p: &'static i32). Иными словами, мы не смогли написать функцию, которая сохраняет ссылку в глобальной переменной, не выразив этого намерения явно в сигнатуре. В Rust сигнатура функции всегда отражает поведение ее тела.

Наоборот, если мы видим функцию с сигнатурой вида g(p: &i32) (или с явно указанным временем жизни, как в g<'a>(p: &'a i32)), то можем сказать, что она заведомо *не записывает* свой аргумент p в какую-то переменную, которая может пережить ее вызов. Даже не надо заглядывать в определение g; сигнатура сама говорит, что может, а чего не может делать g со своим аргументом. Это оказывается очень полезным, когда мы пытаемся установить, безопасен ли вызов функции.

Передача ссылок в качестве аргументов

Поняв, как сигнатура функции соотносится с ее телом, посмотрим, как она соотносится с вызывающей стороной. Возьмем такой код:

```
// Это можно записать короче: fn g(p: &i32),
// но для начала оставим явное указание времени жизни.
fn g<'a>(p: &'a i32) { ... }

let x = 10;
g(&x);
```

Видя только сигнатуру g, Rust знает, что функция не сохраняет p в переменной, которая может пережить вызов: в качестве 'a должно подойти любое время жизни, охватывающее вызов. Поэтому Rust назначает &x наименьшее возможное время жизни: совпадающее с вызовом g. Так удовлетворяются все ограничения: &x не переживает x и охватывает весь вызов g. Так что этот код испытание выдержал.

Отметим, что хотя g принимает параметрическое время жизни 'a, мы не обязаны упоминать его при вызове g. О параметрическом времени жизни нужно думать только при определении функции и типов, а во время использования Rust сам выводит время жизни.

А что, если мы попытаемся передать &x ранее написанной функции f, которая сохраняет свой аргумент в статической переменной?

```
fn f(p: &'static i32) { ... }

let x = 10;
f(&x);
```

Этот код не компилируется: ссылка `&x` не должна жить дольше `x`, но, передавая ее функции `f`, мы говорим, что ее срок жизни не меньше `'static`. Удовлетворить всех невозможно, поэтому Rust отвергает такой код.

Возврат ссылок

Часто бывает так, что функция принимает ссылку на структуру данных и возвращает ссылку на некоторую часть этой структуры. Вот, например, функция, возвращающая ссылку на наименьший элемент срезки:

```
// v должна содержать хотя бы один элемент
fn smallest(v: &[i32]) -> &i32 {
    let mut s = &v[0];
    for r in &v[1..] {
        if *r < *s { s = r; }
    }
    s
}
```

Мы, как обычно, опустили время жизни в сигнатуре функции. Если функция принимает единственную ссылку в качестве аргумента и возвращает одну ссылку, то Rust предполагает, что обе ссылки должны иметь одинаковое время жизни. В явном виде это выглядело бы так:

```
fn smallest<'a>(v: &'a [i32]) -> &'a i32 { ... }
```

Предположим, что функция `smallest` вызывается следующим образом:

```
let s;
{
    let parabola = [9, 4, 1, 0, 1, 4, 9];
    s = smallest(&parabola);
}
assert_eq!(*s, 0); // ошибка: указывает на элемент уничтоженного массива
```

Глядя на сигнатуру `smallest`, мы заключаем, что ее аргумент и возвращаемое значение должны иметь одинаковое время жизни, `'a`. В показанном выше вызове аргумент `&parabola` не должен жить дольше, чем сама переменная `parabola`, а вместе с тем значение, возвращаемое `smallest`, должно жить никак не меньше, чем `s`. Не существует времени жизни, которое удовлетворяло бы обоим ограничениям, поэтому Rust сообщает об ошибке:

```
error: `parabola` does not live long enough
  --> references_lifetimes_propagated.rs:12:5
   |
11 |         s = smallest(&parabola);
   |                      -------- borrow occurs here
12 |     }
   |     ^ `parabola` dropped here while still borrowed
13 |     assert_eq!(*s, 0); // ошибка: указывает на элемент уничтоженного массива
14 | }
   | - borrowed value needs to live until here
```

Если перенести объявление `s` в другое место, так чтобы его время жизни было вложено во время жизни переменной `parabola`, то ошибка исчезнет:

```
{
    let parabola = [9, 4, 1, 0, 1, 4, 9];
    let s = smallest(&parabola);
    assert_eq!(*s, 0); // fine: parabola still alive
}
```

Указание времен жизни в сигнатурах функций дает Rust возможность узнать о связях между переданными функции ссылками и возвращаемым значением и гарантировать их безопасное использование.

Структуры, содержащие ссылки

Как Rust обращается со ссылками в структурах данных? Ниже приведена та же ошибочная программа, что и выше, только ссылка помещена в структуру:

```
// Не компилируется.
struct S {
    r: &i32
}

let s;
{
    let x = 10;
    s = S { r: &x };
}
assert_eq!(*s.r, 10); // ошибка: чтение уничтоженной `x`
```

Ограничения безопасности, налагаемые на ссылки, не могут волшебным образом исчезнуть только потому, что ссылка скрыта внутри структуры. Каким-то образом они должны быть применены к `S`. И действительно, Rust отнесся к этой программе скептически:

```
error[E0106]: missing lifetime specifier
 --> references_in_struct.rs:7:12
  |
7 |         r: &i32
  |            ^ expected lifetime parameter
```

Если ссылочный тип встречается внутри определения другого типа, то время его жизни следует указывать явно. Можно написать так:

```
struct S {
    r: &'static i32
}
```

Здесь говорится, что `r` может ссылаться только на значения типа `i32`, существующие на протяжении всего времени жизни программы, а это довольно сильное ограничение. Альтернатива – включить в тип параметрическое время жизни `'a` и указать его в определении поля `r`:

```
struct S<'a> {
    r: &'a i32
}
```

Теперь у типа `S` имеется время жизни, как у ссылочных типов. Каждое значение типа `S` получает свое время жизни `'a`, на которое налагаются ограничения в со-

ответствии с характером его использования. Время жизни любой ссылки, хранящейся в поле `r`, должно охватывать `'a`, а `'a` должно быть дольше, чем время жизни того места, в котором хранится `S`.

Но вернемся к приведенному выше коду: выражение `S { r: &x }` создает новое значение `S` с каким-то временем жизни `'a`. Сохраняя ссылку `&x` в поле `r`, мы требуем, что `'a` не выходило за пределы времени жизни `x`.

Присваивание `s = S { ... }` сохраняет это значение типа `S` в переменной, время жизни которой простирается до конца примера, т. е. на `'a` налагается ограничение: быть дольше времени жизни `s`. И снова Rust сталкивается с противоречивыми требованиями: `'a` не должно существовать дольше `x`, но при этом должно существовать по меньшей мере столько же, сколько `s`. Подобрать такое время жизни невозможно, поэтому Rust отвергает код. Катастрофу удалось предотвратить!

Как тип с параметрическим временем жизни ведет себя, будучи помещен внутрь другого типа?

```
struct T {
    s: S  // не годится
}
```

Rust недоволен – так же, как и тогда, когда мы поместили ссылку в `S`, не указав время ее жизни:

```
error[E0106]: missing lifetime specifier
 --> references_in_nested_struct.rs:8:8
  |
8 |     s: S // не годится
  |        ^ expected lifetime parameter
```

Опустить время жизни `S` в данном случае нельзя: Rust должен знать, как время жизни типа `T` соотносится со временем жизни ссылки в его поле типа `S`, чтобы применить к `T` те же проверки, что для `S` и для простых ссылок.

Можно было бы назначить `s` время жизни `'static`. Это работает:

```
struct T {
    s: S<'static>
}
```

При таком определении поле `s` может заимствовать только значения, существующие на всем протяжении работы программы. Это несколько ограничительно, но означает, что `T` точно не может заимствовать локальную переменную; на время жизни `T` не налагается никаких специальных ограничений.

Другой подход состоит в том, чтобы включить в тип `T` параметрическое время жизни и передать его `S`:

```
struct T<'a> {
    s: S<'a>
}
```

Благодаря использованию параметрического времени жизни `'a` в типе поля `s` мы дали Rust возможности соотнести время жизни значения типа `T` со временем жизни ссылки, хранящейся в его поле типа `S`.

Выше было показано, как сигнатура функции отражает действия, которые она производит с переданными ей ссылками. Теперь нечто похожее продемонстри-

ровано для типов: параметрическое время жизни типа всегда показывает, есть ли в нем ссылки с интересным (отличающимся от `'static`) временем жизни, и если да, то каким может быть это время жизни.

Пусть, к примеру, имеется функция разбора, которая принимает байтовую срезку и возвращает структуру, содержащую результаты разбора:

```
fn parse_record<'i>(input: &'i [u8]) -> Record<'i> { ... }
```

Даже не заглядывая в определение типа `Record`, мы можем сказать, что если от функции `parse_record` получено значение типа `Record`, то все ссылки, которые оно, возможно, содержит, должны указывать в переданный входной буфер и никуда больше (за исключением, быть может, значений со временем жизни `'static`).

На самом деле такое раскрытие внутреннего поведения и есть причина, по которой Rust требует, чтобы типы, содержащие ссылки, явно принимали параметрическое время жизни. Ничто не мешало бы Rust просто задать свое время жизни для каждой ссылки в структуре и избавить нас от необходимости указывать их явно. Ранние версии Rust именно так себя и вели, но разработчикам это показалось неудобным: полезно знать, когда одно значение что-то заимствует у другого, особенно если приходится разбираться в сообщении об ошибке.

Но не только ссылки и типы, подобные `S`, имеют время жизни. Время жизни есть у каждого типа Rust, включая `i32` и `String`. В большинстве случаев это просто `'static`, т. е. значения данного типа могут жить сколько угодно; например, тип `Vec<i32>` вполне автономен, его не нужно уничтожать, прежде чем какая-то переменная выйдет из области видимости. Но у типа `Vec<&'a i32>` есть время жизни, которое должно охватываться `'a`: этот вектор необходимо уничтожить, пока все объекты ссылок еще живы.

Различные параметрические времена жизни

Рассмотрим структуру, содержащую две ссылки:

```
struct S<'a> {
    x: &'a i32,
    y: &'a i32
}
```

У обеих ссылок одинаковое время жизни `'a`. Это может оказаться проблемой, если программа захочет сделать нечто подобное:

```
let x = 10;
let r;
{
    let y = 20;
    {
        let s = S { x: &x, y: &y };
        r = s.x;
    }
}
```

Этот код не создает висячих указателей. Ссылка на `y` остается в переменной `s`, которая выходит из области видимости раньше `y`. Ссылка на `x` сохраняется в переменной `r`, которая существует не дольше `x`.

Но при попытке откомпилировать этот код Rust ругается, что `y` живет недостаточно долго, хотя это, очевидно, не так. Так чем же обеспокоен Rust? Внимательно изучив код, мы сможем повторить его рассуждения:

- оба члена `S` – ссылки с одинаковым временем жизни `'a`, так что Rust должен найти время жизни, которое годилось бы одновременно для `s.x` и `s.y`;
- мы присвоили `r = s.x`, потребовав тем самым, чтобы `'a` охватывало время жизни `r`;
- мы инициализировали `s.y` ссылкой `&y`, потребовав, чтобы `'a` было не дольше времени жизни `y`.

Этим ограничениям удовлетворить невозможно: не существует времени жизни, которое было бы короче области видимости `y`, но длиннее области видимости `r`. Rust ругается.

Проблема возникает, потому что обе ссылки в `S` имеют одно и то же время жизни `'a`. Все становится на свои места, если изменить определение `S`, так чтобы у каждой ссылки было независимое время жизни:

```
struct S<'a, 'b> {
    x: &'a i32,
    y: &'b i32
}
```

Теперь, что бы мы не делали с `s.x`, это не оказывает влияния на то, что хранится в `s.y`, поэтому обоим ограничениям легко удовлетворить: `'a` может быть просто временем жизни `r`, а `'b` – временем жизни `s`. (В качестве `'b` можно было бы взять и время жизни `y`, но Rust всегда выбирает наименьшее подходящее время жизни.) Все концы срослись.

С сигнатурами функций дело обстоит аналогично. Рассмотрим такую функцию:

```
fn f<'a>(r: &'a i32, s: &'a i32) -> &'a i32 { r } // пожалуй, слишком строго
```

Здесь у обоих параметров одинаковое время жизни `'a`, из-за чего на вызывающую сторону могут быть наложены излишние ограничения – такие же, как мы видели выше. Если это вызывает проблемы, можно сделать так, чтобы времена жизни могли изменяться независимо:

```
fn f<'a, 'b>(r: &'a i32, s: &'b i32) -> &'a i32 { r } // не так строго
```

Недостаток такого решения в том, что из-за увеличения количества разных времен жизни типы и сигнатуры функций становится труднее читать. Авторы этой книги обычно начинают с простейшего возможного определения и ослабляют ограничения, если код не компилируется. Поскольку Rust не даст запустить небезопасный код, то дождаться его подсказки о том, где скрывается проблема, – вполне приемлемая тактика.

Опускание параметрического времени жизни

Мы уже видели немало функций, которые возвращают ссылки или принимают их в качестве параметров, но обычно не указывали время жизни. Время жизни есть всегда, но Rust позволяет опускать его, если очевидно, каким оно должно быть.

В простейшем случае, если функция не возвращает ссылки (или другие типы, требующие параметрического времени жизни), то явно задавать время жизни па-

раметров и не придется. Rust просто назначает независимое время жизни каждому месту, которое в этом нуждается. Например:

```
struct S<'a, 'b> {
    x: &'a i32,
    y: &'b i32
}

fn sum_r_xy(r: &i32, s: S) -> i32 {
    r + s.x + s.y
}
```

Сигнатура этой функции – сокращенная запись следующей:

```
fn sum_r_xy<'a, 'b, 'c>(r: &'a i32, s: S<'b, 'c>) -> i32
```

Если же функция возвращает ссылки или иные типы, требующие параметрического времени жизни, то Rust все равно пытается найти простое, не допускающее двоякого толкования решение. Если в параметрах функции встречается только одно время жизни, то Rust предполагает, что и в возвращаемом значении времена жизни должны быть такими же:

```
fn first_third(point: &[i32; 3]) -> (&i32, &i32) {
    (&point[0], &point[2])
}
```

Если явно указать все времена жизни, то получится такая сигнатура:

```
fn first_third<'a>(point: &'a [i32; 3]) -> (&'a i32, &'a i32)
```

Если в параметрах встречается несколько времен жизни, то нет оснований предпочесть одно из них в качестве времени жизни возвращенного значения, поэтому Rust требует явного указания.

Но все же имеется еще одна возможность сократить код: если функция является методом некоторого типа и принимает параметр `self` по ссылке, то дилемма решается: Rust предполагает, что время жизни во всех компонентах возвращаемого значения совпадает со временем жизни `self`. (Параметр `self` ссылается на значение, от имени которого вызван метод, это Rust'овский эквивалент `this` в C++, Java и JavaScript или `self` в Python. Методы рассматриваются в главе 9.)

Например, мы можем написать такой код:

```
struct StringTable {
    elements: Vec<String>,
}

impl StringTable {
    fn find_by_prefix(&self, prefix: &str) -> Option<&String> {
        for i in 0 .. self.elements.len() {
            if self.elements[i].starts_with(prefix) {
                return Some(&self.elements[i]);
            }
        }
        None
    }
}
```

Сигнатура метода `find_by_prefix` – сокращенная запись такой:

```
fn find_by_prefix<'a, 'b>(&'a self, prefix: &'b str) -> Option<&'a String>
```

Rust предполагает, что все заимствования производятся у `self`.

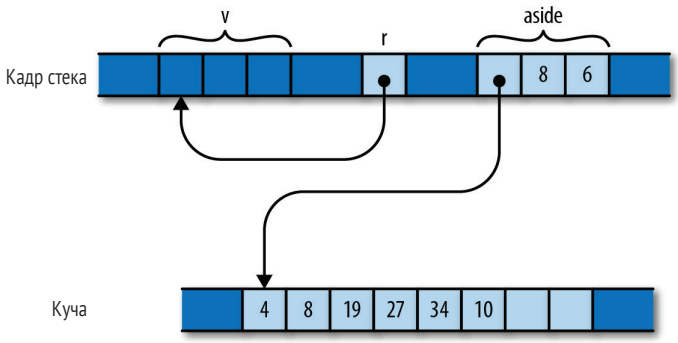
Повторим, что все это – всего лишь сокращения, призванные приносить пользу без сюрпризов. Если получается не то, что вы имели в виду, то всегда можно указать времена жизни явно.

Разделяемость и изменяемость

До сих пор мы обсуждали, как Rust гарантирует, что никакая ссылка никогда не указывает на переменную, покинувшую область видимости. Но висячие указатели могут возникать и по-другому. Вот простой пример:

```
let v = vec![4, 8, 19, 27, 34, 10];
let r = &v;
let aside = v;  // передать владение вектором переменной aside
r[0];           // ошибка: использование неинициализированной переменной `v`
```

Присваивание переменной `aside` приводит к передаче вектора, а `v` становится неинициализированной, так что `r` оказывается висячим указателем.



Хотя `v` остается в области видимости на протяжении всего времени жизни `r`, проблема в том, что значение `v` передается в другое место, оставляя вектор `v` неинициализированным в момент, когда `r` все еще указывает на его элемент. Естественно, Rust обнаруживает такую ошибку:

```
error[E0505]: cannot move out of `v` because it is borrowed
  --> references_sharing_vs_mutation_1.rs:10:9
   |
9  |     let r = &v;
   |             - borrow of `v` occurs here
10 |     let aside = v; // передать владение вектором переменной aside
   |         ^^^^^ move out of `v` occurs here
```

На протяжении всего времени своей жизни разделяемая ссылка делает свой объект доступным только для чтения: не разрешается присваивать что-то объекту ссылки или передавать кому-то владение значением. В приведенном выше

коде на протяжении времени жизни `r` производится попытка передать владение вектором, поэтому Rust отвергает программу. Изменив программу, как показано ниже, мы устраним проблему:

```
let v = vec![4, 8, 19, 27, 34, 10];
{
    let r = &v;
    r[0];       // все правильно: вектор по-прежнему здесь
}
let aside = v;
```

В этом варианте `r` покидает область видимости раньше, время жизни ссылки заканчивается, прежде чем владение `v` передано `aside`, и все кончается благополучно.

Вот другой способ внести хаос. Рассмотрим функцию, которая расширяет вектор, добавляя в него элементы срезки:

```
fn extend(vec: &mut Vec<f64>, slice: &[f64]) {
    for elt in slice {
        vec.push(*elt);
    }
}
```

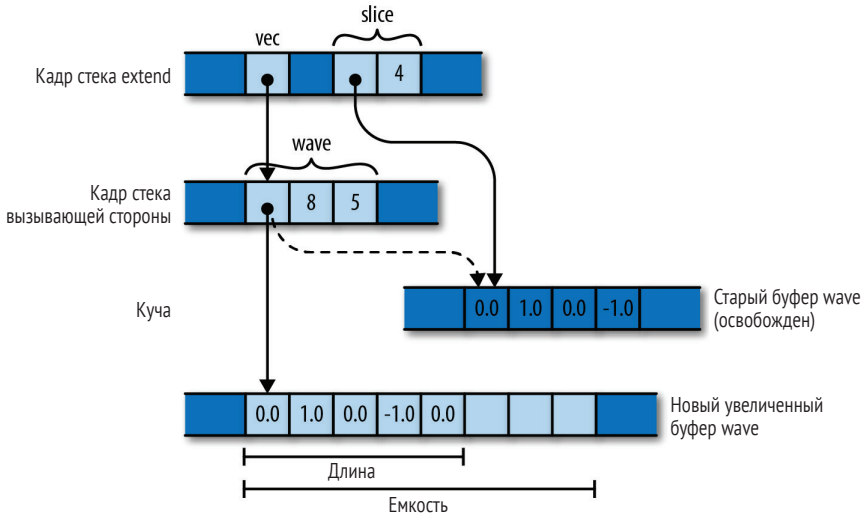
Это менее гибкая (и гораздо хуже оптимизированная) версия метода вектора `extend_from_slice` из стандартной библиотеки. Мы можем воспользоваться ей для построения вектора из срезок других векторов или массивов:

```
let mut wave = Vec::new();
let head = vec![0.0, 1.0];
let tail = [0.0, -1.0];

extend(&mut wave, &head);   // расширить wave другим вектором
extend(&mut wave, &tail);   // расширить wave массивом

assert_eq!(wave, vec![0.0, 1.0, 0.0, -1.0]);
```

Итак, мы построили один период синусоиды. А чтобы добавить еще один, можно ли дописать вектор в конец самого себя?

```
extend(&mut wave, &wave);
assert_eq!(wave, vec![0.0, 1.0, 0.0, -1.0,
                      0.0, 1.0, 0.0, -1.0]);
```

На первый взгляд, ничего страшного. Но вспомним, что при добавлении элемента в момент, когда буфер вектора заполнен, необходимо выделить память для нового буфера большего размера. Предположим, что изначально в векторе `wave` было место для четырех элементов, так что при попытке добавить пятый происходит выделение большего буфера. В конечном итоге память будет выглядеть так:



Аргумент `vec` функции `extend` заимствует значение `wave` (принадлежащее вызывающей стороне), под которое выделен новый буфер на восемь элементов. Но `slice` по-прежнему указывает на старый четырехэлементный буфер, который уже уничтожен.

Эта проблема свойственна не только Rust: модификация коллекций в момент, когда имеется указатель на их элементы, – тонкая материя во многих языках. В C++ спецификация `std::vector` предупреждает, что «перераспределение [буфера вектора] делает недействительными все ссылки, указатели и итераторы, указывающие на элементы последовательности». А в Java о модификации объекта `java.util.Hashtable` сказано так:

> *Если структурная модификация Hashtable произведена после создания итератора любым способом, кроме собственного метода итератора* `remove`*, то итератор возбуждает исключение* `ConcurrentModificationException`*.*

Самое неприятное, что подобные ошибки происходят не каждый раз. При тестировании может всякий раз оказываться, что в буфере вектора достаточно места и он не перераспределяется, так что проблема никогда не проявляется.

Но Rust сообщает об ошибке при обращении к `extend` на этапе компиляции:

```
error[E0502]: cannot borrow `wave` as immutable because it is also borrowed as mutable
 --> references_sharing_vs_mutation_2.rs:9:24
  |
9 |     extend(&mut wave, &wave);
  |                ---- ^^^^- mutable borrow ends here
  |                |    |
  |                |    immutable borrow occurs here
  |                mutable borrow occurs here
```

Иными словами, разрешается заимствовать изменяемую ссылку на вектор и разрешается заимствовать разделяемую ссылку на его элемент, но времена жизни этих двух ссылок не должны пересекаться. В нашем случае оба времени жизни содержат обращение к `extend`, поэтому Rust отвергает код.

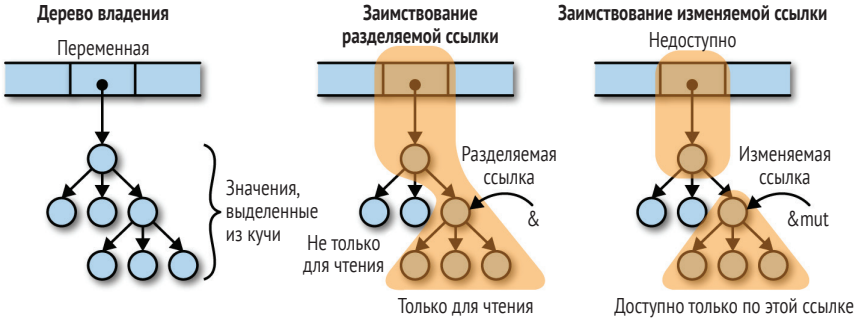
Описанные выше ошибки проистекают из нарушения правил Rust, касающихся изменяемости и разделяемости:

- *разделяемый доступ предназначен только для чтения.* Значения, заимствуемые разделяемыми ссылками, можно только читать. На всем протяжении времени жизни разделяемой ссылки ни значение самого объекта ссылки, ни значение чего-либо, достижимого из этого объекта, нельзя изменять *никаким способом*. Не существует ни одной активной изменяемой ссылки на что-либо внутри структуры, владелец структуры может только читать ее и т. д. Структура действительно заморожена;
- *напротив, изменяемый доступ носит монопольный характер.* Значение, заимствованное изменяемой ссылкой, доступно только по этой ссылке. На протяжении всего времени ее жизни не существует никакого возможного пути к объекту ссылки или к значениям, достижимым из этого объекта. Единственные ссылки, время жизни которых может пересекаться со временем жизни изменяемой ссылки, – те, которые заимствованы у нее самой.

В примере функции `extend` Rust сообщил о нарушении второго правила: поскольку мы заимствовали изменяемую ссылку на `wave`, эта ссылка должна быть единственным путем доступа к вектору или к его элементам. Однако разделяемая ссылка на срезку является еще одним способом добраться до элементов вектора, что нарушает второе правило.

Но Rust мог бы рассматривать эту ошибку и как нарушение первого правила: поскольку мы заимствовали разделяемую ссылку на элементы `wave`, то и элементы, и сам вектор `Vec` должны быть доступны только для чтения. Нельзя заимствовать изменяемую ссылку на значение, доступное только для чтения.

Наличие ссылки любого вида влияет на то, что можно делать со значениями на пути владения объектом ссылки и со значениями, достижимыми из этого объекта:



Заметим, что в обоих случаях путь владения, ведущий к объекту ссылки, нельзя изменять на всем протяжении времени жизни ссылки. Для разделяемого заимствования путь владения доступен только для чтения, а для изменяемого вообще недоступен. Поэтому программа не может совершить ничего такого, что сделало бы ссылку недействительной.

Проиллюстрируем эти принципы на простейших примерах.

```
let mut x = 10;
let r1 = &x;
let r2 = &x;      // правильно: несколько разделяемых заимствований разрешено
```

```
x += 10;          // ошибка: нельзя присвоить значение `x`, потому что она заимствована
let m = &mut x;   // ошибка: нельзя создать изменяемую ссылку на `x`, потому что
                  // на нее уже существует разделяемая ссылка

let mut y = 20;
let m1 = &mut y;
let m2 = &mut y;  // ошибка: нельзя заимствовать изменяемым способом более одного раза
let z = y;        // ошибка: нелья использовать `y`, потому что на нее существует
                  // изменяемая ссылка
```

Разрешается повторно заимствовать разделяемую ссылку у разделяемой ссылки.

```
let mut w = (107, 109);
let r = &w;
let r0 = &r.0;          // правильно: повторное разделяемое заимствование у разделяемой ссылки
let m1 = &mut r.1;      // ошибка: нельзя повторно заимствовать изменяемую ссылку
                        // у разделяемой
```

Повторно заимствовать у разделяемой ссылки разрешается:

```
let mut v = (136, 139);
let m = &mut v;
let m0 = &mut m.0;      // правильно: повторное заимствование изменяемой ссылки у изменяемой
*m0 = 137;
let r1 = &m.1;          // правильно: повторное заимствование разделяемой ссылки у изменяемой,
                        // не пересекается с m0
v.1;                    // ошибка: доступ по другим путям все еще запрещен
```

Эти ограничения довольно строгие. Возвращаясь к нашей попытке вызвать `extend(&mut wave, &wave)`, мы не найдем простого и быстрого способа исправить код, чтобы он работал так, как мы хотим. И эти правила Rust применяет повсеместно: если, скажем, мы позаимствовали разделяемую ссылку на ключ в структуре `HashMap`, то не сможем позаимствовать изменяемую ссылку на `HashMap`, пока не закончится время жизни разделяемой.

Но всему этому есть убедительное обоснование: проектировать коллекции, без ограничений поддерживающие одновременное итерирование и модификацию, трудно, и зачастую при этом ускользают от внимания более простые и эффективные реализации. Классы `Hashtable` в Java и `vector` в C++, равно как словари в Python и объекты в JavaScript, не определяют точного поведения при таком доступе. Для других типов коллекций в JavaScript определение есть, но зато и реализация становится более тяжеловесной. Класс C++ `std::map` обещает, что вставка новых записей не делает указатели на другие записи отображения недействительными, но, давая такое обещание, стандарт отвлекает от более эффективных для реализации кэша структур типа имеющейся в Rust `BTreeMap`, где в каждом узле дерева хранится несколько записей.

Вот еще один пример ошибки, отлавливаемой благодаря этим правилам. Рассмотрим следующий написанный на C++ код для управления дескриптором файла. Для простоты мы включили только конструктор и оператор копирующего присваивания и опустили обработку ошибок:

```
struct File {
  int descriptor;

  File(int d) : descriptor(d) { }
```

```
  File& operator=(const File &rhs) {
    close(descriptor);
    descriptor = dup(rhs.descriptor);
  }
};
```

Оператор присваивания совсем простой, но приводит к катастрофической ошибке в такой ситуации:

```
File f(open("foo.txt", ...));
...
f = f;
```

Если присвоить объект `File` самому себе, то `rhs` и `*this` будут указывать на один и тот же объект, поэтому `operator=` закрывает тот самый дескриптор файла, который собирается передать функции `dup`. Мы уничтожили ресурс, который должны были скопировать.

В Rust аналогичный код имел бы вид:

```
struct File {
    descriptor: i32
}

fn new_file(d: i32) -> File {
    File { descriptor: d }
}

fn clone_from(this: &mut File, rhs: &File) {
    close(this.descriptor);
    this.descriptor = dup(rhs.descriptor);
}
```

(Это не идиоматичный Rust. В главе 9 описано, как правильно наделять типы Rust конструкторами и методами, но в данном примере достаточно и таких определений.)

На Rust код, соответствующий приведенному выше использованию `File`, выглядел бы так:

```
let mut f = new_file(open("foo.txt", ...));
...
clone_from(&mut f, &f);
```

Но, разумеется, Rust даже компилировать такой код отказывается:

```
error[E0502]: cannot borrow `f` as immutable because it is also borrowed as mutable
  --> references_self_assignment.rs:18:25
   |
18 |     clone_from(&mut f, &f);
   |                -   ^- mutable borrow ends here
   |                |   |
   |                |   immutable borrow occurs here
   |                mutable borrow occurs here
```

Здесь нам все уже знакомо. Как выясняется, за двумя классическими ошибками в C++ – неправильной обработкой присваивания самому себе и использованием недействительных итераторов – на самом деле стоит одна и та же ошибка! В обо-

их случаях программа предполагает, что модифицирует одно значение, сверяясь с другим, тогда как фактически это одно и то же значение! Еще одну форму эта ошибка принимает, когда при обращении к функциям `memcpy` или `strcpy` в C или C++ оказывается, что начальная и конечная области пересекаются. Требуя, чтобы доступ по изменяемой ссылке был монопольным, Rust избавился от целого класса широко распространенных ошибок.

Истинная сила различия между разделяемыми и изменяемыми ссылками раскрывается при написании конкурентного кода. Гонка за данные возможна только тогда, когда к некоторому значению осуществляется изменяющий и разделяемый доступ из разных потоков, а это именно то, что правила работы со ссылками в Rust запрещают. Конкурентная Rust-программа, в которой нет кода, помеченного `unsafe`, *по построению* не содержит гонок за данные. Мы рассмотрим этот аспект более детально, когда будем говорить о конкурентности в главе 19, но уже сейчас можно сказать, что написание конкурентных программ на Rust гораздо проще, чем на большинстве других языков.

Сравнение разделяемых ссылок в Rust с указателями на const в C

На первый взгляд, разделяемые ссылки в Rust сильно напоминают указатели на `const` в C и C++. Однако правила Rust, относящиеся к разделяемым ссылкам, гораздо строже. Рассмотрим, к примеру, следующий код на C:

```
int x = 42;          // переменная типа int, не const
const int *p = &x;   // указатель на const int
assert(*p == 42);
x++;                 // изменить переменную напрямую
assert(*p == 43);    // значение "константного" объекта ссылки изменилось
```

Тот факт, что `p` имеет тип `const int *`, означает лишь, что объект ссылки нельзя изменить с помощью самой ссылки `p`: выражение `(*p)++` запрещено. Но к значению объекта ссылки можно обратиться просто как к `x`, в объявлении которого нет слова `const`, и таким образом изменить его. У ключевого слова `const` в языках, производных от C, есть свои применения, но гарантия постоянства не из их числа.

В Rust разделяемая ссылка запрещает любые модификации своего объекта до тех пор, пока не закончится время ее жизни:

```
let mut x = 42;      // неконстантная переменная типа i32
let p = &x;          // разделяемая ссылка на i32
assert_eq!(*p, 42);
x += 1;              // ошибка: нельзя изменить значение x, потому что оно заимствовано
assert_eq!(*p, 42);  // если убрать присваивание, это утверждение по-прежнему истинно
```

Чтобы гарантировать постоянство значения, мы должны проследить все возможные пути к нему и для каждого убедиться, что либо на нем значение не модифицируется, либо он вообще не используется. Указатели в C и C++ недостаточно ограничительны, чтобы компилятор мог выполнить такие проверки. Со ссылками в Rust всегда связано конкретное время жизни, поэтому их можно проверить на этапе компиляции.

Оружие против моря объектов

С тех пор как в 1990-х годах получило распространение автоматическое управление памятью, архитектура всех программ по умолчанию представляет собой «море объектов», изображенное на рис. 5.1.

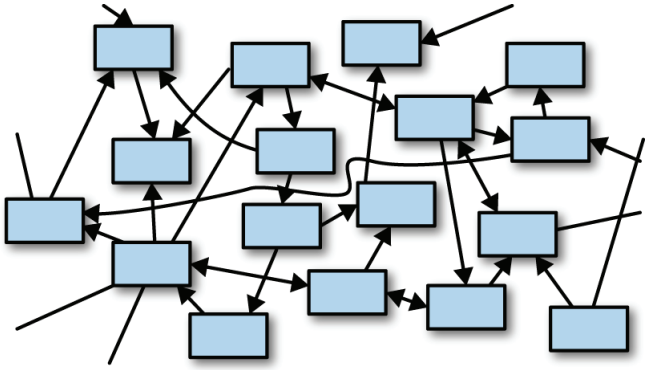


Рис. 5.1 ❖ Море объектов

Так происходит, если система располагает сборщиком мусора и вы начинаете писать программу без предварительного проектирования. Всем нам доводилось создавать такие системы.

У такой архитектуры много достоинств, не отраженных на рисунке: быстрый прогресс вначале, простота непредусмотренных изменений. В общем, несколько ближайших лет не сулят таких проблем, из-за которых все пришлось бы переписывать (вспомните песню «Дорога в ад» группы AC/DC).

Но, конечно, есть и недостатки. Когда все зависит от всего, становится трудно тестировать программу, развивать ее и даже размышлять о каждом компоненте в отдельности.

Замечательная особенность Rust состоит в том, что модель владения ставит ограничитель скорости на дороге в ад. Чтобы организовать циклическую зависимость в Rust, нужно приложить усилия, поскольку в этом случае два значения будут содержать ссылки друг на друга. Придется воспользоваться типом интеллектуального указателя, например `Rc`, и внутренней изменяемостью (эту тему мы еще не рассматривали). Rust предпочитает, чтобы указатели, владение и поток данных в системе смотрели в одном направлении, как на рис. 5.2.

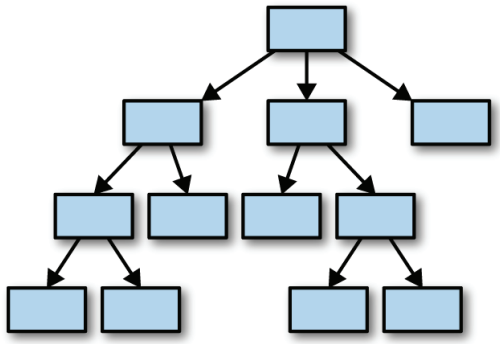


Рис. 5.2 ❖ Дерево значений

Мы подняли этот вопрос именно сейчас, потому что после прочтения данной главы естественно возникает желание, не откладывая дело в долгий ящик, создать «море структур», связанных `Rc`-указателями, и воспроизвести тем самым все антипаттерны объектно-ориентированного программирования, с которыми вы знакомы. Но так просто не получится. Модель владения Rust создаст затруднения. А лекарство состоит в том, чтобы сначала сесть и подумать, а потом написать более качественную программу.

Весь смысл Rust заключается в том, чтобы перенести трудные раздумья о структуре программы из будущего в настоящее. И получается на удивление хорошо: Rust не только заставляет понять, почему программа потокобезопасна, но и требует выполнения некоторого объема высокоуровневого архитектурного проектирования.

Глава 6

Выражения

> Программисты на LISP знают значение всего, но не знают цену ничему.
>
> *— Алан Перлис, эпиграмма 55*

В этой главе мы рассмотрим *выражения* Rust – те строительные блоки, из которых состоит тело функции. Ряд концепций, а именно замыкания и итераторы, настолько глубок, что мы посвятим им отдельные главы. А пока наша цель – рассмотреть столько синтаксических конструкций, сколько возможно на нескольких страницах.

Язык выражений

Rust визуально напоминает семейство языков C, но это только на поверхностный взгляд. В C существует четкое различие между *выражениями*, например:

```
5 * (fahr-32) / 9
```

и предложениями, которые выглядят так:

```
for (; begin != end; ++begin) {
    if (*begin == target)
        break;
}
```

У выражения есть значение, у предложения нет.

Язык Rust относится к так называемым *языкам выражений*, т. е. следует более давней традиции, восходящей к Lisp, где на выражения возлагалась вся работа.

В C конструкции `if` и `switch` являются предложениями. Они не порождают значений и не могут встречаться в середине выражения. В Rust `if` и `match` *могут* порождать значения. В главе 2 мы видели, как выражение `match` порождает числовое значение:

```
pixels[r * bounds.0 + c] =
    match escapes(Complex { re: point.0, im: point.1 }, 255) {
        None => 0,
        Some(count) => 255 - count as u8
    };
```

Выражение `if` может использоваться для инициализации переменной:

```
let status =
    if cpu.temperature <= MAX_TEMP {
```

```
        HttpStatus::Ok
    } else {
        HttpStatus::ServerError  // сервер расплавился
    };
```

Выражение `match` можно передавать в качестве аргумента функции или макросу:

```
println!("Inside the vat, you see {}.",
    match vat.contents {
        Some(brain) => brain.desc(),
        None => "nothing of interest"
    });
```

Этим объясняется отсутствие в Rust тернарного оператора *(expr1 ? expr2 : expr3)*. В C это удобный аналог предложения `if`, являющийся выражением. А в Rust он был бы избыточным: выражение `if` может служить обеим целям.

В C поток управления в основном реализован с помощью предложений, а в Rust всюду используются выражения.

Блоки и точки с запятой

Блоки тоже являются выражениями. Блок порождает значение и может использоваться всюду, где требуется значение.

```
let display_name = match post.author() {
    Some(author) => author.name(),
    None => {
        let network_info = post.get_network_metadata()?;
        let ip = network_info.client_address();
        ip.to_string()
    }
};
```

Код после `Some(author) =>` – простое выражение `author.name()`. Код после `None =>` – блочное выражение. Для Rust между ними нет разницы. Значением этого блока является значение последнего выражения в нем, `ip.to_string()`.

Обратите внимание на отсутствие точки с запятой после этого выражения. Большинство строк в Rust-программе заканчивается точкой с запятой или фигурной скобкой, как в C и Java. И если блок выглядит как код на C, где все точки с запятой стоят на знакомых местах, то он и выполняется, как блок в C, а его значением будет `()`. Но в главе 2 мы уже отмечали, что если последняя строка блока не заканчивается точкой с запятой, то этот блок порождает значение, а именно значение последнего выражения.

В некоторых языках, в частности в JavaScript, разрешено опускать точки с запятой, и язык дописывает их сам – это просто мелкое удобство. Но тут мы имеем другой случай – в Rust у точки с запятой есть семантика.

```
let msg = {
    // объявление let: точка с запятой обязательна
    let dandelion_control = puffball.open();

    // выражение + точка с запятой: вызывается метод,
```

```
    // возвращенное значение отбрасывается
    dandelion_control.release_all_seeds(launch_codes);

    // выражение без точка с запятой: вызывается метод,
    // возвращенное значение сохраняется в `msg`
    dandelion_control.get_status()
};
```

Способность блока содержать объявления и при этом порождать в конце значение – удобная особенность, которая быстро становится привычной и естественной. Недостаток в том, что если случайно забыть точку с запятой, то выдается странное сообщение об ошибке:

```
...
if preferences.changed() {
    page.compute_size()  // пропущена точка с запятой
}
...
```

Если сделать такую ошибку в программе на C или Java, то компилятор просто скажет, что отсутствует точка с запятой. А вот что говорит Rust:

```
error[E0308]: mismatched types
    --> expressions_missing_semicolon.rs:19:9
    |
19 |           page.compute_size() // пропущена точка с запятой
    |            ^^^^^^^^^^^^^^^^^^^ expected (), found tuple
    |
    = note: expected type `()`
               found type `(u32, u32)`
```

Rust считает, что точка с запятой опущена сознательно, он даже не рассматривает возможности, что это опечатка. И в результате – непонятное сообщение. Увидев слова `expected type ()`, прежде всего смотрите, не пропущена ли точка с запятой.

В блоках разрешены также *пустые предложения*. Пустое предложение состоит из одной лишь точки с запятой:

```
loop {
    work();
    play();
    ;         // <-- пустое предложение
}
```

Здесь Rust следует традиции C. Пустое предложение всего лишь навевает легкую меланхолию, а больше ничего не делает. Мы упомянули их только для полноты.

Объявления

Помимо выражений и точек с запятой, блок может содержать объявления в любом количестве. Чаще всего встречаются объявления локальных переменных `let`:

```
let name: type = expr;
```

Тип и инициализатор необязательны. Точка с запятой обязательна.

В объявлении `let` можно объявить переменную, не инициализируя ее. Впоследствии переменная будет инициализирована присваиванием. Иногда это полезно, поскольку бывает, что переменную следует инициализировать в середине некоторой управляющей конструкции:

```
let name;
if user.has_nickname() {
    name = user.nickname();
} else {
    name = generate_unique_name();
    user.register(&name);
}
```

Здесь мы видим два разных способа инициализации локальной переменной `name`, но в любом случае она инициализируется ровно один раз, поэтому указывать в объявлении ключевое слово `mut` необязательно.

Попытка использовать переменную до ее инициализации является ошибкой. (Здесь есть тесная связь с другой ошибкой: использованием значения, переданного кому-то другому. Rust категорически настаивает на том, что значения должны использоваться только тогда, когда они существуют!)

Иногда можно встретить код, в котором существующая переменная якобы переопределяется, например:

```
for line in file.lines() {
    let line = line?;
    ...
}
```

Этот код эквивалентен такому:

```
for line_result in file.lines() {
    let line = line_result?;
    ...
}
```

В объявлении `let` создается новая переменная другого типа. Переменная `line_result` имеет тип `Result<String, io::Error>`, а вторая переменная, `line`, – тип `String`. Разрешается давать второй переменной такое же имя, как первой. В этой книге мы в таких ситуациях предпочитаем использовать суффикс `_result`, чтобы у всех переменных были разные имена.

Блок может также содержать *объявления артикулов*. Артикул (item) – это просто любое объявление, которое может встречаться глобально в программе или модуле, например `fn`, `struct` или `use`.

В следующих главах артикулы будут рассмотрены подробно. А пока в качестве примера возьмем `fn`. Любой блок может содержать объявление `fn`:

```
use std::io;
use std::cmp::Ordering;

fn show_files() -> io::Result<()> {
    let mut v = vec![];
    ...
```

```
    fn cmp_by_timestamp_then_name(a: &FileInfo, b: &FileInfo) -> Ordering {
        a.timestamp.cmp(&b.timestamp)    // сначала сравниваем временные метки
            .reverse()                   // чем файл новее, тем он меньше
            .then(a.path.cmp(&b.path))   // для устранения неоднозначностей сравниваем пути
    }

    v.sort_by(cmp_by_timestamp_then_name);
    ...
}
```

Когда функция `fn` объявлена внутри блока, ее областью видимости является весь блок, т. е. она может *использоваться* в любом месте объемлющего блока. Но вложенная функция не может обращаться к локальным переменным или аргументам в этой области видимости. Так, функция `cmp_by_timestamp_then_name` не может непосредственно использовать `v`. (В Rust имеются также замыкания, которые могут заглядывать в объемлющие области видимости. См. главу 14.)

Блок может содержать даже целый модуль. Может показаться, что это уже перебор – так ли уж нужно иметь возможность вкладывать *любую* языковую конструкцию в любую другую? – но программисты (а особенно те из них, кто пользуется макросами) находят применение для любого вида ортогональности, предлагаемого языком.

IF И MATCH

Выражение `if` всем знакомо:

```
if condition1 {
    block1
} else if condition2 {
    block2
} else {
    block_n
}
```

Каждое *условие* (condition) должно быть выражением типа `bool`; оставаясь верным своим принципам, Rust не выполняет неявного преобразования чисел или указателей в булевы значения.

В отличие от C, круглые скобки вокруг условий необязательны. Более того, `rustc` выдает предупреждение, если встретит ненужные скобки. Но фигурные скобки должны присутствовать.

Блоки `else if`, а также заключительная часть `else` необязательны. Выражение `if` без блока `else` ведет себя так, будто присутствует пустой блок `else`.

Выражение `match` чем-то напоминает предложение `switch` в C, но обладает большей гибкостью. Вот простой пример:

```
match code {
    0 => println!("OK"),
    1 => println!("Перепутались провода"),
    2 => println!("Пользователь спит"),
    _ => println!("Неизвестная ошибка {}", code)
}
```

Такое можно было бы сделать и с помощью предложения switch. Выполняется ровно одна из четырех ветвей выражения match, в зависимости от значения code. Универсальный образец _ сопоставляется со всем на свете, т. е. выступает в роли ветви default:.

Выражения match такого вида компилятор может оптимизировать с помощью таблицы переходов, как то происходит с предложением switch в C++. Подобная оптимизация применяется, когда каждая ветвь match порождает константу. В таком случае компилятор строит массив констант, и match компилируется в оператор доступа к массиву. Если не считать проверки выхода за границы, в откомпилированном коде вообще нет ветвлений.

Гибкость match проистекает из многообразия поддерживаемых *образцов*, которые могут находиться слева от => в каждой ветви. Выше в роли образцов выступали просто целые константы. Мы также встречали выражения match, различающие два варианта значения Option:

```
match params.get("name") {
    Some(name) => println!("Привет, {}!", name),
    None => println!("Приветствую, незнакомец.")
}
```

И это лишь малая толика того, на что способны образцы. Образец может сопоставляться с диапазоном. Он может распаковывать кортежи. Он может сопоставляться с отдельными полями структуры. Он умеет прослеживать ссылки. Он может заимствовать части значения. И еще много чего. Образцы в Rust образуют отдельный мини-язык. В главе 10 мы посвятим им несколько страниц.

Общий синтаксис выражения match таков:

```
match value {
    pattern => expr,
    ...
}
```

Запятую после ветви можно опускать, если выражение *expr* – блок.

Rust сравнивает значение *value* с каждым образцом по очереди, начиная с первого. После нахождения подходящего образца вычисляется соответствующее ему выражение *expr*, и на этом вычисление выражения match завершается, остальные образцы не проверяются. Хотя бы один образец должен подойти. Rust запрещает выражения match, в которых не охвачены все возможные значения:

```
let score = match card.rank {
    Jack => 10,
    Queen => 10,
    Ace => 11
};  // ошибка: образцы не исчерпывают всех возможностей
```

Все блоки выражения if должны порождать значения одного и того же типа:

```
let suggested_pet =
    if with_wings { Pet::Buzzard } else { Pet::Hyena };  // правильно

let favorite_number =
    if user.is_hobbit() { "eleventy-one" } else { 9 };  // ошибка
```

```
let best_sports_team =
    if is_hockey_season() { "Predators" };  // ошибка
```

(Последний пример ошибочен, потому что в июле результат будет равен `()`[1].)

Точно так же все ветви выражения `match` должны быть одного типа:

```
let suggested_pet =
    match favorites.element {
        Fire => Pet::RedPanda,
        Air => Pet::Buffalo,
        Water => Pet::Orca,
        _ => None  // ошибка: несовместимые типы
};
```

if let

Существует еще одна форма выражения `if` – `if let`:

```
if let pattern = expr {
    block1
} else {
    block2
}
```

Если выражение *expr* соответствует образцу *pattern*, то выполняется блок *block1*, иначе блок *block2*. Иногда это оказывается элегантным способом получить данные из `Option` или `Result`:

```
if let Some(cookie) = request.session_cookie {
    return restore_session(cookie);
}

if let Err(err) = present_cheesy_anti_robot_task() {
    log_robot_attempt(err);
    politely_accuse_user_of_being_a_robot();
} else {
    session.mark_as_human();
}
```

Необходимости в использовании выражения `if let` не бывает никогда, потому что все то же самое можно сделать с помощью `match`. Так что `if let` – всего лишь сокращенная запись `match` с единственным образцом:

```
match expr {
    pattern => { block1 }
    _ => { block2 }
}
```

Циклы

Существуют четыре выражения цикла:

```
while condition {
    block
}
```

[1] Речь идет о лучшей хоккейной команде сезона. Понятно, что в июле никакой лучшей команды нет. – *Прим. перев.*

```
while let pattern = expr {
    block
}

loop {
    block
}

for pattern in collection {
    block
}
```

В Rust циклы являются выражениями, но не порождают полезных значений. Значением цикла всегда является `()`.

Цикл `while` ведет себя в точности так же, как в C, с тем исключением, что условие *condition* должно иметь тип `bool` и никакой другой.

Цикл `while let` аналогичен выражению `if let`. В начале каждой итерации выражение *expr* сопоставляется с образцом *pattern*, и если они соответствуют друг другу, то блок выполняется, а в противном случае происходит выход из цикла.

Ключевое слово `loop` служит для записи бесконечных циклов. Блок *block* выполняется до тех пор, пока не встретится выражение `break` или `return` либо в потоке не возникнет паника.

Цикл `for` вычисляет выражение *collection*, а затем выполняет блок *block* по одному разу для каждого значения в коллекции. Поддерживаются разнообразные коллекции. Стандартный цикл `for` в языке C

```
for (int i = 0; i < 20; i++) {
    printf("%d\n", i);
}
```

в Rust записывается так:

```
for i in 0..20 {
    println!("{}", i);
}
```

Как и в C, последним напечатанным числом будет `19`.

Оператор `..` порождает *диапазон*, простую структуру с двумя полями: `start` и `end`. Диапазон `0..20` – то же самое, что `std::ops::Range { start: 0, end: 20 }`. Диапазоны можно использовать в циклах `for`, потому что тип `Range` допускает итерирование: он реализует характеристику `std::iter::IntoIterator`, которую мы обсудим в главе 15. Все стандартные коллекции, а также массивы и срезки итерируемы.

В соответствии с семантикой передачи владения цикл `for` получает значение во владение:

```
let strings: Vec<String> = error_messages();
for s in strings {                  // здесь каждая строка передается в s...
    println!("{}", s);
}                                   // ...а здесь уничтожается
println!("{} error(s)", strings.len()); // ошибка: использование переданного значения
```

Это не всегда удобно. Чтобы избежать этого, можно сделать переменной цикла ссылку на коллекцию. Тогда она будет последовательно принимать ссылку на значение каждого элемента коллекции:

```
for rs in &strings {
    println!("String {:?} is at address {:p}.", *rs, rs);
}
```

Здесь `&strings` имеет тип `&Vec<String>`, а `rs` – тип `&String`.

Итерирование по изменяемой ссылке дает изменяемую ссылку на каждый элемент:

```
for rs in &mut strings {  // rs имеет тип &mut String
    rs.push('\n');  // добавляем в конец каждой строки символ новой строки
}
```

В главе 15 мы подробнее рассмотрим циклы `for` и продемонстрируем еще много способов использования итераторов.

Выражение `break` осуществляет выход из объемлющего цикла. (В Rust `break` можно употреблять только в циклах. В выражениях `match` оно не нужно, в отличие от предложений `switch` в C.)

Выражение `continue` служит для перехода в начало следующей итерации цикла.

```
// Читать данные по одной строке.
for line in input_lines {
    let trimmed = trim_comments_and_whitespace(line);
    if trimmed.is_empty() {
        // Перейти в начало цикла и продолжить со следующей входной строки.
        continue;
    }
    ...
}
```

В цикле `for` выражение `continue` переходит к следующему значению в коллекции. Если значений больше не осталось, происходит выход из цикла. А в цикле `while` выражение `continue` осуществляет повторную проверку условия цикла. Если оно принимает значение `false`, то происходит выход из цикла.

Цикл может быть *помечен* временем жизни. В примере ниже `'search:` – метка внешнего цикла `for`. Поэтому `break 'search` означает выход из этого цикла, а не из внутреннего.

```
'search:
for room in apartment {
    for spot in room.hiding_spots() {
        if spot.contains(keys) {
            println!("Выше ключи находятся в тайнике {} в комнате {}.", spot, room);
            break 'search;
        }
    }
}
```

Метку можно указывать также в выражении `continue`.

Выражение return

Выражение `return` осуществляет выход из текущей функции с возвратом значения вызывающей стороне.

`return` без указания значения – сокращение для `return ()`:

```
fn f() {      // тип возвращаемого значения опущен, по умолчанию ()
    return;   // возвращаемое значение опущено, по умолчанию ()
}
```

Как и выражение `break`, `return` может прекратить текущую работу. Например, в главе мы использовали оператор `?` для проверки ошибок после вызова функции, которая может завершиться неудачно:

```
let output = File::create(filename)?;
```

и объяснили, что эта строчка – сокращенная запись следующей конструкции:

```
let output = match File::create(filename) {
    Ok(f) => f,
    Err(err) => return Err(err)
};
```

Этот код начинается с вызова `File::create(filename)`. Если возвращено `Ok(f)`, то значением всего выражения `match` будет `f`, поэтому `f` сохраняется в переменной `output`, и мы продолжаем выполнение со строки, следующей за `match`.

Иначе произойдет сопоставление с `Err(err)`, и мы встретим выражение `return`. Если это случится, то уже не важно, что мы находимся в середине выражения `match` и пытаемся вычислить значение переменной `output`. Вся текущая работа прекращается, и мы выходим из объемлющей функции, возвращая ошибку, полученную от `File::create()`.

Оператор `?` будет детально рассмотрен в следующей главе.

Зачем в Rust цикл loop

Некоторые компоненты компилятора Rust анализируют поток управления в программе:

- Rust проверяет, что на каждом пути выполнения функции возвращается значение ожидаемого типа. Чтобы сделать это правильно, необходимо знать, можно ли вообще дойти до конца функции;
- Rust проверяет, что неинициализированные переменные никогда не используются. Для этого нужно убедиться, что ни на одном пути выполнения функции невозможно дойти до использования переменной раньше, чем она инициализирована;
- Rust предупреждает о недостижимом коде. Код называется недостижимым, если не существует *ни одного* пути выполнения, по которому можно дойти до него.

Такие виды анализа называются *потокозависимыми* (flow-sensitive). Ничего нового в них нет, в Java уже много лет применяется аналогичный анализ «заведомого присваивания».

Навязывая такие правила, язык должен соблюсти баланс между простотой, чтобы программисту было легко понять, что имел в виду компилятор, выдавая сообщение об ошибке, и изощренностью, чтобы не выдавать ложных предупреждений и не отвергать вполне безопасную программу. Rust отдает предпочтение простоте. В ходе потокозависимого анализа условия цикла вообще не проверя-

ются, а просто предполагается, что любое условие в программе может быть либо истинным, либо ложным.

В результате Rust отвергает некоторые безопасные программы:

```
fn wait_for_process(process: &mut Process) -> i32 {
    while true {
        if process.wait() {
            return process.exit_code();
        }
    }
}  // ошибка: не на всех путях управления возвращается значение
```

На самом деле ошибки здесь нет, потому что невозможно дойти до конца функции, не вернув значения.

Выражение `loop` введено в язык, чтобы программист мог решить эту проблему, выразив именно то, что имел в виду.

Система типов в Rust также зависит от потока управления. Выше мы сказали, что все ветви выражения `if` должны иметь одинаковый тип. Но было бы глупо требовать выполнения этого правила для блоков, которые завершаются выражением `break` или `return`, для бесконечных циклов `loop` или для вызовов `panic!()` либо `std::process:exit()`. У всех этих выражений имеется одна общая черта: они не завершаются обычным способом с порождением значения. Выражения `break` и `return` производят немедленный выход из блока, бесконечный цикл `loop` вообще не завершается и т. д.

Поэтому в Rust эти выражения не имеют нормального типа. Выражениям, которые не завершаются нормально, назначается специальный тип `!`, и к ним не применяются правила об обязательном совпадении типов. Тип `!` встречается в сигнатуре функции `std::process::exit()`:

```
fn exit(code: i32) -> !
```

Здесь `!` означает, что `exit()` никогда не возвращает управление. Это *уходящая функция* (divergent function).

Вы можете и сами написать уходящую функцию, применяя такой же синтаксис, и в некоторых случаях это совершенно естественное решение:

```
fn serve_forever(socket: ServerSocket, handler: ServerHandler) -> ! {
    socket.listen();
    loop {
        let s = socket.accept();
        handler.handle(s);
    }
}
```

Разумеется, Rust сочтет ошибкой нормальный возврат из такой функции.

На этом завершается часть главы, посвященная потоку управления. Далее будут рассмотрены функции, методы и операторы в Rust.

Вызовы функций и методов

Синтаксис вызова функций и методов в Rust такой же, как во многих других языках:

```
let x = gcd(1302, 462);  // вызов функции
let room = player.location();  // вызов метода
```

Во втором примере `player` – переменная пользовательского типа `Player`, в котором имеется метод `.location()`. (О том, как определять собственные методы, мы будем говорить в главе 9 при обсуждении пользовательских типов.)

Обычно в Rust проводится четкое различие между ссылками и значениями, на которые они ссылаются. Если передать значение типа `&i32` функции, ожидающей получить значение типа `i32`, то это будет считаться ошибкой типизации. Но оператор `.` немного ослабляет эти правила. В приведенной выше строке `player` может иметь тип `Player`, тип ссылки `&Player` или тип интеллектуального указателя `Box<Player>` либо `Rc<Player>`. Метод `player.location()` может принимать игрока по ссылке или по значению. Один и тот же синтаксис `player.location()` работает во всех случаях, потому что оператор `.` автоматически разыменовывает `player` или заимствует ссылку на `player` – в зависимости от контекста.

Третий вариант синтаксиса применяется для вызова статических методов, например `Vec::new()`.

```
let mut numbers = Vec::new();  // вызов статического метода
```

Различие между статическими и нестатическими методами такое же, как в объектно-ориентированных языках: нестатические методы вызываются от имени значений (`my_vec.len()`), а статические – от имени типов (`Vec::new()`).

Естественно, вызовы методов можно сцеплять:

```
Iron::new(router).http("localhost:3000").unwrap();
```

В Rust имеется особенность: при вызове функции или метода обычный синтаксис не работает для универсальных типов, например `Vec<T>`:

```
return Vec<i32>::with_capacity(1000);  // ошибка: что-то насчет сцепленных сравений
let ramp = (0 .. n).collect<Vec<i32>>();  // та же ошибка
```

Проблема в том, что в выражениях `<` – это оператор «меньше». Компилятор Rust услужливо предлагает в таких случаях писать `::<T>` вместо `<T>`, и это решает проблему:

```
return Vec::<i32>::with_capacity(1000);  // правильно, используется ::<
let ramp = (0 .. n).collect::<Vec<i32>>();  // правильно, используется ::<
```

Символ `::<...>` в сообществе Rust любовно называют «турборыба».

Вместо этого зачастую можно просто опустить параметрические типы и позволить Rust вывести их.

```
return Vec::with_capacity(10);  // правильно, если fn возвращает значение типа Vec<i32>
let ramp: Vec<i32> = (0 .. n).collect();  // правильно, указан тип переменной
```

Считается хорошим стилем опускать типы, если их можно вывести.

Поля и элементы

Для доступа к полям структуры применяется знакомый синтаксис. Кортежи в этом отношении похожи, только у их полей имеются номера, а не имена.

```
game.black_pawns  // поле структуры
coords.1          // элемент кортежа
```

Если значение слева от точки – ссылка или интеллектуальный указатель, то производится автоматическое разыменование, как при вызове метода.

Для доступа к элементам массива, срезки или вектора служат квадратные скобки:

```
pieces[i]         // элемент массива
```

Значение слева от скобок автоматически разыменовывается.

Все три показанных выше выражения и им подобные называются *l-значениями*, потому что могут находиться в левой части присваивания:

```
game.black_pawns = 0x00ff0000_00000000_u64;
coords.1 = 0;
pieces[2] = Some(Piece::new(Black, Knight, coords));
```

Конечно, это разрешено, только если переменные `game`, `coords` и `pieces` объявлены изменяемыми.

Вырезание части массива или вектора обозначается так:

```
let second_half = &game_moves[midpoint .. end];
```

Здесь `game_moves` может быть массивом, срезкой или вектором, в любом случае результатом является срезка длины `end - midpoint`.`game_moves`, заимствованная на время жизни переменной `second_half`.

Оператор `..` позволяет опускать любой операнд. Он порождает объекты четырех разных типов в зависимости от того, какие операнды присутствуют:

```
..      // RangeFull
a ..    // RangeFrom { start: a }
.. b    // RangeTo { end: b }
a .. b  // Range { start: a, end: b }
```

Диапазоны в Rust *полуоткрытые*: они включают начальное значение, если оно задано, и не включают конечное. Диапазон `0 .. 4` состоит из чисел `0`, `1`, `2`, `3`.

Только диапазоны, имеющие начальное значение, допускают итерирование, поскольку цикл должен с чего-то начинаться. Но для срезания массивов полезны все четыре формы. Если начало или конец диапазона опущены, то по умолчанию подразумевается начало или конец срезаемых данных.

Таким образом, реализация алгоритма быстрой сортировки, классического примера алгоритмов вида «разделяй и властвуй» может выглядеть так:

```
fn quicksort<T: Ord>(slice: &mut [T]) {
    if slice.len() <= 1 {
        return;  // нечего сортировать.
    }

    // Разделить срезку на две части, переднюю и заднюю.
    let pivot_index = partition(slice);

    // Рекурсивно отсортировать переднюю половину `slice`.
    quicksort(&mut slice[.. pivot_index]);

    // А затем заднюю половину.
    quicksort(&mut slice[pivot_index + 1 ..]);
}
```

Операторы ссылки

Операторы взятия адреса `&` и `&mut` рассматриваются в главе 5.

Унарный оператор `*` служит для доступа к значению, на которое указывает ссылка. Как мы уже видели, Rust автоматически проследует по ссылкам, когда оператор `.` используется для доступа к полю или к методу, поэтому оператор `*` необходим только тогда, когда мы хотим прочитать или записать все значение, адресуемое ссылкой.

Например, иногда итератор порождает ссылки, а программе нужны сами значения:

```
let padovan: Vec<u64> = compute_padovan_sequence(n);
for elem in &padovan {
    draw_triangle(turtle, *elem);
}
```

Здесь `elem` имеет тип `&u64`, поэтому `*elem` имеет тип `u64`.

Арифметические, поразрядные, логические операторы и операторы сравнения

Бинарные операторы в Rust ведут себя так же, как во многих других языках. Для экономии времени мы предполагаем, что читатель знаком с каким-то из этих языков, и сосредоточимся только на моментах, в которых Rust отступает от традиции.

В Rust имеются обычные арифметические операторы `+`, `-`, `*`, `/` и `%`. Как отмечалось в главе 3, в отладочных сборках целочисленное переполнение обнаруживается и приводит к панике. Для неконтролируемой арифметики стандартная библиотека предоставляет метод `a.wrapping_add(b)` и ему подобные.

Деление целого числа на нуль приводит к панике даже в выпускных сборках. У целых чисел имеется метод `a.checked_div(b)`, который возвращает значение типа `Option` (`None`, если `b` равно 0) и никогда не паникует.

Унарный `-` дает число с противоположным знаком. Он поддерживается только для целых со знаком. Унарный `+` не существует вовсе.

```
println!("{}", -100);     // -100
println!("{}", -100u32);  // ошибка: унарный `-` неприменим к типу `u32`
println!("{}", +100);     // ошибка: ожидается выражений, найден `+`
```

Как и в C, выражение `a % b` вычисляет остаток от деления. Знак результата совпадает со знаком левого операнда. Отметим, что оператор `%` применим не только к целым, но и к числам с плавающей точкой.

```
let x = 1234.567 % 10.0;  // приблизительно 4.567
```

Rust унаследовал от C также поразрядные операторы над целыми числами `&`, `|`, `^`, `<<` и `>>`. Однако для поразрядной операции НЕ в Rust используется знак `!`, а не `~`:

```
let hi: u8 = 0xe0;
let lo = !hi;  // 0x1f
```

Следовательно, `!n` в применении к целому числу `n` не означает «n равно 0». Для этого нужно написать `n == 0`.

Поразрядный сдвиг всегда распространяет знаковый разряд в применении к целым типам со знаком и дополняет нулями в применении к целым без знака. Поскольку в Rust имеются целые без знака, необходимости в операторе `>>>`, как в Java, не возникает.

В отличие от C, приоритет поразрядных операций выше, чем операций сравнения, поэтому `x & BIT != 0` означает `(x & BIT) != 0`, и, скорее всего, именно это и имелось в виду. Такая интерпретация гораздо полезнее, чем принятая в C – `x & (BIT != 0)`, – которая проверяет не тот бит!

В Rust имеются следующие операторы сравнения: `==`, `!=`, `<`, `<=`, `>`, `>=`. Как и для любых бинарных операторов, типы операндов должны совпадать.

В Rust также есть закороченные логические операторы `&&` и `||`. Оба операнда должны иметь тип `bool`.

Присваивание

Оператор `=` применяется для присваивания изменяемым (`mut`) переменным, а также их полям или элементам. Но присваивание не так широко распространено, как в других языках, поскольку переменные в Rust по умолчанию неизменяемые.

Как было описано в главе 4, присваивание *передает владение* некопируемыми типами, а не выполняет их неявное копирование.

Поддерживается составное присваивание:

```
total += item.price;
```

Это эквивалентно выражению `total = total + item.price;`. Другие операторы тоже поддерживаются: `-=`, `*=` и т. д. Полный перечень приведен в табл. 6.1 в конце главы.

В отличие от C, Rust не поддерживает цепного присваивания: нельзя написать `a = b = 3`, чтобы присвоить значение `3` обеим переменным `a` и `b`. Присваивание встречается в Rust настолько редко, что вряд ли вам будет остро недоставать этого удобства.

В Rust нет операторов инкремента и декремента `++` и `--`.

Приведение типов

Для преобразования значения из одного типа в другой в Rust обычно необходимо явное приведение с помощью ключевого слова `as`:

```
let x = 17;              // x имеет тип i32
let index = x as usize;  // преобразовать в тип usize
```

Допустимо несколько видов приведения:

- число можно приводить из любого встроенного числового типа к любому другому.

  Приведение целого к другому целому типу всегда корректно определено. Приведение к более узкому типу приводит к усечению. Приведение целого со знаком к более широкому типу производится с распространением знака, а целого без знака – с дополнением нулями и т. д. Короче, никаких сюрпризов.

Однако на момент написания книги приведение большого числа с плавающей точкой к целому типу, слишком узкому для его представления, может закончиться неопределенным поведением и привести к краху даже в безопасном Rust. Эта ошибка компилятора, описанная по адресу *github.com/rust-lang/rust/issues/10184*;

- значения типа `bool`, `char` и типа перечисления, аналогичного C, можно приводить к любому целому типу (перечисления рассматриваются в главе 10). Приведение в обратном направлении не разрешено, поскольку типы `bool`, `char` и `enum` налагают на значения ограничения, которые пришлось бы проверять на этапе выполнения. Так, приведение `u16` к типу `char` запрещено, потому что некоторым значениям типа `u16` values, например `0xd800`, соответствуют суррогатные кодовые позиции в Юникоде, а значит, им нельзя сопоставить допустимое значение типа `char`. Существует стандартный метод `std::char::from_u32()`, который производит проверку во время выполнения и возвращает значение типа `Option<char>`, но, по правде сказать, необходимость в таком преобразовании возникает редко. Обычно мы преобразуем целые строки или потоки, а алгоритмы для работы с Юникод-текстами зачастую нетривиальны, и лучше их оставить библиотекам.
  Но в виде исключения тип `u8` можно привести к типу `char`, поскольку всем целым числам от 0 до 255 соответствуют допустимые кодовые позиции, которые можно сохранить в типе `char`;
- разрешены также некоторые приведения небезопасных указательных типов. Подробнее об этом написано в главе 21.

Мы сказали, что для преобразования типов *обычно* необходимо приведение. Но несколько преобразований ссылочных типов настолько просто, что язык выполняет их и без приведения. Тривиальный пример – преобразование изменяемой ссылки в неизменяемую.

Но есть и несколько других важных автоматических преобразований:

- значения типа `&String` автоматически преобразуются в тип `&str` без явного приведения;
- значения типа `&Vec<i32>` автоматически преобразуются в тип `&[i32]`;
- значения типа `&Box<Chessboard>` автоматически преобразуются в тип `&Chessboard`.

Все эти преобразования называются *Deref-преобразованиями*, поскольку применяются к типам, реализующим встроенную характеристику `Deref`. Цель Deref-преобразований – сделать так, чтобы типы интеллектуальных указателей, например `Box`, вели себя настолько похоже на указываемые значения, насколько это возможно. Благодаря `Deref` использование типа `Box<Chessboard>` мало чем отличается от использования простого типа `Chessboard`.

Пользовательские типы тоже могут реализовывать характеристику `Deref`. Если вам захочется написать свой тип интеллектуального указателя, обратитесь к разделу «Deref и DerefMut» главы 13.

Замыкания

В Rust имеются *замыкания* – значения, напоминающие функции, но не столь тяжеловесные. Замыкание обычно состоит из списка аргументов, заключенного между вертикальными черточками, за которым следует выражение:

```
let is_even = |x| x % 2 == 0;
```

Rust выводит типы аргументов и возвращаемого значения, но при желании их можно указывать и явно, как в функции. Если тип возвращаемого значения указан, то тело замыкания должно быть блоком, чтобы синтаксис был не слишком запутанным:

```
let is_even = |x: u64| -> bool x % 2 == 0;       // ошибка
let is_even = |x: u64| -> bool { x % 2 == 0 };   // правильно
```

Синтаксис вызова замыкания такой же, как вызова функции:

```
assert_eq!(is_even(14), true);
```

Замыкания – одна из самых восхитительных возможностей Rust, о которой можно говорить еще долго. Мы вернемся к ним в главе 14.

Приоритеты и ассоциативность

В табл. 6.1 приведена сводка синтаксиса выражений в Rust. Операторы перечислены в порядке приоритета, от наибольшего к наименьшему. (Как и в большинстве языков программирования, в Rust у каждого оператора имеется приоритет, благодаря чему определяется порядок операций в выражении, содержащем несколько соседствующих операторов. Например, в выражении `limit < 2 * broom.size + 1` оператор `.` имеет наивысший приоритет, поэтому первой операцией будет доступ к полю.)

Таблица 6.1. Выражения

Тип выражения	Пример	Характеристика
Литеральный массив	`[1, 2, 3]`	
Литеральный массив с повторением	`[0; 50]`	
Кортеж	`(6, "crullers")`	
Группировка	`(2 + 2)`	
Блок	`{ f(); g() }`	

Выражения управления потоком	`if ok { f() }`
	`if ok { 1 } else { 0 }`
	`if let Some(x) = f() { x } else { 0 }`
	`match x { None => 0, _ => 1 }`
	`for v in e { f(v); } while ok { ok = f(); }`
	`while let Some(x) = it.next() { f(x); }`
	`loop { next_event(); }`
	`break`
	`continue`
	`return 0`
		`std::iter::IntoIterator`

Вызов макроса	`println!("ok")`	
Путь	`std::f64::consts::PI`	
Литеральная структура	`Point {x: 0, y: 0}`	
Доступ к полю кортежа	`pair.0`	`Deref, DerefMut`
Доступ к полю структуры	`point.x`	`Deref, DerefMut`
Вызов метода	`point.translate(50, 50)`	`Deref, DerefMut`

Вызов функции	`stdin()`	`Fn(Arg0, ...) -> T,`
		`FnMut(Arg0, ...) -> T,`
		`FnOnce(Arg0, ...) -> T`

Окончание табл. 6.1

Тип выражения	Пример	Характеристика
Индекс	`arr[0]`	`Index`, `IndexMut` `Deref`, `DerefMut`
Проверка ошибки	`create_dir("tmp")?`	
Логическое/поразрядное НЕ	`!ok`	`Not`
Изменение знака	`-num`	`Neg`
Разыменование	`*ptr`	`Deref`, `DerefMut`
Заимствование	`&val`	
Приведение типа	`x as u32`	
Умножение	`n * 2`	`Mul`
Деление	`n / 2`	`Div`
Остаток	`n % 2`	`Rem`
Сложение	`n + 1`	`Add`
Вычитание	`n - 1`	`Sub`
Сдвиг влево	`n << 1`	`Shl`
Сдвиг вправо	`n >> 1`	`Shr`
Поразрядное И	`n & 1`	`BitAnd`
Поразрядное исключительное ИЛИ	`n ^ 1`	`BitXor`
Поразрядное ИЛИ	`n | 1`	`BitOr`
Меньше	`n < 1`	`std::cmp::PartialOrd`
Меньше или равно	`n <= 1`	`std::cmp::PartialOrd`
Больше	`n > 1`	`std::cmp::PartialOrd`
Больше или равно	`n >= 1`	`std::cmp::PartialOrd`
Равно	`n == 1`	`std::cmp::PartialEq`
Не равно	`n != 1`	`std::cmp::PartialEq`
Логическое И	`x.ok && y.ok`	
Логическое ИЛИ	`x.ok || backup.ok`	
Диапазон	`start .. stop`	
Присваивание	`x = val`	

Составное присваивание	`x *= 1`
	`x /= 1`
	`x %= 1`
	`x += 1`
	`x -= 1`
	`x <<= 1`
	`x >>= 1`
	`x &= 1`
	`x ^= 1`
	`x |= 1`
		`MulAssign`
		`DivAssign`
		`RemAssign`
		`AddAssign`
		`SubAssign`
		`ShlAssign`
		`ShrAssign`
		`BitAndAssign`
		`BitXorAssign`
		`BitOrAssign`

Замыкание	`|x, y| x + y`	

Все операторы, которые имеет смысл сцеплять, левоассоциативны. Это означает, что в цепочке операций вида `a - b - c` подразумевается расстановка скобок `(a - b) - c`, а не `a - (b - c)`. Состав операторов, допускающих такое сцепление, не вызывает удивления:

```
*   /   %   +   -   <<   >>   &   ^   |   &&   ||   as
```

Операторы сравнения, присваивания и диапазона (`..`) сцеплять нельзя.

Что дальше

Выражения – это то, что составляет «исполняемый код». Это та часть Rust-программы, которая транслируется в машинные команды. И все же это лишь малая часть языка в целом.

То же можно сказать о большинстве языков программирования. Основная задача программы – выполняться, но эта задача не единственная. Программы должны общаться между собой. Они должны быть тестопригодными. Они должны оставаться организованными и гибкими, чтобы было возможно развитие. Программы должны быть интероперабельны с кодом и службами, написанными другими коллективами. И даже для того чтобы просто выполняться, программы на таком статически типизированном языке, как Rust, нуждаются в дополнительных средствах организации данных, помимо кортежей и массивов.

Поэтому в нескольких последующих главах мы будем говорить о средствах, служащих этим целям: модулях и крейтах, привносящих организацию в саму программу, а также о структурах и перечислениях, преследующих ту же цель применительно к данным.

Но сначала посвятим несколько страниц важной теме: что делать, когда дела идут не так, как должно.

Глава 7

Обработка ошибок

> Я знал, что если проживу достаточно долго, что-то в этом роде должно было случиться.
>
> *— Эпитафия, которую Джордж Бернард Шоу написал для себя*

Обработка ошибок в Rust настолько отлична от привычной, что заслуживает отдельной короткой главы. Никаких трудных идей здесь нет, просто они могут показаться вам необычными. В этой главе рассматриваются два вида обработки ошибок в Rust: паника и значения типа `Result`.

Обычные ошибки обрабатываются с помощью типа `Result`. Как правило, они вызваны причинами, которые программа не контролирует: ошибочные входные данные, пропадание сети или проблемы с правами. То, что такие ситуации возникают, – не наша вина, даже безошибочная программа время от времени сталкивается с ними. Большая часть этой главы посвящена именно таким ошибкам. Но сначала мы рассмотрим панику, потому что она проще.

Паника – это ошибка другого рода – такая, которая *никогда не должна происходить*.

Паника

Программа паникует, встретив нечто настолько ужасное, что можно заподозрить дефект в самой программе, например:

- доступ к элементу за границами массива;
- деление целого числа на нуль;
- вызов метода `.unwrap()` для значения типа `Option`, оказавшегося равным `None`;
- ошибочное утверждение.

Существует также макрос `panic!()` на случай, если программа обнаружит, что делает что-то неправильное, и потому должна сама поднять панику. Этот макрос принимает необязательные аргументы в стиле `println!()` для построения сообщения об ошибке.

Общее у этих условий то, что все они – будем говорить откровенно – допущены по вине программиста. Есть хорошее правило: «Без паники». Но все мы делаем ошибки. А если ошибка, которой не должно было быть, все-таки случается, что тогда? На такой случай Rust предоставляет нам выбор: раскрутить стек в случае возникновения паники или снять процесс. По умолчанию подразумевается раскрутка стека.

Раскрутка стека

Когда пираты делят награбленную добычу, капитан получает половину, а рядовые члены команды делят вторую половину поровну. (Пираты терпеть не могут дроби, поэтому, когда добыча не делится нацело, результат округляется, и остаток достается корабельному попугаю.)

```
fn pirate_share(total: u64, crew_size: usize) -> u64 {
    let half = total / 2;
    half / crew_size as u64
}
```

Веками все шло прекрасно, пока в один прекрасный день не оказалось, что после набега выжил один лишь капитан. Если этой функции передать аргумент `crew_size`, равный нулю, то она попытается разделить на нуль. В C++ это неопределенное поведение, а в Rust программа паникует, что приводит к следующим событиям:

- На терминал выводится сообщение об ошибке.

  ```
  thread 'main' panicked at 'attempt to divide by zero', pirates.rs:3780
  note: Run with `RUST_BACKTRACE=1` for a backtrace.
  ```

  Если установлена переменная среды `RUST_BACKTRACE`, как рекомендуется в сообщении, то Rust дополнительно распечатает содержимое стека.
- Производится раскрутка стека – очень похоже на то, что происходит при обработке исключения в C++.

  Все временные значения, локальные переменные и аргументы текущей функции уничтожаются в порядке, обратном созданию. Уничтожение означает очистку: память, выделенная используемым строкам и векторам, освобождается, открытые файлы закрываются и т. д. Вызываются также пользовательские методы `drop` (см. раздел «Уничтожение» главы 13). В случае нашей функции `pirate_share()` очищать нечего.

  Произведя очистку для вызова текущей функции, мы переходим к вызвавшей ее функции и уничтожаем ее локальные переменные и аргумент. То же самое делается для функции, вызвавшей ее, и так далее вверх по стеку.
- Наконец, поток завершается. Если паника возникла в главном потоке, то вместе с ним завершается весь процесс (с ненулевым кодом выхода).

Пожалуй, «паника» – не вполне подходящее название для этой в высшей степени упорядоченной процедуры. Паника – это не крах. И не неопределенное поведение. Она больше напоминает исключение `RuntimeException` в Java или `std::logic_error` в C++. Поведение определено корректно, только вот случаться оно не должно.

Паника безопасна. Она не нарушает никаких правил Rust, касающихся безопасности. Даже если паника случилась в середине метода из стандартной библиотеки, она никогда не оставляет за собой висячих указателей или наполовину инициализированного значения в памяти. Идея в том, что Rust перехватывает некорректный доступ к массиву или что там еще, *до того* как произошла катастрофа. Продолжать работу было бы небезопасно, поэтому Rust раскручивает стек. Однако другие части процесса могут работать дальше.

Паника ограничена одним потоком. Один поток может запаниковать, тогда как остальные продолжат заниматься своими делами. В главе 19 мы покажем, как ро-

дительский поток может обнаружить, что дочерний поток паникует, и корректно обработать ошибку.

Существует также способ *перехватить* раскрутку стека, позволив потоку выжить и продолжить работу. Это делает функция `std::panic::catch_unwind()` из стандартной библиотеки. Мы не будем здесь рассматривать, как она используется, но это именно тот механизм, который применяется в тестовой обвязке Rust для восстановления после неверного утверждения в тесте. (Она также может понадобиться при написании Rust-кода, который может вызываться из C или C++, потому что раскрутка частей стека, созданных кодом на других языках, – неопределенное поведение; см. главу 21).

В идеале хотелось бы иметь не содержащий ошибок код, который никогда не паникует. Но нет в мире совершенства. Вы можете использовать потоки и функцию `catch_unwind()` для обработки паники, сделав свою программу более стабильной. Однако есть важная тонкость – так можно перехватить только панику, которая раскручивает стек. Но не каждая паника ведет себя подобным образом.

Снятие процесса

Раскрутка стека – поведение паники по умолчанию, но есть два случая, когда Rust не пытается раскрутить стек.

Если метод `.drop()` повторно паникует, когда Rust пытается выполнить очистку после первой паники, то ошибка считается фатальной. Rust прекращает раскрутку и снимает процесс целиком.

Кроме того, поведение паники можно настраивать. Если откомпилировать программу с флагом `-C panic=abort`, то к немедленному завершению процесса приводит уже *первая* паника. (В этом режиме Rust может не знать, как раскручивать стек, поэтому размер откомпилированного кода уменьшается.)

На этом мы завершаем обсуждение паники в Rust. Тут мало что можно добавить, потому что обычный код на Rust не обязан обрабатывать панику. Даже если вы пользуетесь потоками или функцией `catch_unwind()`, весь код обработки паники, скорее всего, будет сосредоточен в немногих местах. Неразумно ожидать, что каждая функция в программе предвидит ошибки в собственном коде и справляется с ними. А ошибки, вызванные иными факторами, – это совсем другая история.

Тип Result

В Rust нет исключений. Вместо этого функции, которые могут завершиться неудачно, возвращают значения такого вида:

```
fn get_weather(location: LatLng) -> Result<WeatherReport, io::Error>
```

Тип `Result` означает возможность неудачного завершения. Функция `get_weather()` возвращает либо *успешный результат* `Ok(weather)`, где `weather` – значение типа `WeatherReport`, либо *ошибочный результат* `Err(error_value)`, где `error_value` – значение типа `io::Error`, объясняющее, что случилось.

Rust требует, чтобы мы так или иначе обработали ошибку при вызове этой функции. Мы не можем получить `WeatherReport`, не сделав *что-то* с `Result`, и получим от компилятора предупреждение, если значение `Result` не используется.

В главе 10 мы увидим, как тип `Result` определен в стандартной библиотеке и как мы сами можем определить подобные типы. А пока пойдем по пути «поваренной книги» и посмотрим, как использовать значение типа `Result` для получения желаемого поведения в части обработки ошибок.

Обнаружение ошибок

Самый скрупулезный способ работы с `Result` был показан в главе 2: воспользоваться выражением `match`.

```
match get_weather(hometown) {
    Ok(report) => {
        display_weather(hometown, &report);
    }
    Err(err) => {
        println!("ошибка при запросе погоды: {}", err);
        schedule_weather_retry();
    }
}
```

Это эквивалент блока `try/catch` в других языках. Такой подход применяется, когда мы хотим обработать ошибки самостоятельно, а не передавать их вызывающей стороне.

Выражение `match` несколько многословно, поэтому тип `Result<T, E>` предоставляет ряд методов, полезных в частных случаях. В основе реализации каждого метода лежит выражение `match`. (Полный перечень методов типа `Result` см. в онлайновой документации. Ниже перечислены только наиболее употребительные.)

- **`result.is_ok()`** и **`result.is_err()`** возвращают значение типа `bool`, показывающее, содержит `result` успешный или ошибочный результат.
- **`result.ok()`** возвращает успешный результат, если таковой присутствует, в виде значения типа `Option<T>`. Если `result` содержит успешный результат, то это будет `Some(success_value)`, в противном случае `None` (а код ошибки отбрасывается).
- **`result.err()`** возвращает ошибочный результат, если таковой присутствует, в виде значения типа `Option<E>`.
- **`result.unwrap_or(fallback)`** возвращает успешный результат, если таковой содержится в `result`. В противном случае возвращается `fallback`, а код ошибки отбрасывается.

  ```
  // Сравнительно безопасный прогноз погоды для Южной Калифорнии.
  const THE_USUAL: WeatherReport = WeatherReport::Sunny(72);

  // Получить истинный прогноз погоды, если возможно.
  // Если нет, считать, что погода обычная.
  let report = get_weather(los_angeles).unwrap_or(THE_USUAL);
  display_weather(los_angeles, &report);
  ```

  Это удобная альтернатива методу `.ok()`, поскольку возвращается тип `T`, а не `Option<T>`. Разумеется, это работает, только если подходящее подменное значение существует.
- **`result.unwrap_or_else(fallback_fn)`** – то же самое, но вместо возврата подменного значения непосредственно вызывает переданную функцию или замы-

кание. Предназначен для случаев, когда было бы расточительно вычислять подменное значение, если мы не собираемся его использовать. `fallback_fn` вызывается только тогда, когда получен ошибочный результат.

```
let report =
    get_weather(hometown)
    .unwrap_or_else(|_err| vague_prediction(hometown));
```

Замыкания рассматриваются в главе 14.

- **`result.unwrap()`** также возвращает успешный результат, если он содержится в `result`. Если же `result` содержит ошибочный результат, то этот метод паникует. У этого метода есть полезные применения, о которых мы поговорим ниже.
- **`result.expect(message)`** – то же, что `.unwrap()`, но позволяет задать сообщение, которое печатается в случае паники.

  Наконец, последние два метода заимствуют ссылки на значение, хранящееся в `Result`.
- **`result.as_ref()`** преобразует `Result<T, E>` в `Result<&T, &E>`, заимствуя ссылку на успешный или ошибочный результат, содержащийся в имеющемся значении `result`.
- **`result.as_mut()`** – то же самое, но заимствуется изменяемая ссылка. Тип возвращаемого значения – `Result<&mut T, &mut E>`.

Одна из причин полезности этих двух методов состоит в том, что все остальные вышеупомянутые методы, кроме `.is_ok()` и `.is_err()`, *потребляют* результат `result`, к которому применяются. Иначе говоря, они принимают аргумент `self` по значению. Иногда желательно получить доступ к данным внутри `result`, не уничтожая их, именно для этого и предназначены методы `.as_ref()` и `.as_mut()`. Например, предположим, что мы хотели бы вызвать `result.ok()`, но так, чтобы `result` остался неприкосновенным. Тогда можно написать `result.as_ref().ok()`, при этом мы просто заимствуем `result` и возвращаем `Option<&T>` вместо `Option<T>`.

Псевдонимы типа Result

Иногда в документации по Rust встречаются места, где кажется, что тип ошибки в `Result` отсутствует:

```
fn remove_file(path: &Path) -> Result<()>
```

Это означает, что используется псевдоним типа `Result`.

Псевдоним типа – это своего рода сокращение имени типа. В модулях часто определяется псевдоним типа `Result`, чтобы не повторять раз за разом тип ошибки, который единообразно используется почти во всех функциях модуля. Например, в модуле `std::io` из стандартной библиотеки есть такая строчка:

```
pub type Result<T> = result::Result<T, Error>;
```

Здесь определяется открытый тип `std::io::Result<T>`. Это псевдоним типа `Result<T, E>`, но в качестве типа ошибки зашит тип `std::io::Error`.

На практике это означает, что, встретив предложение `use std::io;`, Rust понимает, что `io::Result<String>` – сокращенная запись `Result<String, io::Error>`.

Встретив в документации нечто вроде `Result<()>`, вы можете щелкнуть по идентификатору `Result`, чтобы увидеть, какой псевдоним типа используется, и узнать тип ошибки. На практике это обычно очевидно из контекста.

Печать информации об ошибках

Иногда единственный способ обработать ошибку – вывести информацию о ней на экран и продолжить работу. Один такой способ мы уже видели:

```
println!("ошибка при запросе погоды: {}", err);
```

В стандартной библиотеке определено несколько типов ошибок со скучными именами: `std::io::Error`, `std::fmt::Error`, `std::str::Utf8Error` и т. д. Все они реализуют общий интерфейс, характеристику `std::error::Error`, т. е. обладают перечисленными ниже свойствами.

- Допускают печать макросом `println!()`. Если ошибка печатается по формату `{}`, то выводится лишь краткое описание. Если же используется спецификатор формата `{:?}`, то выводится отладочное представление ошибки. Оно не столь дружелюбно к пользователю, зато содержит дополнительную техническую информацию.

```
// результат работы `println!("error: {}", err);`
error: failed to lookup address information: No address associated with
hostname

// результат работы `println!("error: {:?}", err);`
error: Error { repr: Custom(Custom { kind: Other, error: StringError(
"failed to lookup address information: No address associated with
hostname") }) }
```

- **`err.description()`** возвращает сообщение об ошибке в виде `&str`.
- **`err.cause()`** возвращает значение типа `Option<&Error>`: истинную ошибку, приведшую к `err`, если таковая имеется.

Например, из-за сетевой ошибки мог произойти сбой банковской транзакции, что, в свою очередь, привело к изъятию вашей яхты за неплатеж. Если метод `err.description()` возвратил строку `"boat was repossessed"`, то `err.cause()` мог бы вернуть ошибку, касающуюся непрошедшей транзакции, а ее метод `.cause()` – ошибку `io::Error`, содержащую детали сетевого сбоя, ставшего первопричиной всей неразберихи. Поскольку эта третья ошибка и есть корень всех зол, то ее метод `.cause()` возвращает `None`.

Поскольку в стандартную библиотеку входят только средства сравнительно низкого уровня, то для ошибок, возвращаемых библиотечными функциями, этот метод обычно возвращает `None`.

При печати значения ошибки ее причина не печатается. Чтобы напечатать всю имеющуюся информацию, воспользуйтесь такой функцией:

```
use std::error::Error;
use std::io::{Write, stderr};

/// Вывести сообщение об ошибке в `stderr`.
///
/// Если при построении сообщения об ошибке или его выводе
/// произойдет еще одна ошибка, она будет проигнорирована.
fn print_error(mut err: &Error) {
    let _ = writeln!(stderr(), "error: {}", err);
    while let Some(cause) = err.cause() {
        let _ = writeln!(stderr(), "caused by: {}", cause);
```

```
            err = cause;
        }
    }
```

Типы ошибок в стандартной библиотеке не включают трассу стека, но крейт `errorchain` позволяет без труда определить пользовательский тип ошибки, который запоминает трассу стека в момент создания экземпляра. Для этого используется крейт `backtrace`.

Распространение ошибок

Как правило, пытаясь сделать нечто такое, что может завершиться ошибкой, мы не хотим обнаруживать и обрабатывать эту ошибку немедленно. Писать предложение `match` на десять строк в любом месте, где может случиться неприятность, попросту нерационально.

Поэтому мы обычно предпочитаем, чтобы ошибку обработала вызывающая сторона, т. е. хотим, чтобы ошибки *распространялись* по стеку вызовов.

Для этой цели в Rust имеется оператор `?`. Его можно добавить в любое выражение, порождающее `Result`, например результат вызова функции:

```
let weather = get_weather(hometown)?;
```

Поведение `?` зависит от того, возвращает ли функция успешный или ошибочный результат.

- В случае успеха оператор разворачивает `Result` методом `unwrap`, чтобы получить находящееся внутри него значение. В данном случае типом переменной `weather` будет не `Result<WeatherReport, io::Error>`, а просто `WeatherReport`.
- В случае ошибки оператор сразу же возвращает управление из объемлющей функции, передавая ошибочный результат вверх по цепочке вызовов. Но чтобы этот механизм работал, оператор `?` следует использовать только в функциях, возвращающих значение типа `Result`.

Ничего магического в операторе `?` нет. То же самое можно сделать с помощью выражения `match`, только гораздо более многословно:

```
let weather = match get_weather(hometown) {
    Ok(success_value) => success_value,
    Err(err) => return Err(err)
};
```

Единственная разница между этим кодом и оператором `?` – некоторые тонкие моменты, касающиеся типов и их преобразований. Мы рассмотрим их в следующем разделе.

В старом коде можно встретить макрос `try!()`, который обычно использовался для распространения ошибок до появления оператора `?` в версии Rust 1.13.

```
let weather = try!(get_weather(hometown));
```

Этот макрос расширяется в показанное выше выражение `match`.

Легко можно забыть о том, насколько вездесущи ошибки в программе, особенно в коде, который взаимодействует с операционной системой. Иногда оператор `?` встречается чуть ли не в каждой строчке функции:

```
use std::fs;
use std::io;
use std::path::Path;

fn move_all(src: &Path, dst: &Path) -> io::Result<()> {
    for entry_result in src.read_dir()? {     // открытие каталога может быть неудачным
        let entry = entry_result?;            // и чтение тоже
        let dst_file = dst.join(entry.file_name());
        fs::rename(entry.path(), dst_file)?;  // и переименование может не получиться
    }
    Ok(())  // уф!
}
```

Работа с ошибками нескольких типов

Зачастую нам грозит сразу несколько неприятностей. Допустим, что мы просто читаем числа из текстового файла.

```
use std::io::{self, BufRead};

/// Читать целые числа из текстового файла.
/// В каждой строке файла должно быть одно число.
fn read_numbers(file: &mut BufRead) -> Result<Vec<i64>, io::Error> {
    let mut numbers = vec![];
    for line_result in file.lines() {
        let line = line_result?;      // чтение строки может завершиться ошибкой
        numbers.push(line.parse()?);  // разбор целого числа может быть неудачным
    }
    Ok(numbers)
}
```

Компилятор Rust выдает следующее сообщение об ошибке:

```
numbers.push(line.parse()?); // разбор целого числа может быть неудачным
             ^^^^^^^^^^^^^ the trait `std::convert::From<std::num::ParseIntError>`
                           is not implemented for `std::io::Error`
```

Термины, употребляемые в этом сообщении, обретут смысл, когда мы дойдем до главы 11, где рассматриваются характеристики. А пока просто заметим, что Rust ругается на невозможность преобразовать значение типа `std::num::ParseIntError` в тип `std::io::Error`.

Проблема в том, что операции чтения строки из файла и разбора целого числа могут давать ошибки разных типов. Функция `line_result` возвращает значение типа `Result<String, std::io::Error>`, а `line.parse()` – значение типа `Result<i64, std::num::ParseIntError>`. Тип значения, возвращаемого функцией `read_numbers()`, рассчитан только на ошибки типа `io::Error`. Rust пытается преобразовать тип `ParseIntError` в `io::Error`, но, поскольку такого преобразования не существует, мы получаем ошибку типизации.

Справиться с этим можно несколькими способами. Например, в крейте `image`, который мы использовали в главе 2 для создания файлов, содержащих изображения множества Мандельброта, определен свой тип ошибки `ImageError` и реализованы преобразования типа `io::Error` и нескольких других в `ImageError`. Если вы хотите пойти этим путем, то обратитесь к вышеупомянутому крейту `errorchain`,

который предназначен специально для определения хороших типов ошибок с помощью всего нескольких строчек кода.

Проще воспользоваться тем, что уже встроено в Rust. Все типы ошибок в стандартной библиотеке можно преобразовать в тип `Box<std::error::Error>`, представляющий «любую ошибку». Поэтому легкий способ обработать ошибки нескольких типов состоит в том, чтобы определить такие псевдонимы типов:

```
type GenError = Box<std::error::Error>;
type GenResult<T> = Result<T, GenError>;
```

Затем нужно изменить тип возвращаемого значения `read_numbers()` на `GenResult<Vec<i64>>`. После этого функция компилируется. Оператор `?` автоматически преобразует любой тип ошибки в `GenError` по мере необходимости.

Кстати говоря, оператор `?` производит это автоматическое преобразование с помощью стандартного метода, которым можете воспользоваться и вы. Чтобы преобразовать любую ошибку в тип `GenError`, вызовите метод `GenError::from()`.

```
let io_error = io::Error::new(          // создать экземпляр типа io::Error
    io::ErrorKind::Other, "timed out");
return Err(GenError::from(io_error));  // вручную преобразовать его в GenError
```

Характеристика `From` и ее метод `from()` подробно описаны в главе 13.

Недостаток подхода, основанного на `GenError`, – в том, что тип возвращаемого значения уже не содержит точную информацию о том, какие именно ошибки может ожидать вызывающая сторона. Она должна быть готова ко всему.

Если вы вызываете функцию, которая возвращает `GenResult`, и хотите обработать только ошибки одного вида, а остальные передать вызывающей стороне, то воспользуйтесь методом `error.downcast_ref::<ErrorType>()`. Он заимствует ссылку на ошибку, *только если* она принадлежит интересующему вас типу.

```
loop {
    match compile_project() {
        Ok(()) => return Ok(()),
        Err(err) => {
            if let Some(mse) = err.downcast_ref::<MissingSemicolonError>() {
                insert_semicolon_in_source_code(mse.file(), mse.line())?;
                continue;  // попробовать еще раз!
            }
            return Err(err);
        }
    }
}
```

Во многие языки встроены синтаксические средства для такого рода действий, но они редко оказываются востребованы. Разработчики Rust предпочли предоставить метод.

Ошибки, которых «не может быть»

Иногда мы точно *знаем*, что некоторая ошибка не может произойти. Предположим, к примеру, что мы пишем код для разбора конфигурационного файла, и в каком-то месте обнаруживаем, что дальше идет строка цифр:

```
if next_char.is_digit(10) {
    let start = current_index;
    current_index = skip_digits(&line, current_index);
    let digits = &line[start..current_index];
    ...
```

Мы хотим преобразовать эту строку цифр в число. Для этого существует стандартный метод:

```
let num = digits.parse::<u64>();
```

Но вот в чем проблема: метод `str.parse::<u64>()` возвращает не `u64`, а `Result`. Он может завершиться ошибкой, потому что не все строки содержат только цифры.

```
"bleen".parse::<u64>()  // ParseIntError: недопустимая цифра
```

Но мы-то знаем, что в данном случае строка `digits` состоит только из цифр. Что же делать?

Если наш код возвращает значение типа `GenResult`, то мы можем прибегнуть к оператору `?` и забыть о проблеме. В противном случае открывается мрачная перспектива: писать код для обработки ошибки, которой не может быть. Тогда оптимальный вариант – воспользоваться методом `.unwrap()` типа `Result`, о котором мы уже упоминали.

```
let num = digits.parse::<u64>().unwrap();
```

Он работает почти так же, как оператор `?`, но если невозможная ошибка все-таки произойдет, то возникнет паника.

На самом деле в данном случае наше оптимистичное предположение не оправдывается. Если входная строка цифр достаточно длинная, то число не поместится в тип `u64`.

```
"99999999999999999999".parse::<u64>()  // переполнение
```

Поэтому в данном случае использование метода `.unwrap()` было бы ошибкой. Некорректные входные данные – не основание для паники.

Но иногда все же возникают ситуации, когда `Result` никак не может содержать ошибочный результат. Например, в главе 18 мы познакомимся с характеристикой `Write`, определяющей ряд общих методов (`.write()` и прочие) для вывода текстовых и двоичных данных. Все эти методы возвращают значения типа `io::Result`, но если мы осуществляем вывод в вектор типа `Vec<u8>`, то они не могут завершиться ошибкой. В таких случаях допускается использовать методы `.unwrap()` или `.expect(message)` для обработки `Result`.

Эти методы также полезны, когда ошибка настолько странная или серьезная, что лучшим способом ее обработки оказывается паника.

```
fn print_file_age(filename: &Path, last_modified: SystemTime) {
    let age = last_modified.elapsed().expect("system clock drift");
    ...
}
```

Здесь метод `.elapsed()` может завершиться неудачно, только если системное время *раньше* времени создания файла. Это может случиться, если файл был создан недавно и системные часы были отведены назад во время работы программы.

В зависимости от того, как устроен код, может оказаться разумным поднять панику, а не пытаться обработать ошибку самостоятельно или передать вызывающей стороне.

Игнорирование ошибок

Иногда мы хотим вообще игнорировать ошибку. Так, в нашей функции `print_error()` необходимо было как-то обработать маловероятную ситуацию, когда в процессе печати сообщения возникает новая ошибка. Это может случиться, например, если поток `stderr` был соединен конвейером с другим процессом, а этот процесс завершился. Поскольку мы все равно тут ничего не можем сделать, то просто игнорируем ошибку, но компилятор Rust предупредит о неиспользованном значении `Result`:

```
writeln!(stderr(), "error: {}", err);  // предупреждение: результат не используется
```

Для подавления этого предупреждения применяют идиому `let _ = ...`:

```
let _ = writeln!(stderr(), "error: {}", err);  // ok, результат игнорируется
```

Обработка ошибок в main()

Почти всюду, где порождается `Result`, допустимо и даже правильно передать ошибку вызывающей стороне. Потому-то оператор `?` в Rust и занимает всего один символ. Как мы видели, в некоторых программах он используется во многих строчках подряд.

Но если ошибка распространяется достаточно далеко, то в конце концов она дойдет до функции `main()`, и тогда такой подход перестанет работать. В `main()` нельзя использовать оператор `?`, потому что она не возвращает значения типа `Result`.

```
fn main() {
    calculate_tides()?;  // ошибка: дальше передавать ошибку некуда
}
```

Для обработки ошибок в `main()` проще всего использовать метод `.expect()`.

```
fn main() {
    calculate_tides().expect("error");  // здесь распространение прекращается
}
```

Если функция `calculate_tides()` возвращает ошибочный результат, то метод `.expect()` паникует. Паника в главном потоке влечет за собой печать сообщения об ошибке и завершение программы с ненулевым кодом. В общем-то, такое поведение можно назвать желательным. В крохотных демонстрационных программах мы всегда так и поступаем.

Правда, само сообщение несколько смущает[1]:

```
$ tidecalc --planet mercury
thread 'main' panicked at 'error: "moon not found"', /buildslave/rust-buildbot/s
lave/nightly-dist-rustc-linux/build/src/libcore/result.rs:837
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

[1] Речь идет о программе расчета приливов. При запуске для Меркурия программа печатает сообщение «не найдена луна», что для планеты без спутников вполне разумно. – *Прим. перев.*

Сообщение об ошибке затерялось в шуме. Кроме того, совет установить `RUST_BACKTRACE=1` в данном случае неуместен. Хорошо было бы напечатать само сообщение об ошибке:

```
fn main() {
    if let Err(err) = calculate_tides() {
        print_error(&err);
        std::process::exit(1);
    }
}
```

Здесь мы используем выражение `if let` для печати только сообщения об ошибке в случае, когда `calculate_tides()` возвращает ошибочный результат. Подробнее о выражениях `if let` см. главу 10. Код функции `print_error` был приведен выше.

Вот теперь вывод аккуратный:

```
$ tidecalc --planet mercury
error: moon not found
```

Объявление пользовательского типа ошибки

Предположим, что мы пишем анализатор JSON и хотим ввести свой собственный тип ошибки. (Мы будем рассматривать пользовательские типы только через несколько глав. Но поскольку типы ошибок действительно нужны, мы решили немного забежать вперед.)

Минимальный код, который предстоит написать, выглядит примерно так:

```
// json/src/error.rs

#[derive(Debug, Clone)]
pub struct JsonError {
    pub message: String,
    pub line: usize,
    pub column: usize,
}
```

Эта структура будет именоваться `json::error::JsonError`, и если мы захотим вернуть ошибку такого типа, то напишем:

```
return Err(JsonError {
    message: "expected ']' at end of array".to_string(),
    line: current_line,
    column: current_column
});
```

Это работает. Но если мы захотим, чтобы наш тип ошибки вел себя как стандартные и не обманывал ожидания пользователей библиотеки, то придется еще немного потрудиться:

```
use std;
use std::fmt;

// Ошибки должны допускать распечатку.
impl fmt::Display for JsonError {
    fn fmt(&self, f: &mut fmt::Formatter) -> Result<(), fmt::Error> {
```

```
        write!(f, "{} ({}:{})", self.message, self.line, self.column)
    }
}

// Ошибки должны реализовывать характеристику std::error::Error.
impl std::error::Error for JsonError {
    fn description(&self) -> &str {
        &self.message
    }
}
```

Смысл ключевых слов `impl` и `self`, как и все остальное, будет разъяснен в последующих главах.

Почему именно тип Result?

Теперь мы знаем достаточно, чтобы понять, почему Rust предпочитает тип `Result` исключениям. Перечислим ключевые моменты проектирования.

- Rust требует, чтобы программист принял и отразил в коде то или иное решение в каждом месте, где может произойти ошибка. Это хорошо, потому что иначе обработкой ошибки можно было бы пренебречь, что неправильно;
- самое частое решение – дать ошибке распространиться вверх по стеку, и для его выражения достаточно одного символа `'?'`. Поэтому обработка ошибок не загромождает код, как в C и Go, но и незаметной тоже не остается: достаточно взглянуть на код, чтобы сразу найти все места, где ошибка распространяется;
- поскольку потенциальная возможность ошибки является частью типа, возвращаемого функцией, то сразу ясно, какие функции могут завершиться неудачно, а какие нет. Если определение функции изменяется, так что она может вернуть ошибку, то мы обязаны изменить тип возвращаемого ей значения, а тогда компилятор заставит изменить весь код, где эта функция используется;
- Rust проверяет, что значения типа `Result` используются, поэтому ошибке не удастся проскочить незаметно (типичный дефект в программах на C);
- поскольку `Result` – такой же тип данных, как и любой другой, хранить успешные и ошибочные результаты можно в одной коллекции. Это упрощает моделирование частичного успеха. Например, если мы пишем программу, загружающую миллионы записей из текстового файла, и необходимо предусмотреть типичную ситуацию, когда большинство записей правильно, но некоторые ошибочны, то мы можем представить ее вектором значений типа `Result` в памяти.

За все это приходится расплачиваться тем, что на обдумывание и реализацию механизма обработки ошибок в Rust приходится тратить больше времени, чем в других языках. Как и во многих других вопросах, отношение Rust к обработке ошибок несколько строже, чем вы привыкли. Но в системном программировании это окупается.

Глава 8

Крейты и модули

> Одно замечание на тему Rust: и у системных программистов могут быть симпатичные штучки.
>
> — *Роберт О'Каллахэн о cargo и crate.io*[1]

Допустим, вы пишете программу, которая моделирует рост папоротников, начиная с отдельной клетки. Ваша программа, как и папоротник, поначалу очень простая, весь ее код помещается в один файл – это только спора идеи. По мере роста она начинает обретать внутреннюю структуру. Разные ее части служат разным целям. Она ветвится, ей нужно уже несколько файлов. Возможно, она даже захватит дерево каталогов. Со временем она может стать важной частью целой программной экосистемы.

В этой главе мы рассмотрим средства Rust, предназначенные для организации программ: крейты и модули. Мы также обсудим широкий спектр вопросов, которые естественно возникают по мере роста проекта, в т. ч. вопрос о документировании и тестировании кода на Rust, о подавлении нежелательных предупреждений компилятора, об использовании Cargo для управления зависимостями и версиями, о публикации библиотек с открытым исходным кодом на сайте crates.io и т. д.

Крейты

Rust-программы состоят из *крейтов* (crate). Каждый крейт представляет собой проект Rust: весь исходный код одной библиотеки или исполняемого файла вместе с сопутствующими тестами, примерами, инструментальными средствами, конфигурационными файлами и всем прочим. Для программы моделирования папоротников могут понадобиться сторонние библиотеки трехмерной графики, биоинформатики, параллельных вычислений и т. п. Все они распространяются в виде крейтов.

---

[1] robert.ocallahan.org.

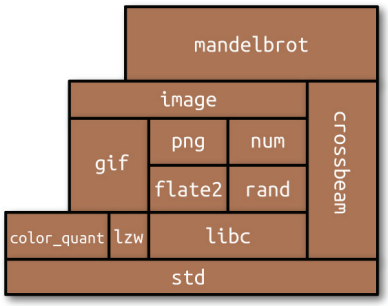


Рис. 8.1 ❖ Крейт и его зависимости

Чтобы понять, что такое крейты и как они работают совместно, проще всего выполнить команду `cargo build` с флагом `-verbose`, которая строит проект с зависимостями. Мы поступили так для программы построения множества Мандельброта из главы 2. Вот что получилось.

```
$ cd mandelbrot
$ cargo clean # delete previously compiled code
$ cargo build --verbose
    Updating registry `https://github.com/rust-lang/crates.io-index`
Downloading image v0.6.1
Downloading crossbeam v0.2.9
Downloading gif v0.7.0
Downloading png v0.4.2

... (скачивание и компиляции еще кучи крейтов)

  Compiling png v0.4.2
    Running `rustc .../png-0.4.2/src/lib.rs
         --crate-name png
         --crate-type lib
         --extern num=.../libnum-a2e6e61627ca7fe5.rlib
         --extern inflate=.../libinflate-331fc425bf167339.rlib
         --extern flate2=.../libflate2-857dff75f2932d8a.rlib
         ...`
  Compiling image v0.6.1
    Running `rustc .../image-0.6.1/./src/lib.rs
         --crate-name image
         --crate-type lib
         --extern png=.../libpng-16c24f58491a5853.rlib
         ...`
  Compiling mandelbrot v0.1.0 (file://.../mandelbrot)
    Running `rustc src/main.rs
         --crate-name mandelbrot
         --crate-type bin
         --extern crossbeam=.../libcrossbeam-ba292320058da7df.rlib
         --extern image=.../libimage-254ec48c8f0684f2.rlib
         ...`
$
```

Для удобства чтения мы изменили формат команд `rustc` и удалили различные параметры компилятора, не имеющие отношения к обсуждаемой теме, заменив их многоточием.

Напомним, что к моменту завершения работы над программой построения множества Мандельброта файл `main.rs` содержал три объявления `extern crate`:

```
extern crate num;
extern crate image;
extern crate crossbeam;
```

Эти строки говорят Rust, что `num`, `image` и `crossbeam` – внешние библиотеки, а не часть самой программы.

Кроме того, в файле `Cargo.toml` мы указали, какие версии крейтов нам нужны:

```
[dependencies]
num = "0.1.27"
image = "0.6.1"
crossbeam = "0.2.8"
```

Здесь словом «dependencies» (зависимости) просто обозначены другие крейты, используемые в проекте: код, от которого зависит наша программа. Мы нашли эти крейты на сайте crates.io, где сообщество Rust складирует крейты с открытым исходным кодом. Например, о библиотеке `image` мы узнали, зайдя на сайт crates.io и поискав библиотеку для обработки изображений. На странице каждого крейта на сайте crates.io имеются ссылки на документацию и исходный код, а также строка конфигурации вида `image = "0.6.1"`, которую можно скопировать и вставить в файл `Cargo.toml`. Выше указаны номера последних версий трех пакетов на момент написания программы.

Из протокола работы программы Cargo видно, как эта информация используется. После запуска команды `cargo build` Cargo сначала загружает исходный код указанных версий крейтов с сайта crates.io. Затем она читает файлы `Cargo.toml` этих крейтов, загружает *их* зависимости и продолжает этот процесс рекурсивно. Так, в составе исходного кода версии 0.6.1 крейта `image` есть такой файл `Cargo.toml`:

```
[dependencies]
byteorder = "0.4.0"
num = "0.1.27"
enum_primitive = "0.1.0"
glob = "0.2.10"
```

Видя это, Cargo понимает, что, прежде чем использовать `image`, необходимо скачать и эти крейты тоже. Позже мы увидим, как сказать Cargo, что исходный код нужно брать из репозитория git или из локальной файловой системы, а не с сайта crates.io.

Получив в свое распоряжение весь исходный код, Cargo приступает к компиляции крейтов. Она запускает компилятор `rustc` для каждого крейта в графе зависимостей проекта. При компиляции библиотек Cargo задает параметр `--crate-type lib`, который говорит `rustc`, что не нужно искать функцию `main()`, а надо создать `rlib`-файл, содержащий откомпилированный код в форме, которую последующие команды `rustc` могут использовать в качестве входных данных. При компиляции программы Cargo задает параметр `--crate-type bin`, в результате чего получается исполняемый файл для целевой платформы, например `mandelbrot.exe` для Windows.

Каждой команде `rustc` Cargo передает параметры `-extern` с указанием имен библиотек, используемых данным крейтом. Так, видя строку `extern crate crossbeam;`, компилятор знает, где на диске искать откомпилированный крейт. Компилятору Rust нужен доступ к `rlib`-файлам, потому что они содержат откомпилированный код библиотек. Rust статически компонует этот код с конечным исполняемым файлом. Кроме того, `rlib`-файл содержит информацию о типах, так что Rust может проверить, что те возможности библиотеки, которые применяются в нашем коде, действительно существуют в крейте и используются нами корректно. Также в этом файле находятся открытые встраиваемые функции, универсальные функции и макросы, т. е. средства, которые невозможно полностью транслировать в машинный код, пока Rust не увидит, как они используются.

Команда `cargo build` поддерживает многочисленные параметры, большинство которых выходит за рамки этой книги, но один мы все же упомянем: `cargo build --release` порождает оптимизированную выпускную сборку. Выпускные сборки работают быстрее, но компилируются дольше, в них нет проверок переполнения целых типов, опущены утверждения `debug_assert!()`, а трассы стека, генерируемые ими в случае паники, вообще говоря, менее надежны.

Сборочные профили

В файл `Cargo.toml` можно поместить несколько конфигурационных параметров, влияющих на то, какие командные строки `rustc` будет генерировать `cargo`.

Командная строка	Секция файла Cargo.toml
`cargo build`	`[profile.debug]`
`cargo build --release`	`[profile.release]`
`cargo test`	`[profile.test]`

Обычно умолчаний достаточно, исключением является ситуация, когда мы хотим использовать профилировщик – программу, которая измеряет, на что тратится процессорное время в программе. Чтобы получить от него наиболее надежные данные, необходимо включить как оптимизацию (обычно она включается только для выпускных сборок), так и отладочные символы (обычно включаются только для отладочных сборок). Чтобы включить то и другое, добавьте в `Cargo.toml` такие строки:

```
[profile.release]
debug = true  # включить отладочные символы в выпускные сборки
```

Параметр `debug` управляет заданием флага `-g` команды `rustc`. При такой конфигурации команда `cargo build -release` будет генерировать двоичный файл с отладочными символами. На параметры оптимизации это не влияет.

В документации по Cargo (*doc.crates.io*) описано много других настраиваемых параметров.

Модули

Модули – это пространства имен в Rust. Они являются контейнерами функций, типов, констант и вообще всего, что составляет программу или библиотеку. Если

цель крейтов – совместное использование кода в разных проектах, то модули направлены на организацию кода *внутри* одного проекта. Выглядят они так:

```
mod spores {
    use cells::Cell;

    /// Клетка, произведенная взрослым папоротником. Распространяется ветром, что
    /// является частью жизненного цикла папоротника. Из споры вырастает зароcток,
    /// отдельный организм шириной до 5 мм, который порождает зиготу, вырастающую
    /// в новый папоротник (пол растения определить сложно).
    pub struct Spore {
    ...
    }

    /// Моделирует производство споры путем мейоза.
    pub fn produce_spore(factory: &mut Sporangium) -> Spore {
        ...
    }

    /// Смешивает гены для мейоза (часть интерфазы).
    fn recombine(parent: &mut Cell) {
        ...
    }

    ...
}
```

Модуль состоит из *артикулов* (item) – именованных конструкций, каковыми являются структура `Spore` и две показанные выше функции. Ключевое слово `pub` делает артикул открытым, т. е. доступным извне самого модуля. Все, что не помечено как `pub`, является закрытым.

```
let s = spores::produce_spore(&mut factory);  // правильно
spores::recombine(&mut cell);  // ошибка: `recombine` - закрытый артикул
```

Модули могут быть вложенными, и не редкость модуль, содержащий только набор подмодулей и больше ничего:

```
mod plant_structures {
    pub mod roots {
        ...
    }
    pub mod stems {
        ...
    }
    pub mod leaves {
        ...
    }
}
```

Действуя таким образом, мы могли бы написать полную программу, содержащую уйму кода и целую иерархию модулей – и всё в одном исходном файле. Но это крайне неудобно, поэтому существует альтернатива.

Модули в отдельных файлах

Модуль можно записать и в таком виде:

```
mod spores;
```

Выше мы включили тело модуля `spores`, заключенное в фигурные скобки. А сейчас мы говорим компилятору Rust, что модуль `spores` находится в отдельном файле с именем `spores.rs`:

```
// spores.rs

/// Клетка, произведенная взрослым папоротником...
pub struct Spore {
    ...
}

/// Моделирует производство споры путем мейоза.
pub fn produce_spore(factory: &mut Sporangium) -> Spore {
    ...
}

/// Смешивает гены для мейоза (часть интерфазы).
fn recombine(parent: &mut Cell) {
    ...
}
```

Файл `spores.rs` содержит только артикулы, составляющие модуль. Чтобы объявить его модулем, не нужен никакой трафаретный код.

Местоположение кода – *единственная* разница между этим модулем `spores` и его версией, показанной в предыдущем разделе. Правила, определяющие, что является открытым, а что закрытым, в точности одинаковы. И Rust никогда не компилирует модули отдельно, даже если они находятся в отдельных файлах; в процессе сборки крейта перекомпилируются все его модули.

У модуля может быть свой каталог. Видя предложение `mod spores;`, Rust проверяет наличие двух файлов: `spores.rs` и `spores/mod.rs`. Если не существует ни того, ни другого или существуют сразу оба, то компилятор считает это ошибкой. В данном примере мы остановились на файле `spores.rs`, потому что у модуля `spores` нет подмодулей. Но рассмотрим написанный выше модуль `plant_structures`. Если бы мы захотели разнести этот модуль и три его подмодуля по отдельным файлам, то проект выглядел бы так:

```
fern_sim/
├── Cargo.toml
└── src/
    ├── main.rs
    ├── spores.rs
    └── plant_structures/
        ├── mod.rs
        ├── leaves.rs
        ├── roots.rs
        └── stems.rs
```

В файле `main.rs` объявляется модуль `plant_structures`:

```
pub mod plant_structures;
```

Тем самым мы заставляем Rust загрузить файл `plant_structures/mod.rs`, в котором объявлены все три подмодуля:

```
// в plant_structures/mod.rs
pub mod roots;
```

```
pub mod stems;
pub mod leaves;
```

Содержимое этих трех модулей хранится в отдельных файлах: `leaves.rs`, `roots.rs` и `stems.rs`, находящихся вместе с `mod.rs` в каталоге `plant_structures`.

Пути и импорт

Оператор `::` служит для доступа к артикулам модуля. Код в любом месте проекта может обратиться к средству из стандартной библиотеки по *абсолютному пути*:

```
if s1 > s2 {
    ::std::mem::swap(&mut s1, &mut s2);
}
```

Имя функции, `::std::mem::swap`, является абсолютным путем, потому что начинается двумя знаками двоеточия. Путь `::std` относится к верхнеуровневому модулю стандартной библиотеки. `::std::mem` – подмодуль стандартной библиотеки, а `::std::mem::swap` – открытая функция в этом подмодуле.

Можно было бы всю дорогу придерживаться такого стиля программирования, выписывая `::std::f64::consts::PI` и `::std::collections::HashMap::new` всякий раз, как понадобится окружность или словарь, но подобный код утомительно писать и трудно читать. Альтернатива – *импортировать* артикулы в модули, где они используются:

```
use std::mem;

if s1 > s2 {
    mem::swap(&mut s1, &mut s2);
}
```

Объявление `use` делает `mem` локальным псевдонимом для `::std::mem` на всем протяжении объемлющего блока или модуля. Пути в объявлении `use` автоматически считаются абсолютными, поэтому в префиксе `::` нет нужды.

Можно было бы написать `use std::mem::swap;`, тогда мы импортировали бы только функцию `swap`, а не весь модуль `mem`. Однако первый вариант обычно считается оптимальным стилем: импортировать типы, характеристики и модули (как в `std::mem`), а затем использовать относительные пути для доступа к функциям, константам и прочим членам внутри модуля.

Можно импортировать сразу несколько имен:

```
use std::collections::{HashMap, HashSet};  // импортируются оба имени
use std::io::prelude::*;  // импортируется всё
```

Это не более чем сокращенная запись всех отдельных предложений импорта:

```
use std::collections::HashMap;
use std::collections::HashSet;

// все открытые артикулы из std::io::prelude:
use std::io::prelude::Read;
use std::io::prelude::Write;
use std::io::prelude::BufRead;
use std::io::prelude::Seek;
```

Модули *не* наследуют автоматически имена из родительских модулей. Предположим, к примеру, что файл `proteins/mod.rs` содержит такие строки:

```
// proteins/mod.rs
pub enum AminoAcid { ... }
pub mod synthesis;
```

Тогда код в файле `synthesis.rs` не видит тип `AminoAcid`:

```
// proteins/synthesis.rs
pub fn synthesize(seq: &[AminoAcid])  // ошибка: не удается найти тип `AminoAcid`
    ...
```

Каждый модуль начинает с чистого листа и должен импортировать имена, которые в нем используются:

```
// proteins/synthesis.rs
use super::AminoAcid;  // явно импортировать из родителя
pub fn synthesize(seq: &[AminoAcid])  // все хорошо
    ...
```

Ключевое слово `super` имеет специальный смысл в предложениях импорта: это псевдоним родительского модуля. Аналогично `self` – псевдоним текущего модуля.

```
// в proteins/mod.rs

// импортировать из подмодуля
use self::synthesis::synthesize;

// импортировать имена из перечисления, чтобы можно было
// обозначать лизин просто `Lys`, а не `AminoAcid::Lys`
use self::AminoAcid::*;
```

Хотя пути в предложениях импорта по умолчанию считаются абсолютными, ключевые слова `self` и `super` позволяют отменить это соглашение и импортировать по относительному пути.

(Пример с `AminoAcid` – это, конечно, отступление от вышеупомянутого стилистического правила об импорте типов, характеристик и модулей. Но если наша программа включает длинные последовательности аминокислот, то такое отступление оправдано шестым правилом Оруэлла: «Лучше нарушить любое из этих правил, чем сказать что-то, варварски неуместное».)

Подмодули могут обращаться к закрытым артикулам родительских модулей, но должны импортировать каждый по имени. Предложение `use super::*;` импортирует только артикулы, помеченные ключевым словом `pub`.

Модули – не то же самое, что файлы, но существует естественная аналогия между модулями и файлами и каталогами в файловой системе Unix. Ключевое слово `use` создает псевдонимы точно так же, как команда `ln` создает ссылки. Пути, как и имена файлов, могут быть абсолютными и относительными. Ключевые слова `self` и `super` – аналоги специальных каталогов `.` и `..`. А `extern crate` включает в проект корневой модуль еще одного крейта – почти то же, что монтирование файловой системы.

Стандартная прелюдия

Чуть выше мы сказали, что каждый модуль начинает жизнь «с чистого листа» в том, что касается импортированных имен. Однако этот лист не *совсем* чистый.

Во-первых, стандартная библиотека `std` автоматически компонуется с каждым проектом, как если бы в файле `lib.rs` или `main.rs` находилось невидимое объявление

```
extern crate std;
```

Во-вторых, некоторые особенно часто используемые имена, например `Vec` и `Result`, включаются в *стандартную прелюдию* и автоматически импортируются. Rust ведет себя так, как будто каждый модуль, включая и корневой, начинается следующим предложением импорта:

```
use std::prelude::v1::*;
```

Стандартная прелюдия содержит несколько десятков наиболее употребительных характеристик и типов. Но она *не* содержит `std`. Поэтому если ваш модуль обращается к `std`, то необходимо импортировать его явно:

```
use std;
```

Но обычно разумнее импортировать именно ту конкретную часть `std`, которой вы пользуетесь.

В главе 2 мы мельком упоминали, что библиотеки иногда предоставляют модули с именем `prelude`. Но `std::prelude::v1` – единственная прелюдия, которая импортируется автоматически. Присваивание модулю имени `prelude` – не более чем соглашение, сообщающее пользователям о том, что этот модуль предполагается импортировать целиком с помощью `*`.

Артикулы – строительные блоки в Rust

Модуль состоит из *артикулов*. Их несколько видов, и список артикулов – это, по сути дела, перечень основных возможностей языка.

- **Функции.** Мы уже встречали их неоднократно.
- **Типы.** Пользовательские типы вводятся ключевыми словами `struct`, `enum` и `trait`. Каждому из них мы посвятим отдельную главу; простая структура выглядит так:

  ```
  pub struct Fern {
      pub roots: RootSet,
      pub stems: StemSet
  }
  ```

  Поля структуры, даже закрытые, доступны из любого места модуля, в котором эта структура объявлена. Вне модуля доступны только открытые поля. Как выясняется, контроль доступа на уровне модуля, а не класса, как в Java или C++, весьма полезен для проектирования программного обеспечения. При этом сокращается потребность в трафаретных методах чтения и установки полей и практически отпадает необходимость в чем-то вроде объявлений друзей (`friend`), как в C++. В одном модуле можно определить несколько работающих совместно типов, например `frond::LeafMap` и `frond::LeafMapIter`,

которые имеют доступ к закрытым полям друг друга, но при этом детали реализации остаются скрыты для остальной части программы.

- **Псевдонимы типов.** Как мы видели, ключевое слово `type` используется так же, как `typedef` в C++, для объявления нового имени существующего типа:

```
type Table = HashMap<String, Vec<String>>;
```

Здесь объявляемый тип `Table` – сокращенная запись типа `HashMap`, параметризованного конкретными типами.

```
fn show(table: &Table) {
    ...
}
```

- **Блоки impl.** Для присоединения методов к типам служат блоки `impl`:

```
impl Cell {
    pub fn distance_from_origin(&self) -> f64 {
        f64::hypot(self.x, self.y)
    }
}
```

Этот синтаксис объясняется в главе 9. Блок `impl` нельзя пометить ключевым словом `pub`, но можно так пометить отдельные методы, чтобы сделать их видимыми вне текущего модуля.
Закрытые методы, как и закрытые поля структуры, видны в любом месте модуля, в котором объявлены.

- **Константы.** Ключевое слово `const` определяет константу. Синтаксис такой же, как в `let`, с тем отличием, что константа может быть помечена как `pub`, а тип обязателен. Кроме того, имена констант обычно записываются заглавными буквами:

```
pub const ROOM_TEMPERATURE: f64 = 20.0; // градусы по Цельсию
```

Ключевое слово `static` определяет статический артикул, который мало чем отличается от константы:

```
pub static ROOM_TEMPERATURE: f64 = 68.0; // градусы по Фаренгейту
```

Константы немного напоминают директиву `#define` в C++: на этапе компиляции значение вставляется во все места кода, где встречается. Статические переменные инициализируются до начала выполнения программы и существуют до ее завершения. Используйте константы для определения магических чисел и строк в коде, а статические переменные – для более объемных данных, а также в тех случаях, когда нужно позаимствовать ссылку на значение константы.
Константа не может быть помечена ключевым словом `mut`. Статические переменные – могут, но, как обсуждалось в главе 5, в Rust нет возможности применить правила исключительного доступа к изменяемым статическим переменным. Поэтому они принципиально не являются потокобезопасными и в безопасном коде употребляться не могут:

```
static mut PACKETS_SERVED: usize = 0;

println!("{} served", PACKETS_SERVED); // ошибка: используется изменяемая
                                        // статическая переменная
```

Rust противится использованию изменяемого глобального состояния. Альтернативы обсуждаются в разделе «Глобальные переменные» главы 19.

- **Модули.** Мы уже говорили о них. Как было сказано, модуль может содержать подмодули, которые могут быть открытыми или закрытыми, как и любой другой именованный артикул.
- **Объявления импорта.** Объявления `use` и `extern crate` тоже являются артикулами. И хотя это просто псевдонимы, их можно делать открытыми:

```
// в plant_structures/mod.rs
...
pub use self::leaves::Leaf;
pub use self::roots::Root;
```

Это означает, что `Leaf` и `Root` – открытые артикулы модуля `plant_structures`, но по-прежнему остаются просто псевдонимами `plant_structures::leaves::Leaf` и `plant_structures::roots::Root`.
Стандартная прелюдия представляет собой просто последовательность таких открытых объявлений импорта.

- **Блоки extern.** Объявляют наборы функций, написанных на другом языке (обычно C или C++), чтобы их можно было вызывать из Rust-программы. Блоки `extern` рассматриваются в главе 21.

Rust предупреждает, если объявленный артикул нигде не используется:

```
warning: function is never used: `is_square`
  --> src/crates_unused_items.rs:23:9
   |
23 | /         pub fn is_square(root: &Root) -> bool {
24 | |             root.cross_section_shape().is_square()
25 | |         }
   | |_________^
   |
```

Это предупреждение озадачивает, потому что у него две совсем разные потенциальные причины. Быть может, сама эта функция в настоящий момент является «мертвым» кодом, а быть может, предполагалось, что она будет использоваться в других крейтах. В последнем случае следует пометить ее *и все объемлющие модули* как открытые.

Превращение программы в библиотеку

По мере развития модели папоротников мы решаем, что одной программы недостаточно. Допустим, что раньше существовала только программа, которая запускалась из командной строки и сохраняла результаты в файле. А теперь мы хотим написать другие программы для научного анализа сохраненных результатов, трехмерного отображения растущих растений в реальном масштабе времени, построения фотореалистичных изображений и т. д. Во всех этих программах должен использоваться общий код моделирования папоротников. Нужна библиотека.

Первый шаг – выделить из существующего проекта две части: библиотечный крейт, содержащий весь общий код, и исполняемый файл, содержащий только тот код, который нужен для имеющейся командной программы.

Чтобы показать, как это делается, возьмем сильно упрощенный пример:

```
struct Fern {
    size: f64,
    growth_rate: f64
}

impl Fern {
    /// Моделирует суточный рост папоротника.
    fn grow(&mut self) {
        self.size *= 1.0 + self.growth_rate;
    }
}

/// Прогоняет модель папоротника на протяжении нескольких суток.
fn run_simulation(fern: &mut Fern, days: usize) {
    for _ in 0 .. days {
        fern.grow();
    }
}

fn main() {
    let mut fern = Fern {
        size: 1.0,
        growth_rate: 0.001
    };
    run_simulation(&mut fern, 1000);
    println!("конечный размер папоротника: {}", fern.size);
}
```

Будем предполагать, что у этой программы тривиальный файл `Cargo.toml`:

```
[package]
name = "fern_sim"
version = "0.1.0"
authors = ["You <you@example.com>"]
```

Превратить эту программу в библиотеку легко. Ниже приведен перечень необходимых шагов:

- переименовать файл `src/main.rs` в `src/lib.rs`;
- добавить ключевое слово `pub` к тем артикулам в файле `src/lib.rs`, которые станут открытыми возможностями библиотеки;
- переместить функцию `main` в какой-нибудь временный файл (скоро мы к ней вернемся).

В результате получится такой файл `src/lib.rs`:

```
pub struct Fern {
    pub size: f64,
    pub growth_rate: f64
}

impl Fern {
    /// Моделирует суточный рост папоротника.
    pub fn grow(&mut self) {
        self.size *= 1.0 + self.growth_rate;
```

```
    }
}

/// Прогоняет модель папоротника на протяжении нескольких суток.
pub fn run_simulation(fern: &mut Fern, days: usize) {
    for _ in 0 .. days {
        fern.grow();
    }
}
```

Отметим, что файл `Cargo.toml` изменять не пришлось. Это потому, что наш минималистский файл `Cargo.toml` оставляет поведение Cargo по умолчанию, а это значит, что команда `cargo build` просматривает файлы в каталоге с исходным кодом и сама определяет, что собирать. Встретив файл `src/lib.rs`, она понимает, что нужно собирать библиотеку.

Код в файле `src/lib.rs` образует *корневой модуль* библиотеки. Другие крейты, пользующиеся этой библиотекой, могут обращаться только к открытым артикулам из корневого модуля.

Каталог src/bin

Вернуть командную программу `fern_sim` в работоспособное состояние также нетрудно: в Cargo встроена поддержка небольших программ, находящихся в той же кодовой базе, что и библиотека.

На самом деле и сама программа Cargo написана так же. Большую ее часть составляет библиотека. А команда `cargo`, которую мы все время используем в книге, – это лишь тонкая обертка, обращающаяся к библиотеке, когда нужно сделать что-то сложное. И библиотека, и командная программа находятся в одном репозитории исходного кода *https://github.com/rust-lang/cargo*.

Свою программу и библиотеку мы тоже можем поместить в одну кодовую базу. Скопируйте следующий код в файл `src/bin/efern.rs`:

```
extern crate fern_sim;
use fern_sim::{Fern, run_simulation};

fn main() {
    let mut fern = Fern {
        size: 1.0,
        growth_rate: 0.001
    };
    run_simulation(&mut fern, 1000);

    println!("конечный размер папоротника: {}", fern.size);
}
```

Функция `main` – как раз та, что раньше мы отложили в сторонку. Мы добавили объявление `extern crate`, поскольку эта программа будет пользоваться библиотечным крейтом `fern_sim`. И, кроме того, мы импортируем из библиотеки артикулы `Fern` и `run_simulation`.

Поскольку мы поместили этот файл в каталог `src/bin`, Cargo будет компилировать и библиотеку `fern_sim`, и эту программу при следующем запуске `cargo build`. Чтобы запустить программу `efern`, выполним команду `cargo run --bin efern`. И вот

что получится, если в обоих случаях задать еще флаг -verbose, чтобы видеть, какие команды запускает Cargo:

```
$ cargo build --verbose
    Compiling fern_sim v0.1.0 (file:///.../fern_sim)
      Running `rustc src/lib.rs --crate-name fern_sim --crate-type lib ...`
      Running `rustc src/bin/efern.rs --crate-name efern --crate-type bin ...`
$ cargo run --bin efern --verbose
        Fresh fern_sim v0.1.0 (file:///.../fern_sim)
      Running `target/debug/efern`
Конечный размер папоротника: 2.7169239322355985
```

Мы по-прежнему не внесли никаких изменений в файл Cargo.toml, поскольку по умолчанию Cargo решает, что нужно сделать, глядя на исходные файлы. Она считает все rs-файлы в каталоге src/bin дополнительными программами, подлежащими сборке.

Но, разумеется, теперь, когда fern_sim стала библиотекой, у нас появился другой вариант действий. Мы могли бы поместить эту программу в отдельный проект, находящийся в своем каталоге с собственным файлом Cargo.toml, в котором fern_sim сделана зависимостью:

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

Быть может, именно так мы и поступили бы для других программ моделирования папоротников, которые еще предстоит написать. Ну а для такой простенькой программы, как efern, достаточно и каталога src/bin.

Атрибуты

Любой артикул в Rust-программе может быть снабжен *атрибутами*. Атрибуты в Rust – это универсальное средство для передачи разнообразных инструкций и рекомендаций компилятору. Допустим, к примеру, что компилятор выдает такое предупреждение:

```
libgit2.rs: warning: type `git_revspec` should have a camel case name
such as `GitRevspec`, #[warn(non_camel_case_types)] on by default
```

Но вы выбрали такое имя не без причины и хотите, чтобы Rust оставил вас в покое. Чтобы подавить предупреждение, добавьте атрибут #[allow] к типу:

```
#[allow(non_camel_case_types)]
pub struct git_revspec {
    ...
}
```

Условная компиляция – еще одна возможность, активируемая атрибутом, на этот раз #[cfg]:

```
// Включать этот модуль в проект, только если производится сборка для Android.
#[cfg(target_os = "android")]
mod mobile;
```

Полностью синтаксис атрибута `#[cfg]` описан в справочном руководстве по Rust (http://doc.rus-lang.org), а наиболее употребительные параметры перечислены в табл. 8.1.

Таблица 8.1. Наиболее употребительные параметры атрибута #[cfg]

Параметр #[cfg(...)]	Разрешен при условии
`test`	Разрешены тесты (компиляция командой `cargo test` или `rustc --test`)
`debug_assertions`	Разрешены отладочные утверждения (обычно в неоптимизированных сборках)
`unix`	Компиляция для Unix, включая macOS
`windows`	Компиляция для Windows
`target_pointer_width = "64"`	Компиляция для 64-разрядной платформы. Другое возможное значение `"32"`
`target_arch = "x86_64"`	Компиляция для архитектуры x86-64. Другие возможные значения: `"x86"`, `"arm"`, `"aarch64"`, `"powerpc"`, `"powerpc64"`, `"mips"`
`target_os = "macos"`	Компиляция для macOS. Другие возможные значения: `"windows"`, `"ios"`, `"android"`, `"linux"`, `"openbsd"`, `"netbsd"`, `"dragonfly"`, `"bitrig"`
`feature = "robots"`	Включено определенное пользователем средство `"robots"` (компиляция командой `cargo build --feature robots` или `rustc --cfg feature='"robots"'`). Средства объявляются в секции `[features]` файла `Cargo.toml`
`not(A)`	Условие `A` не удовлетворяется. Чтобы предоставить две разные реализации функции, пометьте одну атрибутом `#[cfg(X)]`, а другую – атрибутом `#[cfg(not(X))]`
`all(A,B)`	Удовлетворяются оба условия `A` и `B` (эквивалент `&&`)
`any(A,B)`	Удовлетворяется хотя бы одно из условий `A` и `B` (эквивалент `||`)

Иногда мы хотим самостоятельно поуправлять расширением встраиваемой функции, хотя обычно такую оптимизацию лучше оставить компилятору. Для этой цели служит атрибут `#[inline]`:

```
/// Изменить уровни ионов и т. д. в двух соседних клетках
/// вследствие осмоса между ними.
#[inline]
fn do_osmosis(c1: &mut Cell, c2: &mut Cell) {
    ...
}
```

Существует одна ситуация, когда встраивание *не* производится, если отсутствует атрибут `#[inline]`. Если функция (или метод), определенная в одном крейте, вызывается из другого крейта, то Rust не производит встраивания, если только эта функция (или метод) не является универсальной (параметризована типами) или явно не помечена атрибутом `#[inline]`.

Во всех остальных случаях компилятор воспринимает атрибут `#[inline]` как рекомендацию. Rust поддерживает также более настойчивую форму `#[inline(always)]`, которая предлагает встраивать функцию в каждом месте ее вызова, и форму `#[inline(never)]`, предлагающую никогда не встраивать функцию.

Некоторые атрибуты, например `#[cfg]` и `#[allow]`, можно присоединять ко всему модулю, и тогда они применяются ко всем содержащимся в нем артикулам. Другие, например `#[test]` и `#[inline]`, разрешается присоединять только к отдельным артикулам. Как и следует ожидать от универсального средства, каждый атрибут создается по отдельному лекалу и поддерживает собственный набор аргументов. В справочном руководстве по Rust подробно описаны все поддерживаемые атрибуты.

Чтобы присоединить атрибут ко всему крейту, поместите его в начало файла `main.rs` или `lib.rs`, раньше всех остальных артикулов, и вместо префикса `#` напишите `#!`.

```
// libgit2_sys/lib.rs
#![allow(non_camel_case_types)]

pub struct git_revspec {
    ...
}

pub struct git_error {
    ...
}
```

Префикс `#!` сообщает Rust, что атрибут относится к объемлющему артикулу, а не к тому, что находится непосредственно после него; в данном случае атрибут `#![allow]` присоединен ко всему крейту `libgit2_sys`, а не только к структуре `struct git_revspec`.

Префикс `#!` можно использовать также внутри функций, структур и т. п., но обычно он ставится в начале файла, чтобы присоединить атрибут ко всему модулю или крейту. Некоторым атрибутам всегда предшествует `#!`, потому что их разрешено применять только ко всему крейту.

Например, атрибут `#![feature]` позволяет включить *нестабильные* средства языка Rust и библиотек, т. е. экспериментальные средства, которые могут содержать ошибки и, возможно, будут изменены или исключены в будущем. Так, на момент написания книги в Rust присутствовала экспериментальная поддержка 128-разрядных целых типов `i128` и `u128`, но, поскольку они экспериментальные, для работы с ними нужно было (1) установить собираемую ночью версию Rust и (2) явно объявить, что они используются в крейте:

```
#![feature(i128_type)]

fn main() {
    // Сделай за меня домашнее задание, Rust!
    println!("{}", 92040938115958335 89_u128 * 19973810893143440503_u128);
}
```

Иногда по прошествии некоторого времени разработчики Rust принимают решение *стабилизировать* экспериментальное средство, и тогда оно становится стандартной частью языка. После этого атрибут `#![feature]` становится лишним, и Rust выдает предупреждение о том, что его нужно убрать.

Тесты и документация

В разделе «Написание и выполнение автономных тестов» главы 2 мы видели, что в Rust встроен простой каркас автономного тестирования. Тесты – это обычные функции, помеченные атрибутом `#[test]`.

```
#[test]
fn math_works() {
    let x: i32 = 1;
    assert!(x.is_positive());
    assert_eq!(x + 1, 2);
}
```

Команда `cargo test` прогоняет все имеющиеся в проекте тесты.

```
$ cargo test
   Compiling math_test v0.1.0 (file:///.../math_test)
     Running target/release/math_test-e31ed91ae51ebf22

running 1 test
test math_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

(Команда может также выдавать информацию о «doc-тестах», о которых мы поговорим чуть позже.)

Этот механизм работает одинаково для исполняемого файла и библиотеки. Можно прогнать только часть тестов, передав Cargo дополнительные аргументы. Так, команда `cargo test math` прогоняет все тесты, в названии которых встречается слово `math`.

В тестах обычно используются макросы `assert!` и `assert_eq!` из стандартной библиотеки Rust. Утверждение `assert!(expr)` выполняется, если выражение `expr` истинно. В противном случае оно паникует, и тест завершается неудачно. Утверждение `assert_eq!(v1, v2)` аналогично `assert!(v1 == v2)` с тем отличием, что если оно не выполнено, то в сообщении об ошибке приводятся оба значения.

Эти макросы можно использовать и в обычном коде для проверки инвариантов, однако отметим, что макросы `assert!` и `assert_eq!` остаются даже в выпускных сборках. Если вы хотите, чтобы утверждения проверялись только в отладочных сборках, то пользуйтесь вместо них макросами `debug_assert!` и `debug_assert_eq!`.

Для тестирования обработки ошибок добавьте к тесту атрибут `#[should_panic]`:

```
/// Этот тест проходит, только если деление на нуль вызывает панику,
/// как описано в предыдущей главе.
#[test]
#[should_panic(expected="divide by zero")]
fn test_divide_by_zero_error() {
    1 / 0;  // should panic!
}
```

Функции с атрибутом `#[test]` компилируются условно. При выполнении команды `cargo test` собирается программа, включающая все тесты и тестовую обвязку. Обычная команда `cargo build` или `cargo build --release` пропускает тестовый код. Это означает, что автономные тесты могут находиться в непосредственной близости от тестируемого кода и при необходимости получать доступ к деталям реализации, а платить за это на этапе выполнения не придется. Однако компилятор при этом может выдавать предупреждения, например:

```
fn roughly_equal(a: f64, b: f64) -> bool {
    (a - b).abs() < 1e-6
}

#[test]
fn trig_works() {
    use std::f64::consts::PI;
    assert!(roughly_equal(PI.sin(), 0.0));
}
```

В тестовой сборке все нормально. Но в нетестовой сборке функция `roughly_equal` не используется, и Rust ругается:

```
$ cargo build
   Compiling math_test v0.1.0 (file:///.../math_test)
warning: function is never used: `roughly_equal`
 --> src/crates_unused_testing_function.rs:7:1
  |
7 | / fn roughly_equal(a: f64, b: f64) -> bool {
8 | |     (a - b).abs() < 1e-6
9 | | }
  | |_^
  |
  = note: #[warn(dead_code)] on by default
```

Поэтому принято соглашение: если тесты настолько объемны, что для них требуется поддерживающий код, то все это добро выносится в отдельный модуль `tests` и весь модуль объявляется тестовым с помощью атрибута `#[cfg]`:

```
#[cfg(test)]    // этот модуль включается только при тестировании
mod tests {
    fn roughly_equal(a: f64, b: f64) -> bool {
        (a - b).abs() < 1e-6
    }

    #[test]
    fn trig_works() {
        use std::f64::consts::PI;
        assert!(roughly_equal(PI.sin(), 0.0));
    }
}
```

Тестовая обвязка Rust запускает одновременно несколько тестов в разных потоках – полезный побочный эффект потокобезопасности написанного на Rust кода. (Чтобы отключить многопоточное выполнение, либо запускайте тесты по одному – `cargo test` *`testname`*`;`, либо присвойте переменной среды `RUST_TEST_THREADS` значение `1`.) Это означает, что, строго говоря, программа построения множества Мандельброта, написанная в главе 2, была не второй многопоточной программой в той главе, а третьей! Первой была команда `cargo test`, запускавшаяся в разделе «Написание и выполнение автономных тестов».

Интеграционные тесты

Программа моделирования папоротников продолжает развиваться. Вы решили перенести всю основную функциональность в библиотеку, которой могут пользоваться различные исполняемые файлы. Хорошо было бы иметь тесты, которые компонуются с библиотекой, как настоящие конечные пользователи, указывая `fern_sim.rlib` в качестве внешнего крейта. Кроме того, у нас уже имеются тесты, которые загружают сохраненные результаты моделирования из двоичного файла, и хранить эти огромные тестовые файлы в каталоге `src` совершенно ни к чему. Обе проблемы помогают решить интеграционные тесты.

Интеграционные тесты – это `rs`-файлы, которые хранятся в каталоге `tests` на том же уровне проекта, что каталог `src`. При выполнении команды `cargo test` каж-

дый интеграционный тест компилируется в отдельный независимый крейт, компонуемый с библиотекой и тестовой обвязкой Rust. Приведем пример.

```
// tests/unfurl.rs - свернутые листья папоротника распускаются при свете

extern crate fern_sim;
use fern_sim::Terrarium;
use std::time::Duration;

#[test]
fn test_fiddlehead_unfurling() {
    let mut world = Terrarium::load("tests/unfurl_files/fiddlehead.tm");
    assert!(world.fern(0).is_furled());
    let one_hour = Duration::from_secs(60 * 60);
    world.apply_sunlight(one_hour);
    assert!(world.fern(0).is_fully_unfurled());
}
```

Отметим, что интеграционный тест включает объявление `extern crate`, поскольку пользуется крейтом `fern_sim` как библиотекой. Идея интеграционных тестов в том и состоит, что они видят крейт извне, как видел бы его пользователь. Они тестируют открытый API крейта.

Команда `cargo test` исполняет как автономные, так и интеграционные тесты. Если нужно выполнить только интеграционные тесты в одном конкретном файле, например `tests/unfurl.rs`, то запустите команду `cargo test --test unfurl`.

Документация

Команда `cargo doc` создает HTML-документацию по библиотеке:

```
$ cargo doc --no-deps --open
 Documenting fern_sim v0.1.0 (file:///.../fern_sim)
```

Флаг `--no-deps` говорит, что нужно сгенерировать документацию только для крейта `fern_sim`, а не для всех крейтов, от которых он зависит.

Флаг `--open` говорит Cargo, что сгенерированную документацию следует открыть в браузере. Результат показан на рис. 8.2. Cargo сохраняет файлы документации в каталоге `target/doc`. Адрес начальной страницы `target/doc/fern_sim/index.html`.

В документацию включаются библиотечные средства, помеченные ключевым словом `pub`, а также сопровождающие их *документирующие комментарии*. Примеры таких комментариев мы уже встречали в этой главе. Выглядят они как обычные комментарии:

```
/// Моделирует образование споры путем мейоза.
pub fn produce_spore(factory: &mut Sporangium) -> Spore {
    ...
}
```

Но, видя комментарий, начинающийся тремя знаками /, Rust трактует их как атрибут `#[doc]`, т. е. приведенный выше код эквивалентен такому:

```
#[doc = "Моделирует образование споры путем мейоза."]
pub fn produce_spore(factory: &mut Sporangium) -> Spore {
    ...
}
```

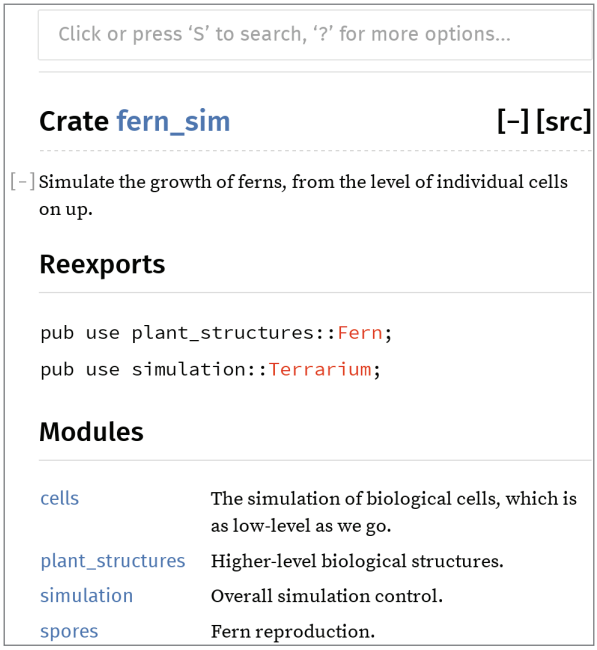


Рис. 8.2 ❖ Пример документации, сгенерированной командой `rustdoc`

При компиляции или тестировании библиотеки эти атрибуты игнорируются, а при генерации документации документирующие комментарии для открытых средств включаются в конечный файл.

Аналогично комментарии, начинающиеся префиксом `//!`, трактуются как атрибуты `#![doc]` и присоединяются к объемлющему артикулу, обычно модулю или крейту. Например, файл `fern_sim/src/lib.rs` мог бы начинаться так:

```
//! Моделирование роста папоротника, начиная
//! с единственной клетки.
```

Сам текст документирующего комментария записывается на языке Markdown, сокращенной нотации простого HTML-форматирования. Звездочками выделяется `*курсив*` и `**полужирный шрифт**`, пустая строка считается концом параграфа и т. д. Но HTML тоже годится: все HTML-теги в документирующих комментариях буквально переносятся в документацию.

`` `Обратные апострофы` `` служат для выделения кода внутри текста. В результирующем документе такие фрагменты будут отображаться моноширинным шрифтом. Более крупные куски кода выделяются красной строкой шириной четыре пробела.

```
/// Блок кода в документирующем комментарии:
///
///     if everything().works() {
///         println!("ok");
///     }
```

Допускаются также «огражденные» блоки кода, как принято в языке Markdown. Результат будет точно таким же.

```
/// Тот же код, но записанный по-другому:
///
/// ```
/// if everything().works() {
///     println!("ok");
/// }
/// ```
```

Вне зависимости от формата, если в документирующий комментарий включен блок кода, то происходит любопытная вещь: Rust автоматически делает из него тест.

Doc-тесты

В процессе тестирования библиотечного крейта Rust проверяет, что весь код, встречающийся в документации, действительно работает. Для этого каждый блок кода в документирующем комментарии компилируется в отдельный исполняемый крейт, компонуется с библиотекой и выполняется.

Ниже приведен изолированный пример doc-теста. Создайте новый проект командой `cargo new ranges` и поместите следующий текст в файл `ranges/src/lib.rs`:

```
use std::ops::Range;

/// Возвращает true, если два диапазона пересекаются.
///
///     assert_eq!(ranges::overlap(0..7, 3..10), true);
///     assert_eq!(ranges::overlap(1..5, 101..105), false);
///
/// Если хотя бы один диапазон пуст, диапазоны считаются непересекающимися.
///
///     assert_eq!(ranges::overlap(0..0, 0..10), false);
///
pub fn overlap(r1: Range<usize>, r2: Range<usize>) -> bool {
    r1.start < r1.end && r2.start < r2.end &&
        r1.start < r2.end && r2.start < r1.end
}
```

На рис. 8.3 показано, как выглядят в документации два небольших блока кода в документирующем комментарии.

Эти блоки также становятся отдельными тестами.

```
$ cargo test
   Compiling ranges v0.1.0 (file:///.../ranges)
...
   Doc-tests ranges

running 2 tests
test overlap_0 ... ok
test overlap_1 ... ok
test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured
```

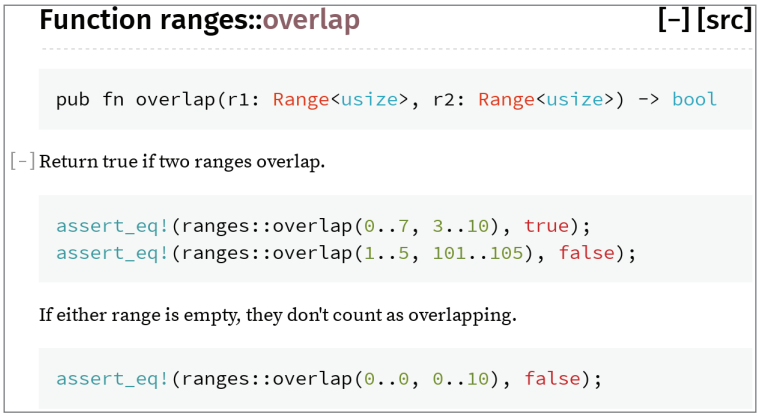


Рис. 8.3 ❖ Документация, демонстрирующая doc-тесты

Запустив Cargo с флагом `--verbose`, мы увидим, что оба теста прогоняются командой `rustdoc --test`. Программа `rustdoc` сохраняет каждый пример код в отдельном файле, добавляя в него несколько трафаретных строк, и порождает две программы. Вот первая из них:

```
extern crate ranges;
fn main() {
    assert_eq!(ranges::overlap(0..7, 3..10), true);
    assert_eq!(ranges::overlap(1..5, 101..105), false);
}
```

А вот вторая:

```
extern crate ranges;
fn main() {
    assert_eq!(ranges::overlap(0..0, 0..10), false);
}
```

Тесты проходят, если эти программы успешно компилируются и выполняются.

В обоих примерах кода есть утверждения, но это только потому, что в данном случае утверждения составляют хорошую документацию. Идея doc-тестов не в том, чтобы включить все вообще тесты в документацию, а в том, чтобы сделать документацию более качественной. Для этого Rust проверяет, что все включенные в документацию примеры кода действительно компилируются и работают.

Очень часто даже для минимального работоспособного примера нужны дополнительные детали, например предложения импорта или код инициализации. Без них код не будет компилироваться, но они и не настолько важны, чтобы включать их в документацию. Чтобы скрыть строку в примере кода, добавьте в ее начало знак # и пробел:

```
/// Выполняет моделирование в течение заданного времени
/// в предположении, что светит солнце.
///
///     # use fern_sim::Terrarium;
///     # use std::time::Duration;
```

```
///     # let mut tm = Terrarium::new();
///     tm.apply_sunlight(Duration::from_secs(60));
///
pub fn apply_sunlight(&mut self, time: Duration) {
    ...
}
```

Иногда полезно привести в документации полный пример программы, включая функцию `main` и объявление `extern crate`. Очевидно, что если такие куски кода уже присутствуют, то мы не хотим, чтобы Rustdoc добавила их еще раз, поскольку тогда программа не будет компилироваться. Поэтому Rustdoc считает, что любой блок кода, содержащий строку `fn main`, является законченной программой и ничего не добавляет от себя.

Для некоторых блоков кода тестирование можно отключить. Чтобы попросить Rust откомпилировать пример, но не запускать его, пользуйтесь огражденным кодовым блоком с аннотацией `no_run`:

```
/// Закачать все локальные террариумы в онлайновую галерею.
///
/// ```no_run
/// let mut session = fern_sim::connect();
/// session.upload_all();
/// ```
pub fn upload_all(&mut self) {
    ...
}
```

Если даже не ожидается, что код должен компилироваться, укажите `ignore` вместо `no_run`. Если блок кода написан вообще не на Rust, укажите язык, например `c++` или `sh`, а если это просто текст, то `text`. Программа `rustdoc` не знает названий сотен языков программирования, просто любая неопознанная аннотация интерпретируется как признак того, что блок кода написан не на Rust. Это подавляет подсветку кода в документации, а также генерацию doc-тестов.

Задание зависимостей

Мы уже видели один способ сказать Cargo, где взять исходный код крейтов, от которых зависит проект: по номеру версии.

```
image = "0.6.1"
```

Есть еще несколько способов описать зависимости, а также сообщить довольно тонкие инструкции о том, какие версии использовать. Уделим этому вопросу несколько страниц.

Прежде всего может статься, что нужные нам зависимости вообще не опубликованы на сайте `crates.io`. Тогда можно, например, указать URL-адрес репозитория Git и номер ревизии:

```
image = { git = "https://github.com/PistonDevelopers/image.git", rev = "528f19c" }
```

Для данного конкретного крейта исходный код открыт и выложен на Github, но ничто не мешает указать закрытый репозиторий Git, размещенный в корпоративной сети. Как показано выше, можно указать конкретную ревизию (`rev`), метку

(tag) или ветвь (branch) – все это способы идентификации версии кода, которую нужно извлечь из Git.

Альтернатива – указать каталог, содержащий исходный код крейта:

```
image = { path = "vendor/image" }
```

Это удобно, когда вся команда пользуется одним репозиторием, содержащим исходный код нескольких крейтов или, быть может, весь граф зависимостей. В каждом крейте его зависимости можно задать относительными путями.

Такой уровень контроля над зависимостями открывает большие возможности. Если вы со временем решите, что какой-то крейт с открытым исходным кодом вам не совсем подходит, то сможете тривиально разветвить его: просто нажмите кнопку «Fork» на Github и измените одну строку в файле Cargo.toml. При следующем запуске cargo build послушно подцепит ваш вариант вместо официальной версии.

Версии

Встретив в файле Cargo.toml нечто вроде image = "0.6.1", Cargo интерпретирует эту строку не буквально, а берет последнюю версию image, считающуюся совместимой с 0.6.1.

Правила совместимости основаны на спецификации семантического версионирования (http://semver.org).

- Версии с номером, начинающимся с 0.0, настолько сырые, что Cargo не предполагает для них совместимости с какой-то другой версией.
- Версия с номером, начинающимся с 0.x, где x отлично от нуля, считается совместимой с другими версиями в серии 0.x. Мы указали версию image 0.6.1, но Cargo взяла бы версию 0.6.3, если бы та была доступна. (Стандарт семантического версионирования не говорит ничего такого о версиях с номером 0.x, но это правило оказалось настолько полезным, что его решили оставить.)
- После опубликования версии 1.0 считается, что совместимость нарушается только с переходом на новую главную версию. Поэтому, если вы запросите версию 2.0.1, Cargo может взять версию 2.17.99, но не 3.0.

Номера версий по умолчанию гибкие, поскольку в противном случае на выбор версии накладывалось бы слишком много ограничений. Предположим, что библиотека libA требует версии num = "0.1.31", а libB – версии num = "0.1.29". Если бы мы требовали точного совпадения номеров версий, то ни в одном проекте эти библиотеки нельзя было бы использовать вместе. Разрешить Cargo брать любую совместимую версию – гораздо более практичное решение.

Тем не менее разные проекты предъявляют различные требования в вопросе зависимостей и версионирования. Операторы позволяют указать точную версию или диапазон версий.

Строка Cargo.toml	Пояснение
image = "=0.10.0"	Использовать в точности версию 0.10.0
image = ">=1.0.5"	Использовать версию 1.0.5 или *любую* более позднюю (даже 2.9, если таковая доступна)
image = ">1.0.5 <1.1.9"	Использовать версию с номером больше 1.0.5, но меньше 1.1.9
image = "<=2.7.10"	Использовать любую версию не старше 2.7.10

Иногда употребляется еще спецификация версии `*`. Для Cargo это означает, что подойдет любая версия. Если в каком-то файле `Cargo.toml` нет более точного ограничения, то Cargo возьмет последнюю доступную версию. В документации по Cargo на странице http://doc.crates.io/crates-io.html спецификации версий рассматриваются более подробно.

Отметим, что правила совместимости означают, что номера версий нельзя выбирать, ориентируясь на чисто маркетинговые соображения. Они действительно кое-что значат. Это контракт между программистами, сопровождающими крейт, и его пользователями. Если вы сопровождаете крейт версии `1.7` и решаете удалить некоторую функцию или внести другое изменение, не обеспечивающее обратную совместимость, то обязаны увеличить номер версии до `2.0`. Присвоив версии номер `1.8`, вы тем самым заявили бы, что новая версия совместима с `1.7`, и тогда пользователи обнаружили бы, что программы, скомпонованные с вашим крейтом, перестали работать.

Cargo.lock

Номера версий в файле `Cargo.toml` сознательно считаются гибкими, но тем не менее мы не хотим, чтобы Cargo обновлял версию библиотеки до последней при каждой сборке. Представьте, что вы с головой погружены в сеанс отладки, как вдруг `cargo build` обновила версию библиотеки. Последствия могут быть крайне печальными. Любое изменение в процессе отладки не сулит ничего хорошего. Да и вообще, когда речь заходит о библиотеках, неожиданным изменениям не должно быть места.

Поэтому в Cargo встроен механизм предотвращения такой беды. При первой сборке проекта Cargo создает файл `Cargo.lock`, в который записываются точные версии всех использованных крейтов. При последующих сборках используются те же самые версии, что перечислены в этом файле. Cargo переходит на новые версии библиотек, только когда вы либо вручную увеличите номер версии в файле `Cargo.toml`, либо выполните команду `cargo update`:

```
$ cargo update
    Updating registry `https://github.com/rust-lang/crates.io-index`
    Updating libc v0.2.7 -> v0.2.11
    Updating png v0.4.2 -> v0.4.3
```

Команда `cargo update` обновляет библиотеки только до последней версии, совместимой с указанной в `Cargo.toml`. Если указана версия `image = "0.6.1"`, а вы хотите обновиться до версии 0.10.0, то должны будете внести изменение в `Cargo.toml`. При следующей сборке Cargo обновит библиотеку `image` и сохранит новый номер версии в файле `Cargo.lock`.

В примере выше показано, как Cargo обновляет два крейта, размещенных на сайте `crates.io`. Нечто подобное происходит для зависимостей, хранящихся в Git. Пусть файл `Cargo.toml` содержит такую строку:

```
image = { git = "https://github.com/PistonDevelopers/image.git", branch = "master" }
```

Команда `cargo build` не станет забирать новые изменения из репозитория Git, если обнаружит файл `Cargo.lock`. Вместо этого она читает `Cargo.lock` и использует те же ревизии, что в прошлый раз. Но команда `cargo update` забирает из-

менения из ветви `master`, поэтому при следующей сборке будет использоваться последняя ревизия.

Файл `Cargo.lock` генерируется автоматически и обычно вручную не редактируется. Тем не менее если в проекте собирается исполняемый файл, то следует записать `Cargo.lock` в систему управления версиями. Тогда каждый, кто собирает проект, будет использовать одни и те же версии. В истории файла `Cargo.lock` будет отражено обновление зависимостей.

Если же речь идет о проекте обычной библиотеки, то записывать `Cargo.lock` в систему управления версиями необязательно. У пользователей библиотеки есть свои файлы `Cargo.lock`, содержащие информацию об их собственных графах зависимостей; файл `Cargo.lock` для вашей библиотеки будет ими проигнорирован. (В редком случае, когда в проекте собирается разделяемая библиотека, т. е. файл с расширением `.dll`, `.dylib` или `.so`, таких пользователей библиотеки не существует, поэтому `Cargo.lock` следует хранить в системе управления версиями.)

Благодаря гибким спецификаторам версий в файле `Cargo.toml` Rust-библиотеки легко использовать в своем проекте и обеспечивать высокую совместимость библиотек. Учет версий с помощью файла `Cargo.lock` поддерживает согласованность и воспроизводимость сборок между машинами. В совокупности эти два механизма помогают избежать ада зависимостей.

Публикация крейтов на сайте crates.io

Вы решили опубликовать библиотеку для моделирования папоротников в виде ПО с открытым исходным кодом. Поздравляем! Это-то как раз просто.

Для начала убедитесь, что Cargo может упаковать вашу библиотеку в крейт.

```
$ cargo package
warning: manifest has no description, license, license-file, documentation,
homepage or repository. See http://doc.crates.io/manifest.html#package-metadata
for more info.
   Packaging fern_sim v0.1.0 (file:///.../fern_sim)
   Verifying fern_sim v0.1.0 (file:///.../fern_sim)
   Compiling fern_sim v0.1.0 (file:///.../fern_sim/target/package/fern_sim-0.1.0)
```

Команда `cargo package` создает файл, в данном случае `target/package/fern_sim-0.1.0.crate`, который содержит все исходные файлы библиотеки, включая `Cargo.toml`. Именно этот файл мы загрузим на `crates.io`, чтобы поделиться плодами своего труда со всем миром. (Чтобы посмотреть, какие файлы вошли в крейт, выполните команду `cargo package --list`.) Затем Cargo проконтролирует себя, построив библиотеку из `crate`-файла, как это будут делать конечные пользователи.

Показанное выше предупреждение напоминает о том, что в секции `[package]` в файле `Cargo.toml` отсутствует информация, важная для конечных пользователей, в т. ч. о лицензии, по которой распространяется код. По указанному в предупреждении URL-адресу находится отличный ресурс, так что мы не станем здесь тратить время на детальные объяснения. Короче говоря, чтобы убрать предупреждение, достаточно добавить несколько строк в `Cargo.toml`:

```
[package]
name = "fern_sim"
```

```
version = "0.1.0"
authors = ["You <you@example.com>"]
license = "MIT"
homepage = "https://fernsim.example.com/"
repository = "https://gitlair.com/sporeador/fern_sim"
documentation = "http://fernsim.example.com/docs"
description = """
Fern simulation, from the cellular level up.
"""
```


После того как крейт опубликован на сайте `crates.io`, всякий, кто его скачает, сможет заглянуть в файл `Cargo.toml`. Поэтому если в поле `authors` указан почтовый адрес, который вы хотели бы сохранить в секрете, то сейчас самое время изменить его.

На этом этапе иногда возникает еще одна проблема: в файле `Cargo.toml` местоположение других крейтов может быть задано в поле `path`, как было показано выше в разделе «Задание зависимостей»:

```
image = { path = "vendor/image" }
```

Для вас и вашей команды это, быть может, и нормально, но у людей, загрузивших библиотеку `fern_sim`, естественно, не будет тех же файлов и каталогов, что на вашем компьютере. Поэтому Cargo *игнорирует* ключ `path` в автоматически загружаемых библиотеках, и это может привести к ошибкам при сборке. Но решить проблему легко: если вы собираетесь опубликовать свою библиотеку на `crates.io`, то там же должны быть и ее зависимости. Поэтому укажите номер версии вместо `path`.

```
image = "0.6.1"
```

Если хотите, можете указать одновременно `path` (используется для локальных сборок) и `version` (для всех остальных пользователей):

```
image = { path = "vendor/image", version = "0.6.1" }
```

Конечно, в этом случае вы должны следить за согласованностью обоих полей.

И наконец, перед тем как публиковать крейт, вы должны зарегистрироваться на сайте `crates.io` и получить ключ API. Это просто: после создания учетной записи на `crates.io` на странице настроек «Account Settings» будет показана команда `cargo login`, например:

```
$ cargo login 5j0dV54BjlXBpUUbfIj7G9DvNl1vsWW1
```

Cargo сохраняет этот ключ в конфигурационном файле, его следует держать в секрете, как пароль. Поэтому запускайте эту команду только на компьютере, который полностью контролируете.

Ну и последний шаг – команда `cargo publish`:

```
$ cargo publish
    Updating registry `https://github.com/rust-lang/crates.io-index`
   Uploading fern_sim v0.1.0 (file:///.../fern_sim)
```

Теперь ваша библиотека присоединилась к тысячам других на сайте `crates.io`.

Рабочие пространства

По мере развития проекта возрастает и количество крейтов. Они размещаются на одном уровне в одном репозитории исходного кода:

```
fernsoft/
├── .git/...
├── fern_sim/
│ ├── Cargo.toml
│ ├── Cargo.lock
│ ├── src/...
│ └── target/...
├── fern_img/
│ ├── Cargo.toml
│ ├── Cargo.lock
│ ├── src/...
│ └── target/...
└── fern_video/
    ├── Cargo.toml
    ├── Cargo.lock
    ├── src/...
    └── target/...
```

Cargo работает так, что у каждого крейта имеется собственный каталог сборки, `target`, в котором находятся сборки всех зависимостей крейта. Эти каталоги абсолютно независимы. Даже если у двух крейтов имеется общая зависимость, у них не может быть общего откомпилированного кода. Это расточительно.

Время компиляции и место на диске можно сэкономить, воспользовавшись *рабочим пространством* Cargo, набором крейтов с общим каталогом сборки и файлом `Cargo.lock`.

Нужно лишь создать файл `Cargo.toml` в корневом каталоге репозитория и поместить в него такие строчки:

```
[workspace]
members = ["fern_sim", "fern_img", "fern_video"]
```

где `fern_sim` и т. д. – имена подкаталогов, содержащих крейты. А из этих подкаталогов удалите находящиеся в них файлы `Cargo.lock` и каталоги `target`.

После этого команда `cargo build`, запущенная в любом крейте, автоматически создает и использует общий каталог сборки, расположенный непосредственно в корневом каталоге (в данном случае `fernsoft/target`). Команда `cargo build --all` собирает все крейты в текущем рабочем пространстве. Команды `cargo test` и `cargo doc` также принимают флаг `--all`.

Прочие вкусности

Если вы еще не пришли в полный восторг, то специально для вас сообщество Rust подготовило еще кое-какие приятные мелочи:

- при публикации крейта с открытым исходным кодом на сайте `crates.io` документация по нему автоматически строится и размещается на сайте `docs.rs`. Спасибо Онуру Аслану (Onur Aslan);

- если проект размещен на Github, то система Travis CI может собирать и тестировать ваш код при каждой записи в репозиторий. Настроить ее на удивление просто, детали смотрите на сайте `travis-ci.org`. Для тех, кто уже знаком с Travis, приведем файл `.travis.yml`, с которого можно начать:

```
language: rust
rust:
  - stable
```

- можно сгенерировать файл `README.md` по документирующему комментарию на верхнем уровне крейта. Эта функция предложена Ливио Рибейро (Livio Ribeiro) в виде плагина к Cargo. Для установки плагина выполните команду `cargo install readme`, а затем команду `cargo readme -help`, чтобы узнать, как им пользоваться.

Можно было бы продолжать и дальше.

Rust – новый язык, но проектируется он для поддержки больших и амбициозных проектов. Для него есть великолепные инструментальные средства и активное сообщество. У системных программистов *действительно* могут быть симпатичные штучки.

Глава 9

Структуры

> Давным-давно, когда пастухам нужно было узнать, изоморфны ли их стада овец, они стремились найти явный изоморфизм.
>
> — *Джон К. Баез и Джеймс Долан «Категорификация»*[1]

Структуры в Rust напоминают типы `struct` в C и C++, классы в Python и объекты в JavaScript. В структуре собрано несколько значений разных типов, образующих единое значение. Мы можем читать и изменять отдельные компоненты структуры. Со структурой могут быть ассоциированы методы для работы с ее компонентами.

В Rust есть три вида структурных типов: *с именованными полями*, *кортежеподобные* и *безэлементные*. Отличаются они способом обращения к компонентам: в структуре с именованными полями каждому компоненту присваивается имя, а в кортежеподобной структуре компоненты идентифицируются порядковым номером. У безэлементных (unit-like) структур вообще нет компонентов; они встречаются не часто, но все же полезнее, чем может показаться на первый взгляд.

В этой главе мы подробно рассмотрим все три вида структур и покажем, как они размещаются в памяти. Мы покажем, как добавлять методы, как определять универсальные структуры, работающие с компонентами переменного типа, и как попросить Rust сгенерировать реализации часто употребляемых характеристик.

Структуры с именованными полями

Определение структуры с именованными полями выглядит примерно так:

```
/// Прямоугольная область восьмиразрядных полутоновых пикселей.
struct GrayscaleMap {
    pixels: Vec<u8>,
    size: (usize, usize)
}
```

Здесь объявлен тип `GrayscaleMap` с двумя полями, `pixels` и `size`, заданных типов. В Rust принято соглашение, в соответствии с которым имена всех типов, в т. ч. структур, записываются в *верблюжьей нотации*, т. е. каждое слово начинается с большой буквы. Имена же полей и методов записываются в *змеиной нотации*: слова разделяются знаками подчеркивания.

---

[1] https://arxiv.org/abs/math/9802029.

Для конструирования выражения этого типа можно воспользоваться *структурным выражением*, например:

```
let width = 1024;
let height = 576;
let image = GrayscaleMap {
    pixels: vec![0; width * height],
    size: (width, height)
};
```

Структурное выражение начинается именем типа (`GrayscaleMap`), а затем в фигурных скобках перечисляются имена и значения полей. Существует также сокращенная запись для присваивания полям значений одноименных локальных переменных или аргументов:

```
fn new_map(size: (usize, usize), pixels: Vec<u8>) -> GrayscaleMap {
    assert_eq!(pixels.len(), size.0 * size.1);
    GrayscaleMap { pixels, size }
}
```

Структурное выражение `GrayscaleMap { pixels, size }` – сокращение `GrayscaleMap { pixels: pixels, size: size }`. В одном структурном выражении разрешается использовать для одних полей синтаксис `key: value`, а для других – сокращенную нотацию.

Для доступа к полям структуры служит оператор `.`:

```
assert_eq!(image.size, (1024, 576));
assert_eq!(image.pixels.len(), 1024 * 576);
```

Как и другие артикулы, структуры по умолчанию закрыты, т. е. видимы лишь в том модуле, в котором объявлены. Чтобы сделать структуру видимой извне ее модуля, нужно в ее определение добавить ключевое слово `pub`. То же самое относится к полям структуры, по умолчанию также закрытым:

```
/// Прямоугольная область восьмиразрядных полутоновых пикселей.
pub struct GrayscaleMap {
    pub pixels: Vec<u8>,
    pub size: (usize, usize)
}
```

Даже если структура объявлена открытой, ее поля могут быть закрытыми:

```
/// Прямоугольная область восьмиразрядных полутоновых пикселей.
pub struct GrayscaleMap {
    pixels: Vec<u8>,
    size: (usize, usize)
}
```

Другие модули могут пользоваться этой структурой и ее открытыми методами, но не могут обращаться к закрытым полям по имени или создавать новые значения типа `GrayscaleMap` с помощью структурных выражений. То есть для создания структурного значения все поля структуры должны быть видимы. Именно поэтому с помощью структурных выражений нельзя создать новый экземпляр типа `String` или `Vec`. Эти стандартные типы являются структурами, но все их поля закрыты. Для создания экземпляра необходимо пользоваться открытыми методами, например `Vec::new()`.

При создании значения структуры с именованными полями можно использовать другую структуру того же типа для подстановки опущенных полей. Если в структурном выражении за именованными полями следует `.. EXPR`, то значения всех не упомянутых явно полей берутся из `EXPR`, которое должно быть другим значением того же структурного типа. Пусть имеется структура для представления монстра в игре:

```
struct Broom {
    name: String,
    height: u32,
    health: u32,
    position: (f32, f32, f32),
    intent: BroomIntent
}

/// Два варианта действий `Broom`.
#[derive(Copy, Clone)]
enum BroomIntent { FetchWater, DumpWater }
```

Самая лучшая сказка для программистов – «Ученик чародея». В ней начинающий волшебник заколдовывает метлу (broom), чтобы она сделала за него работу, но не знает, как потом остановить ее. Разрубание метлы пополам приводит к появлению двух метел меньшего размера, каждая из которых продолжает заниматься тем же делом, что исходная.

```
// Получить входную метлу Broom по значению, приняв владение ей.
fn chop(b: Broom) -> (Broom, Broom) {
    // Инициализировать большую часть полей `broom1` из `b`, изменив только `height`.
    // Поскольку `String` - некопируемый тип, `broom1` принимает владение полем name `b`.
    let mut broom1 = Broom { height: b.height / 2, .. b };

    // Инициализировать большую часть полей `broom2` из `broom1`. Поскольку
    // `String` - некопируемый тип, мы должны клонировать `name` явно.
    let mut broom2 = Broom { name: broom1.name.clone(), .. broom1 };

    // Дать каждой половине метлы свое имя.
    broom1.name.push_str(" I");
    broom2.name.push_str(" II");

    (broom1, broom2)
}
```

При таком определении мы можем создать метлу, разрубить ее пополам и посмотреть, что получится:

```
let hokey = Broom {
    name: "Hokey".to_string(),
    height: 60,
    health: 100,
    position: (100.0, 200.0, 0.0),
    intent: BroomIntent::FetchWater
};

let (hokey1, hokey2) = chop(hokey);
assert_eq!(hokey1.name, "Hokey I");
```

```
assert_eq!(hokey1.health, 100);

assert_eq!(hokey2.name, "Hokey II");
assert_eq!(hokey2.health, 100);
```

Кортежеподобные структуры

Структуры второго вида называются кортежеподобными, потому что напоминают кортеж:

```
struct Bounds(usize, usize);
```

Значение такого типа конструируется почти так же, как кортеж, только необходимо указать имя структуры:

```
let image_bounds = Bounds(1024, 768);
```

Значения, хранящиеся в кортежеподобной структуре, называются «элементами», как и значения кортежа. Доступ к ним производится так же, как к элементам кортежа:

```
assert_eq!(image_bounds.0 * image_bounds.1, 786432);
```

Отдельные элементы кортежеподобной структуры могут быть открытыми или закрытыми:

```
pub struct Bounds(pub usize, pub usize);
```

Выражение `Bounds(1024, 768)` выглядит как вызов функции и на самом деле таковым и является: определение типа неявно определяет некоторую функцию:

```
fn Bounds(elem0: usize, elem1: usize) -> Bounds { ... }
```

На самом нижнем уровне оба вида структур – с именованными полями и кортежеподобные – очень похожи. Какую выбрать, зависит от таких факторов, как удобочитаемость, недвусмысленность и краткость. Если вы собираетесь часто использовать оператор `.` для доступа к компонентам значения, то идентификация полей по имени даст читателю больше информации и, наверное, лучше защитит от опечаток. Если для поиска элементов будет в основном использоваться сравнение с образцом, то хорошо подойдут кортежеподобные структуры.

Кортежеподобные структуры особенно хороши для *неотипов* (newtype) – структур с одним полем, определяемых только ради более строгой проверки типов. Например, при работе с текстом, содержащим только символы ASCII, можно было бы определить такой неотип:

```
struct Ascii(Vec<u8>);
```

Использовать его для представления ASCII-строк гораздо лучше, чем просто передавать буферы типа `Vec<u8>` и в комментариях объяснять, что они значат. Неотип помогает Rust обнаруживать ошибки, состоящие в том, что функции, ожидающей получить ASCII-текст, передается какой-то другой байтовый буфер. В главе 21 будет приведен пример применения неотипов для эффективных преобразований типов.

Безэлементные структуры

Третий вид структур вызывает некоторое недоумение: это структурный тип вообще без элементов:

```
struct Onesuch;
```

Значение такого типа не занимает места в памяти, как и единичный тип `()`. Rust не утруждает себя сохранением значений безэлементных структур в памяти или генерацией кода для работы с ними, поскольку все, что ему нужно знать о значении, можно определить по одному только типу. Однако, с логической точки зрения, пустая структура – это тип, имеющий значение, как и любой другой, – точнее, это тип, у которого есть только одно значение:

```
let o = Onesuch;
```

Мы уже встречали безэлементную структуру в разделе «Поля и элементы» главы 6. Если выражение вида `3..5` – сокращенная запись структурного значения `Range { start: 3, end: 5 }`, то выражение `..` (диапазон, оба конца которого опущены) – сокращенная запись безэлементной структуры `RangeFull`.

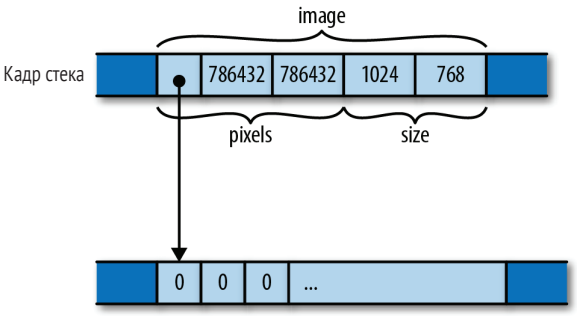
Безэлементные структуры также полезны при работе с характеристиками, о которых мы будем говорить в главе 11.

Размещение структуры в памяти

В памяти структуры с именованными полями и кортежеподобные структуры неразличимы: набор значений, возможно разных типов, размещенных определенным образом. Например, выше мы определили такую структуру:

```
struct GrayscaleMap {
    pixels: Vec<u8>,
    size: (usize, usize)
}
```

Значение типа `GrayscaleMap` размещается в памяти следующим образом:



В отличие от C и C++, Rust не дает конкретных обещаний о порядке размещения полей или элементов структуры в памяти, на рисунке показана лишь одна из возможных конфигураций. Однако Rust все же гарантирует, что значения полей

хранятся непосредственно в блоке памяти, занятом структурой. Если в JavaScript, Python и Java значения `pixels` и `size` были бы помещены в отдельные области кучи, а поля `GrayscaleMap` просто указывали бы на них, то Rust размещает `pixels` и `size` прямо в значении `GrayscaleMap`. И лишь выделенный из кучи буфер, принадлежащий вектору `pixels`, занимает собственный блок памяти.

Задание атрибута `#[repr(C)]` означает, что Rust должен размещать структуры в памяти способом, совместимым с C и C++. Детально этот вопрос будет рассмотрен в главе 21.

Определение методов с помощью ключевого слова impl

В этой книге мы то и дело вызываем методы самых разных значений. Мы добавляли элементы в векторы методом `v.push(e)`, узнавали длину вектора методом `v.len()`, проверяли, содержит ли значение `Result` ошибку, методом `r.expect("msg")` и т. д.

Методы можно определять для любой структуры. Но находятся они не внутри определения структуры, как в C++ или Java, а в отдельном блоке `impl`, например:

```
/// Очередь символов, обслуживаемая по принципу "последним пришел, первым ушел".
pub struct Queue {
    older: Vec<char>,   // старые элементы, последний - самый старый.
    younger: Vec<char>  // молодые элементы, последний - самый молодой.
}

impl Queue {
    /// Добавить символ в конец очереди.
    pub fn push(&mut self, c: char) { self.younger.push(c);
    }

    /// Взять символ из начала очереди. Вернуть `Some(c)`, если в очереди был
    /// хотя бы один символ, или `None`, если очередь была пуста.
    pub fn pop(&mut self) -> Option<char> {
        if self.older.is_empty() {
            if self.younger.is_empty() {
                return None;
            }

            // Переместить элементы из younger в older и расположить их
            // в обещанном порядке.
            use std::mem::swap;
            swap(&mut self.older, &mut self.younger);
            self.older.reverse();
        }

        // Теперь гарантируется, что в older что-то есть. Метод pop типа Vec
        // уже возвращает Option, так что мы можем ничего не делать.
        self.older.pop()
    }
}
```

Блок `impl` – это просто набор определений функций `fn`, каждая из которых становится методом структурного типа, чье имя указано в начале блока. В данном

случае мы определили открытую структуру Queue, а затем снабдили ее двумя открытыми методами, push и pop.

Методы называют также «ассоциированными функциями», потому что они ассоциированы с определенным типом. Противоположностью ассоциированной является «свободная функция», не определенная ни в каком блоке impl.

В первом аргументе Rust передает методу значение, для которого этот метод вызван. Этот аргумент должен иметь специальное имя self. Поскольку типом self, очевидно, является тип, указанный в начале блока impl, или ссылка на него, то Rust позволяет опускать тип и писать self, &self или &mut self вместо self: Queue, self: &Queue или self: &mut Queue соответственно. При желании можете использовать более длинную форму, но почти во всех Rust-программах употребляется сокращенная.

В нашем примере методы push и pop обращаются к полям Queue как к self.older и self.younger. В отличие от C++ и Java, где члены объекта «this» видны в теле метода как идентификаторы без дополнительной квалификации, в методах Rust употребление self для указания значения, от имени которого метод вызван, обязательно. Это аналогично использованию self в Python и this в JavaScript.

Поскольку методы push и pop модифицируют очередь Queue, они принимают значение типа &mut self. Но при вызове метода заимствовать изменяемую ссылку самостоятельно не следует, т. к. это подразумевается неявно. Таким образом, определенный выше тип Queue можно использовать так:

```
let mut q = Queue { older: Vec::new(), younger: Vec::new() };

q.push('0');
q.push('1');
assert_eq!(q.pop(), Some('0'));

q.push('∞');
assert_eq!(q.pop(), Some('1'));
assert_eq!(q.pop(), Some('∞'));
assert_eq!(q.pop(), None);
```

Написав просто q.push(...), мы позаимствовали изменяемую ссылку на q, как если бы написали (&mut q).push(...), поскольку это именно то, что требует аргумент self метода. Если методу не нужно модифицировать свой аргумент self, то можно принять его по разделяемой ссылке, например:

```
impl Queue {
    pub fn is_empty(&self) -> bool {
        self.older.is_empty() && self.younger.is_empty()
    }
}
```

И снова выражение вызова метода знает, как заимствовать ссылку:

```
assert!(q.is_empty());
q.push('☉');
assert!(!q.is_empty());
```

А если метод хочет стать владельцем self, то может принять self по значению:

```
impl Queue {
    pub fn split(self) -> (Vec<char>, Vec<char>) {
```

```
        (self.older, self.younger)
    }
}
```

Такой метод `split` вызывается так же, как любой другой:

```
let mut q = Queue { older: Vec::new(), younger: Vec::new() };

q.push('P');
q.push('D');
assert_eq!(q.pop(), Some('P'));
q.push('X');

let (older, younger) = q.split();
// Теперь q не инициализирована.
assert_eq!(older, vec!['D']);
assert_eq!(younger, vec!['X']);
```

Отметим, однако, что, поскольку `split` принимает `self` по значению, владение очередью `Queue` *передается* от `q`, так что `q` остается неинициализированной. Поскольку аргумент `self` метода `split` теперь владеет очередью, он может выделить из нее отдельные векторы и вернуть их вызывающей стороне.

Можно также определять методы, вообще не принимающие аргумента `self`. Они становятся ассоциированы с самим структурным типом, а не с конкретным значением этого типа. Следуя традиции, сложившейся в C++ и Java, Rust называет такие методы *статическими*. Они часто используются в качестве конструкторов:

```
impl Queue {
    pub fn new() -> Queue {
        Queue { older: Vec::new(), younger: Vec::new() }
    }
}
```

Мы обращаемся к этому методу по имени `Queue::new`: имя типа, двойное двоеточие и имя метода. Теперь наш код становится чуть более изящным:

```
let mut q = Queue::new();

q.push('*');
...
```

В Rust принято называть функции-конструкторы `new`; нам уже встречались функции `Vec::new`, `Box::new`, `HashMap::new` и другие. Однако в имени `new` нет ничего специального. Это не ключевое слово, и зачастую у типов есть и другие статические методы, выступающие в роли конструктора, например `Vec::with_capacity`.

Хотя для одного типа может существовать несколько блоков `impl`, все они должны находиться в том же крейте, что и сам тип. Впрочем, Rust все же позволяет присоединять свои методы к чужим типам; как это делается, мы объясним в главе 11.

Если вы привыкли к C++ или Java, то отделение методов типа от его определения может показаться необычным, но у такого решения несколько преимуществ:

- всегда просто найти данные-члены типа. В определении большого класса на C++ иногда приходится просмотреть сотни строк, содержащих определения функций-членов, чтобы найти все данные-члены класса. В Rust данные-члены собраны в одном месте;

- представить себе включение методов непосредственно в структуры с именованными полями еще можно, но вот для кортежеподобных и безэлементных структур это сложнее. А вынесение методов в блоки `impl` обеспечивает единообразный синтаксис. На самом деле в Rust тот же самый синтаксис используется и для определения методов в типах, вовсе не являющихся структурами, например в перечислениях `enum` или примитивных типах вроде `i32` (тот факт, что у любого типа могут быть методы, является одной из причин, по которым в Rust не используется термин «объект», а предпочтение отдается слову «значение»);
- тот же самый синтаксис блоков `impl` пригоден для реализации характеристик, о чем пойдет речь в главе 11.

Универсальные структуры

Приведенное выше определение типа `Queue` неудовлетворительно: оно предназначено для хранения символов, но ни в самой структуре, ни в методах нет ничего специфичного для символов. Если мы бы определили еще одну структуру для хранения, скажем, строк `String`, то весь код остался бы таким же, только нужно было бы заменить `char` на `String`. Это пустая трата времени.

По счастью, структуры в Rust можно сделать *универсальными* (generic), т. е. их определение – шаблон, в который можно подставить любые типы. Вот, например, определение очереди `Queue`, в которой можно хранить значения любого типа:

```
pub struct Queue<T> {
    older: Vec<T>,
    younger: Vec<T>
}
```

Часть `<T>` в `Queue<T>` можно прочитать как «для любого типа элементов `T`...». Следовательно, все определение читается так: «Для любого типа `T` тип `Queue<T>` состоит из двух полей типа `Vec<T>`». Например, в типе `Queue<String>` `T` – это `String`, так что `older` и `younger` имеют тип `Vec<String>`. В типе `Queue<char>` `T` – это `char`, и мы получаем структуру, идентичную той, с которой начали. На самом деле структура `Vec` тоже универсальная и определена похожим образом.

В определении универсальной структуры имена типов в угловых скобках называются «параметрическими типами». Блок `impl` для универсальной структуры выглядит так:

```
impl<T> Queue<T> {
    pub fn new() -> Queue<T> {
        Queue { older: Vec::new(), younger: Vec::new() }
    }

    pub fn push(&mut self, t: T) { self.younger.push(t);
    }

    pub fn is_empty(&self) -> bool {
        self.older.is_empty() && self.younger.is_empty()
    }

    ...
}
```

Прочитать это объявление `impl<T> Queue<T>` можно следующим образом: «Для любого типа `T` ниже приведены некоторые методы типа `Queue<T>`». Далее параметрический тип `T` можно использовать в качестве типа в определениях методов.

Мы воспользовались сокращенной записью параметров `self`; всюду писать полный тип `Queue<T>` было бы громоздко и только отвлекало бы внимание. Есть и еще одно сокращение: для каждого блока `impl`, простого или универсального, определен специальный параметрический тип `Self` (обратите внимание на верблюжью нотацию), обозначающий тот тип, для которого мы определяем методы. В примере выше `Self` совпадает с типом `Queue<T>`, так что определение метода `Queue::new` можно еще подсократить:

```
pub fn new() -> Self {
    Queue { older: Vec::new(), younger: Vec::new() }
}
```

Возможно, вы обратили внимание, что в теле метода `new` нам нет необходимости указывать параметрический тип в выражении конструирования, `Queue { ... }` вполне достаточно. Это пример вывода типа в действии: поскольку на роль типа возвращаемого функцией значения подходит всего один тип – `Queue<T>`, Rust подставляет его за нас. Однако в сигнатурах функций и в определениях типов указывать параметрические типы обязательно. Rust их не выводит, а использует в качестве исходной информации, на основе которой можно вывести типы внутри тела функции.

При вызове статических методов параметрический тип можно указывать явно с помощью нотации турборыбы `::<>`:

```
let mut q = Queue::<char>::new();
```

Но на практике обычно можно поручить вывод типа Rust:

```
let mut q = Queue::new();
let mut r = Queue::new();

q.push("CAD");  // очевидно, Queue<&'static str>
r.push(0.74);   // очевидно, Queue<f64>

q.push("BTC");  // биткойнов за доллар, май 2017
r.push(2737.7); // Rust не способен обнаружить иррациональный всплеск
```

На самом деле именно так мы и поступаем с универсальным структурным типом `Vec` всюду в этой книге.

Универсальными могут быть не только структуры. Перечисления тоже принимают параметрические типы, и синтаксис очень похожий. Мы убедимся в этом в разделе «Перечисления» главы 10.

Структуры с параметрическим временем жизни

В разделе «Структуры, содержащие ссылки» главы 5 мы говорили, что если структура содержит ссылки, то для них нужно указывать время жизни. Так, ниже приведена структура, в которой могли бы храниться ссылки на наибольший и наименьший элемент некоторой срезки:

```
struct Extrema<'elt> {
    greatest: &'elt i32,
    least: &'elt i32
}
```

Выше мы предлагали считать, что объявление вида `struct Queue<T>` означает, что для любого конкретного типа `T` очередь `Queue<T>` содержит значения типа `T`. Аналогично можно считать, что тип `struct Extrema<'elt>` означает, что для любого времени жизни `'elt` структура `Extrema<'elt>` будет содержать ссылки с таким временем жизни.

Следующая функция просматривает срезку и возвращает значение типа `Extrema`, поля которого ссылаются на элементы этой срезки:

```
fn find_extrema<'s>(slice: &'s [i32]) -> Extrema<'s> {
    let mut greatest = &slice[0];
    let mut least = &slice[0];

    for i in 1..slice.len() {
        if slice[i] < *least    { least    = &slice[i]; }
        if slice[i] > *greatest { greatest = &slice[i]; }
    }
    Extrema { greatest, least }
}
```

Поскольку `find_extrema` заимствует элементы срезки `slice`, время жизни которой равно `'s`, то в возвращаемой ей структуре `Extrema` время жизни ссылок тоже равно `'s`. Rust всегда выводит время жизни параметров вызова, поэтому при вызове `find_extrema` указывать их явно необязательно:

```
let a = [0, -3, 0, 15, 48];
let e = find_extrema(&a);
assert_eq!(*e.least, -3);
assert_eq!(*e.greatest, 48);
```

Поскольку возвращаемое значение очень часто имеет такое же время жизни, как аргумент, Rust позволяет опускать времена жизни, когда имеется всего один очевидный кандидат. Мы могли бы записать сигнатуру `find_extrema`, как показано ниже, семантика ее при этом не изменилась бы:

```
fn find_extrema(slice: &[i32]) -> Extrema {
    ...
}
```

Да, *не исключено*, что мы имели в виду `Extrema<'static>`, но это довольно необычно. Rust предлагает сокращенную запись для типичного случая.

Выведение стандартных характеристик для структурных типов

Определить структуру можно очень просто:

```
struct Point {
    x: f64,
    y: f64
}
```

Но, начав использовать тип `Point`, вы быстро столкнетесь со сложностями. В таком виде тип `Point` не является ни копируемым, ни клонируемым. Его нельзя распечатать с помощью макроса `println!("{:?}", point);`, и он не поддерживает операторов `==` и `!=`.

У всех этих свойств в Rust есть имена: `Copy`, `Clone`, `Debug` и `PartialEq`. Называются они характеристиками. В главе 11 мы покажем, как определить характеристики структурных типов вручную. Но эти стандартные характеристики, а также некоторые другие, реализовывать вручную не нужно, если только вы не хотите получить какое-то специальное поведение. Rust может реализовать их автоматически, с механической точностью. Просто снабдите структуру атрибутом `#[derive]`:

```
#[derive(Copy, Clone, Debug, PartialEq)]
struct Point {
    x: f64,
    y: f64
}
```

Все эти характеристики можно реализовать автоматически при условии, что их реализуют все поля структуры. Мы можем попросить Rust вывести характеристику `PartialEq` для структуры `Point`, потому что оба ее поля имеют тип `f64`, который уже реализует `PartialEq`.

Rust может вывести также `PartialCmp`, добавив поддержку операторов сравнения `<`, `>`, `<=` и `>=`. Мы не стали этого делать, потому что сравнение двух точек на «больше-меньше» – вещь довольно странная. Не существует общепринятого отношения порядка на множестве точек, поэтому мы и не поддерживаем этих операторов для значений типа `Point`. По этой-то причине Rust и требует, чтобы мы сами добавляли атрибут `#[derive]`, вместо того чтобы автоматически выводить все характеристики, которые сможет. Другая причина состоит в том, что характеристика всегда открыта, поэтому такие свойства открытого API структуры, как копируемость, клонируемость и т. д., следует выбирать обдуманно.

В главе 13 мы подробно опишем стандартные характеристики Rust и расскажем, какие из них можно вывести с помощью атрибута `#[derive]`.

Внутренняя изменяемость

Изменяемость – такая же вещь, как любая другая: когда ее слишком много, возникают проблемы, но в небольших количествах она зачастую желательна. Допустим, к примеру, что в системе управления робота-паука имеется центральная структура `SpiderRobot`, содержащая настройки и описатели ввода-вывода. Она инициализируется на этапе загрузки робота, а впоследствии значения полей не изменяются.

```
pub struct SpiderRobot {
    species: String,
    web_enabled: bool,
    leg_devices: [fd::FileDesc; 8],
    ...
}
```

Для описания каждой из основных подсистем робота имеется отдельная структура, и все они указывают на `SpiderRobot`:

```
use std::rc::Rc;

pub struct SpiderSenses {
    robot: Rc<SpiderRobot>,  // <-- указатель на настройки и описатели ввода-вывода
    eyes: [Camera; 32],
```

```
    motion: Accelerometer,
    ...
}
```

Структуры для плетения паутины, хищного образа жизни, управления истечением яда и т. д. также содержат интеллектуальный указатель `Rc<SpiderRobot>`. Напомним, что `Rc` означает «reference counting» (подсчет ссылок) и что значение, обернутое `Rc`, всегда является разделяемым, а значит, неизменяемым.

Теперь предположим, что мы хотим оснастить структуру `SpiderRobot` средствами протоколирования, воспользовавшись стандартным типом `File`. Возникает проблема: поле типа `File` должно быть изменяемым (`mut`). Все методы для записи в файл ожидают получить изменяемую ссылку.

Такие ситуации возникают довольно часто. Для решения проблемы нам нужно включить капельку изменяемых данных (`File`) в значение, которое в остальном неизменяемо (структуру `SpiderRobot`). Это называется *внутренней изменяемостью*. Rust предлагает несколько вариантов, в этом разделе мы обсудим два самых простых типа: `std::cell::Cell<T>` и `std::cell::RefCell<T>`.

`Cell<T>` – структура, содержащая единственное закрытое значение типа `T`. Необычного в ней только то, что получить и установить значение этого поля можно даже тогда, когда сама структура `Cell` не помечена ключевым словом `mut`.

- **`Cell::new(value)`** создает новое значение `Cell`, передавая ему владение данным значением `value`.
- **`cell.get()`** возвращает копию значения, хранящегося внутри `cell`.
- **`cell.set(value)`** сохраняет значение `value` в `cell`, уничтожая ранее хранившееся значение.

  Этот метод принимает `self` как неизменяемую ссылку:

  ```
  fn set(&self, value: T)   // обратите внимание: не `&mut self`
  ```

  Разумеется, это необычное поведение для метода с именем «`set`». Мы уже привыкли, что если требуется вносить изменения в данные, то необходим `mut`-доступ. Но уж если на то пошло, в этой необычной детали и заключается смысл существования типа `Cell`. Это просто безопасный способ обойти правила неизменяемости – не больше, не меньше.

У ячеек (cell) есть и другие методы, о которых можно прочитать в документации по адресу https://doc.rust-lang.org/std/cell/struct.Cell.html.

Тип `Cell` пригодился бы для добавления простого счетчика в структуру `SpiderRobot`. Мы можем написать:

```
use std::cell::Cell;

pub struct SpiderRobot {
    ...
    hardware_error_count: Cell<u32>,
    ...
}
```

и тогда даже методы `SpiderRobot` без `mut` смогут обращаться к этому полю типа `u32` с помощью методов `.get()` и `.set()`:

```
impl SpiderRobot {
    /// Увеличить счетчик ошибок на 1.
```

```
    pub fn add_hardware_error(&self) {
        let n = self.hardware_error_count.get();
        self.hardware_error_count.set(n + 1);
    }

    /// True, если была хотя бы одна аппаратная ошибка.
    pub fn has_hardware_errors(&self) -> bool {
        self.hardware_error_count.get() > 0
    }
}
```

Все это замечательно, но не решает нашу проблему протоколирования. Тип `Cell` *не* позволяет вызывать `mut`-методы для разделяемого значения. Метод `.get()` возвращает копию значения в ячейке, поэтому работает, только если тип `T` реализует характеристику `Copy`. Для протоколирования нам нужен изменяемый `File`, а тип `File` некопируемый.

В данном случае следует воспользоваться типом `RefCell`. Как и `Cell<T>`, `RefCell<T>` – универсальный тип, содержащий единственное поле типа `T`. Но, в отличие от `Cell`, `RefCell` поддерживает заимствование ссылок на свое значение.

- **`RefCell::new(value)`** создает новое значение `RefCell`, передавая ему владение данным значением `value`.
- **`ref_cell.borrow()`** возвращает `Ref<T>` – разделяемую ссылку на значение, хранящееся в `ref_cell`.
  Этот метод паникует, если на значение уже заимствована изменяемая ссылка, детали см. ниже.
- **`ref_cell.borrow_mut()`** возвращает `RefMut<T>`, изменяемую ссылку на значение, хранящееся в `ref_cell`.
  Этот метод паникует, если ссылка на значение уже заимствована, детали см. ниже.

В типе `RefCell` есть и другие методы, см. документацию по адресу https://doc.rust-lang.org/std/cell/struct.RefCell.html.

Оба метода `borrow` паникуют при попытке нарушить правило Rust, согласно которому `mut`-ссылки дают исключительный доступ. Например, в следующем фрагменте кода возникла бы паника:

```
let ref_cell: RefCell<String> = RefCell::new("hello".to_string());

let r = ref_cell.borrow();    // ok, возвращает Ref<String>
let count = r.len();          // ok, возвращает "hello".len()
assert_eq!(count, 5);

let mut w = ref_cell.borrow_mut();  // паника: уже заимствована
w.push_str(" world»);
```

Чтобы избежать паники, можно было бы поместить заимствования в разные блоки. Тогда `r` была бы уничтожена прежде попытки заимствовать `w`.

Это очень похоже на работу нормальных ссылок. Разница только в том, что обычно при заимствовании ссылки на переменную Rust *на этапе компиляции* проверяет, что ссылка используется корректно, и если это не так, компилятор выдает ошибку. Тип `RefCell` применяет то же правило на этапе выполнения. Поэтому если оно будет нарушено, то возникнет паника.

Теперь мы готовы задействовать `RefCell` в типе `SpiderRobot`.

```
pub struct SpiderRobot {
    ...
    log_file: RefCell<File>,
    ...
}

impl SpiderRobot {
    /// Вывести строку в файл журнала.
    pub fn log(&self, message: &str) {
        let mut file = self.log_file.borrow_mut();
        writeln!(file, "{}", message).unwrap();
    }
}
```

Переменная `file` имеет тип `RefMut<File>`. Ее можно использовать как изменяемую ссылку на `File`. Подробнее о записи в файлы см. главу 18.

Работать с ячейками просто. Необходимость вызывать методы `.get()` и `.set()` или `.borrow()` и `.borrow_mut()` несколько утомляет, но это плата за нарушение правил. Еще один недостаток не столь очевиден и более серьезен: ячейки – и любые содержащиеся в них значения – не являются потокобезопасными. Поэтому Rust не разрешает обращаться к ним одновременно из нескольких потоков. О потокобезопасных вариантах внутренней изменяемости мы поговорим в главе 19 при обсуждении мьютексов, атомарных типов и глобальных переменных.

Вне зависимости от того, содержит ли структура именованные поля или является кортежеподобной, она представляет собой агрегат других значений: имея структуру `SpiderSenses`, я имею `Rc`-указатель на разделяемую структуру `SpiderRobot` и глаза, и акселерометр и т. д. Таким образом, внутренний смысл структуры заключен в слове «и»: я имею X *и* Y. Но можно ли построить другой вид типов на основе слова «или»? Такой, что, имея значения этого типа, мы имеем *или* X, *или* Y? Такие типы очень полезны и встречаются в Rust повсеместно. Это и будет темой следующей главы.

Глава 10

Перечисления и образцы

> … Удивительно, сколь многое в информатике обретает смысл, если рассматривать с точки зрения трагической утраты вариантных типов (ср. с утратой лямбда-выражений).
>
> *— Грейдон Хоар*[1]

Первая тема этой главы стара, как мир: радостная помощь в стремительном достижении многого (за плату). Она известна под разными именами во многих культурах. Но мы говорим не о дьяволе, а о разновидности пользовательского типа данных, давно известного программистам на ML и Haskell как вариантные типы (sum type), размеченные объединения (discriminated union) или алгебраические типы данных. В Rust они называются *перечислениями*. В отличие от дьявола, они вполне безопасны, а запрашиваемая цена не станет для вас серьезным лишением.

В C++ и C# есть перечисления; их можно использовать для определения собственных типов, принимающих значения из множества именованных констант. Например, можно определить тип `Color` со значениями `Red`, `Orange`, `Yellow` и т. д. Такие перечисления есть и в Rust. Но Rust пошел дальше. Перечисление в Rust может содержать еще и данные – и даже различных типов. Так, тип `Result<String, io::Error>` в Rust является перечислением; он принимает либо значение `Ok`, содержащее `String`, либо значение `Err`, содержащее `io::Error`. Перечисления в C++ и C# на такое не способны. Это больше походит на объединение `union` в C, но, в отличие от объединений, перечисления в Rust типобезопасны.

Перечисления полезны, когда значение может быть либо тем, либо другим. «Плата» за их использование – необходимость безопасного доступа к данным посредством сопоставления с образцами. Этой темой мы займемся во второй части главы.

Образцы тоже знакомы тем, кто встречался с распаковкой в Python или деструктуризацией в JavaScript, но Rust и здесь идет дальше. Образцы в Rust напоминают регулярные выражения для данных. Они позволяют узнать, имеет ли значение определенный вид. С их помощью можно извлечь сразу несколько полей структуры или кортежа в локальные переменные. И, подобно регулярным выражениям, они лаконичны и, как правило, позволяют добиться желаемого, написав всего одну строчку кода.

---

[1] https://twitter.com/graydon_pub/status/555046888714416128.

Перечисления

Перечисления в стиле C не вызывают никаких сложностей:

```
enum Ordering {
    Less,
    Equal,
    Greater
}
```

Здесь объявлен тип `Ordering` с тремя возможными значениями, которые называются *вариантами*, или *конструкторами*: `Ordering::Less`, `Ordering::Equal`, `Ordering::Greater`. Это перечисление находится в стандартной библиотеке, поэтому его можно импортировать – само по себе:

```
use std::cmp::Ordering;

fn compare(n: i32, m: i32) -> Ordering {
    if n < m {
        Ordering::Less
    } else if n > m {
        Ordering::Greater
    } else {
        Ordering::Equal
    }
}
```

или вместе со всеми конструкторами:

```
use std::cmp::Ordering;
use std::cmp::Ordering::*;   // `*` означает импорт всех потомков

fn compare(n: i32, m: i32) -> Ordering {
    if n < m {
        Less
    } else if n > m {
        Greater
    } else {
        Equal
    }
}
```

После импорта конструкторов мы можем писать `Less` вместо `Ordering::Less` и т. д., но поскольку такая запись менее явная, в общем случае считается правильнее *не* импортировать конструкторы, разве что это делает код гораздо более удобочитаемым.

Для импорта конструкторов перечисления, объявленного в текущем модуле, импортируйте `self`:

```
enum Pet {
    Orca,
    Giraffe,
    ...
}

use self::Pet::*;
```

В памяти значения перечислений в стиле C хранятся как целые числа. Иногда бывает полезно сообщить Rust, какие именно целые числа следует использовать:

```
enum HttpStatus {
    Ok = 200,
    NotModified = 304,
    NotFound = 404,
    ...
}
```

Если этого не сделать, то Rust сопоставляет элементам перечисления последовательные целые числа, начиная с 0.

По умолчанию для хранения перечислений в стиле C используется самый узкий встроенный целый тип, в который могут поместиться все значения. Обычно хватает одного байта.

```
use std::mem::size_of;
assert_eq!(size_of::<Ordering>(), 1);
assert_eq!(size_of::<HttpStatus>(), 2);  // 404 не помещается в тип u8
```

Сделанный Rust выбор представления в памяти можно переопределить, добавив к перечислению атрибут `#[repr]`. Детали смотрите в главе 21.

Разрешается приводить перечисление в стиле C к целому типу:

```
assert_eq!(HttpStatus::Ok as i32, 200);
```

Однако приведение в обратном направлении, из целого в перечисление, запрещено. В отличие от C и C++, Rust гарантирует, что значением перечисления может быть только одно из значений, указанных в объявлении `enum`. Неконтролируемое приведение из целого типа в тип перечисления могло бы нарушить это обязательство, поэтому запрещено. Вы можете либо написать собственное контролируемое преобразование:

```
fn http_status_from_u32(n: u32) -> Option<HttpStatus> {
    match n {
        200 => Some(HttpStatus::Ok),
        304 => Some(HttpStatus::NotModified),
        404 => Some(HttpStatus::NotFound),
        ...
        _ => None
    }
}
```

либо воспользоваться крейтом **enum_primitive** (https://crates.io/crates/enum_primitive). В нем имеется макрос, который автоматически генерирует подобный код.

Как и в случае структур, компилятор может реализовать такие возможности, как оператор `==`, но его об этом нужно попросить.

```
#[derive(Copy, Clone, Debug, PartialEq)]
enum TimeUnit {
    Seconds, Minutes, Hours, Days, Months, Years
}
```

У перечислений, как и у структур, могут быть методы:

```
impl TimeUnit {
    /// Вернуть название единицы времени во множественном числе.
```

```
    fn plural(self) -> &'static str {
        match self {
            TimeUnit::Seconds => "seconds",
            TimeUnit::Minutes => "minutes",
            TimeUnit::Hours => "hours",
            TimeUnit::Days => "days",
            TimeUnit::Months => "months",
            TimeUnit::Years => "years"
        }
    }

    /// Вернуть название единицы времени в единственном числе.
    fn singular(self) -> &'static str {
        self.plural().trim_right_matches('s')
    }
}
```

Ну и довольно о перечислениях в стиле C. Интереснее перечисления Rust, содержащие данные.

Перечисления, содержащие данные

В некоторых программах всегда нужно отображать полные дату и время с точностью до миллисекунды, но в большинстве вполне достаточно грубого приближения, например: «два месяца назад». В этом нам может помочь перечисление.

```
/// Округленная временная метка. Программа говорит "6 months ago",
/// а не "February 9, 2016, at 9:49 AM".
#[derive(Copy, Clone, Debug, PartialEq)]
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32)
}
```

Два варианта в этом перечислении, `RoughTime::InThePast` и `RoughTime::InTheFuture`, принимают аргументы. Они называются *кортежными вариантами*. Как и кортежеподобные структуры, эти конструкторы представляют собой функции, которые создают новые значения типа `RoughTime`.

```
let four_score_and_seven_years_ago =
    RoughTime::InThePast(TimeUnit::Years, 4*20 + 7);

let three_hours_from_now =
    RoughTime::InTheFuture(TimeUnit::Hours, 3);
```

В перечислениях могут быть также *структурные варианты*, которые содержат именованные поля, как обычные структуры:

```
enum Shape {
    Sphere { center: Point3d, radius: f32 },
    Cuboid { corner1: Point3d, corner2: Point3d }
}

let unit_sphere = Shape::Sphere { center: ORIGIN, radius: 1.0 };
```

Короче говоря, в Rust имеются три вида вариантов перечислений, аналогичных трем видам структур, рассмотренным в предыдущей главе. Вариантам без данных соответствуют безэлементные структуры, кортежным вариантам – кортежеподобные структуры, а структурным вариантам – структуры с именованными полями. В одном перечислении могут встречаться варианты всех трех видов.

```
enum RelationshipStatus {
    Single,
    InARelationship,
    ItsComplicated(Option<String>),
    ItsExtremelyComplicated {
        car: DifferentialEquation,
        cdr: EarlyModernistPoem
    }
}
```

Все конструкторы и поля открытого перечисления автоматически открыты.

Перечисления в памяти

В памяти перечисления без данных хранятся в виде небольшого целого *тега* и области, достаточной для хранения всех полей самого большого варианта. Тег используется для внутренних нужд Rust. Он говорит, какой конструктор создал значение и, следовательно, какие поля в нем присутствуют.

В версии Rust 1.18 для хранения типа `RoughTime` нужно 8 байтов, как показано на рис. 10.1:

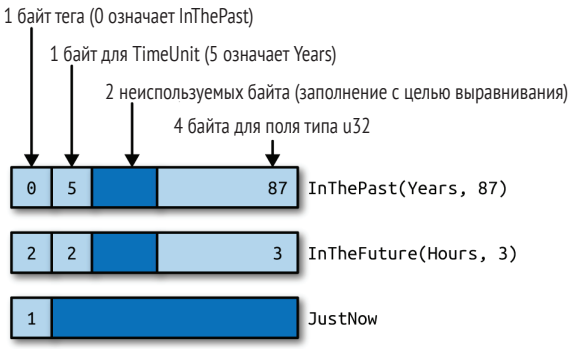


Рис. 10.1 ❖ Значения типа RoughTime в памяти

Rust не дает никаких обещаний относительно точного размещения перечисления в памяти, оставляя открытой дверь для будущих оптимизаций. В некоторых случаях можно упаковать перечисление более эффективно, чем показано на рисунке. Ниже в этой главе мы покажем, что Rust уже способен отказаться от поля тега для некоторых перечислений.

Обогащенные структуры данных на основе перечислений

Перечисления полезны также для быстрой реализации древовидных структур данных. Допустим, к примеру, что Rust-программа должна работать с произволь-

ными данными в формате JSON. В памяти любой JSON-документ можно представить значением следующего типа Rust:

```
enum Json {
    Null,
    Boolean(bool),
    Number(f64),
    String(String),
    Array(Vec<Json>),
    Object(Box<HashMap<String, Json>>)
}
```

Пересказ смысла этой структуры по-русски мало что даст по сравнению с кодом на Rust. В стандарте JSON определены различные типы данных, допустимые в JSON-документе: `null`, булевы значения, числа, строки, массивы элементов JSON и объекты, в которых ключами являются строки, а значениями – элементы JSON. В перечислении `Json` эти типы как раз и перечислены.

Это не гипотетический пример. Очень похожее перечисление имеется в библиотеке сериализации Rust-структур `serde_json`, одном из самых часто скачиваемых крейтов на сайте `crates.io`.

Бокс, который обертывает тип `HashMap`, представляющий вариант `Object`, нужен только для того, чтобы сделать все значения типа `Json` более компактными. В памяти значение этого типа занимает четыре машинных слова. `String` и `Vec` – это три слова, и еще один байт добавляет Rust для тега. Размер вариантов `Null` и `Boolean` меньше, чем эта область, но все значения типа `Json` должны быть одного размера. Лишнее место остается неиспользуемым. На рис. 10.2 приведено несколько примеров размещения значений типа `Json` в памяти.

Но структура `HashMap` все же больше. Если бы мы оставляли место для нее в каждом значении типа `Json`, то они были бы слишком большими – порядка восьми слов. Однако `Box<HashMap>` занимает всего одно слово, это просто указатель на данные, находящиеся в куче. Мы могли бы сделать тип `Json` еще более компактным, обернув боксом больше полей.

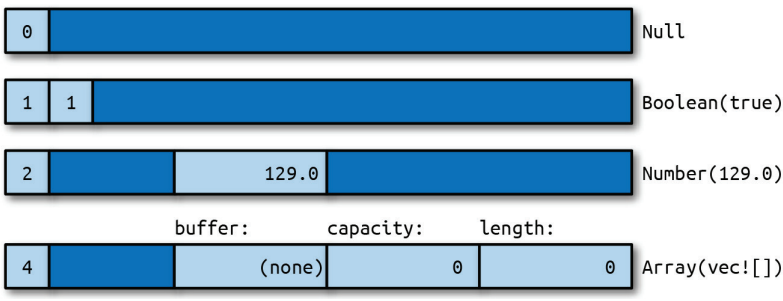


Рис. 10.2 ❖ Значения типа Json в памяти

Замечательна простота задания всего этого. На C++ соответствующий класс мог бы выглядеть так:

```
class JSON {
private:
```

```
    enum Tag {
        Null, Boolean, Number, String, Array, Object
    };
    union Data {
        bool boolean;
        double number;
        shared_ptr<string> str;
        shared_ptr<vector<JSON>> array;
        shared_ptr<unordered_map<string, JSON>> object;

        Data() {}
        ~Data() {}
        ...
    };

    Tag tag;
    Data data;

public:
    bool is_null() const { return tag == Null; }
    bool is_boolean() const { return tag == Boolean; }
    bool get_boolean() const {
        assert(is_boolean());
        return data.boolean;
    }
    void set_boolean(bool value) {
        this->~JSON();  // clean up string/array/object value ***ПЕРЕВОД???***
        tag = Boolean;
        data.boolean = value;
    }
    ...
};
```

Уже 30 строк кода, а мы едва начали. Классу еще понадобятся конструкторы, деструктор и оператор присваивания. Альтернатива – создать иерархию классов с базовым классом `JSON` и подклассами `JSONBoolean`, `JSONString` и т. д. Но при любом подходе библиотека JSON на C++ будет иметь больше десятка методов. Чтобы вникнуть в код и начать его использовать, программисту придется-таки сначала почитать. А на Rust все перечисление занимает 8 строк кода.

Универсальные перечисления

Перечисления могут быть универсальными. Следующие два примера из стандартной библиотеки относятся к числу самых востребованных типов:

```
enum Option<T> {
    None,
    Some(T)
}

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

Эти типы нам уже достаточно хорошо знакомы, и, как видим, синтаксис универсальных перечислений такой же, как универсальных структур. Одна неочевидная деталь состоит в том, что Rust может отказаться от хранения поля тега `Option<T>`, если тип `T` – `Box` или еще какой-то тип интеллектуального указателя. Значение типа `Option<Box<i32>>` хранится в памяти как одно машинное слово, равное 0 для варианта `None` и отличное от нуля, если в боксе находится вариант `Some`.

Для построения универсальных структур данных достаточно всего нескольких строк кода:

```
// Упорядоченная коллекция значений типа `T`.
enum BinaryTree<T> {
    Empty,
    NonEmpty(Box<TreeNode<T>>)
}

// Узел BinaryTree.
struct TreeNode<T> {
    element: T,
    left: BinaryTree<T>,
    right: BinaryTree<T>
}
```

В этих четырех строчках кода определен тип `BinaryTree`, позволяющий хранить произвольное количество значений типа `T`.

В этих двух определениях заключен большой объем информации, поэтому потратим некоторое время, чтобы перевести код на русский язык. Каждое значение типа `BinaryTree` может быть равно либо `Empty`, либо `NonEmpty`. Если оно равно `Empty`, то дерево не содержит никаких данных. Вариант `NonEmpty` содержит значение типа `Box`, указатель на выделенное в куче значение типа `TreeNode`.

Каждое значение типа `TreeNode` содержит один фактический элемент и два значения типа `BinaryTree`. Это означает, что дерево может содержать поддеревья и, следовательно, у непустого (`NonEmpty`) дерева может быть сколь угодно много потомков.

На рис. 10.3 схематически изображено значение типа `BinaryTree<&str>`. Как и в случае `Option<Box<T>>`, Rust обошелся без поля тега, так что значение типа `BinaryTree` занимает всего одно машинное слово.

Построить один узел дерева очень просто:

```
use self::BinaryTree::*;
let jupiter_tree = NonEmpty(Box::new(TreeNode {
    element: "Jupiter",
    left: Empty,
    right: Empty
}));
```

Более крупные деревья строятся из меньших:

```
let mars_tree = NonEmpty(Box::new(TreeNode {
    element: "Mars",
    left: jupiter_tree,
    right: mercury_tree
}));
```

Естественно, при таком присваивании владение узлами `jupiter_node` и `mercury_node` передается их новому родительскому узлу.

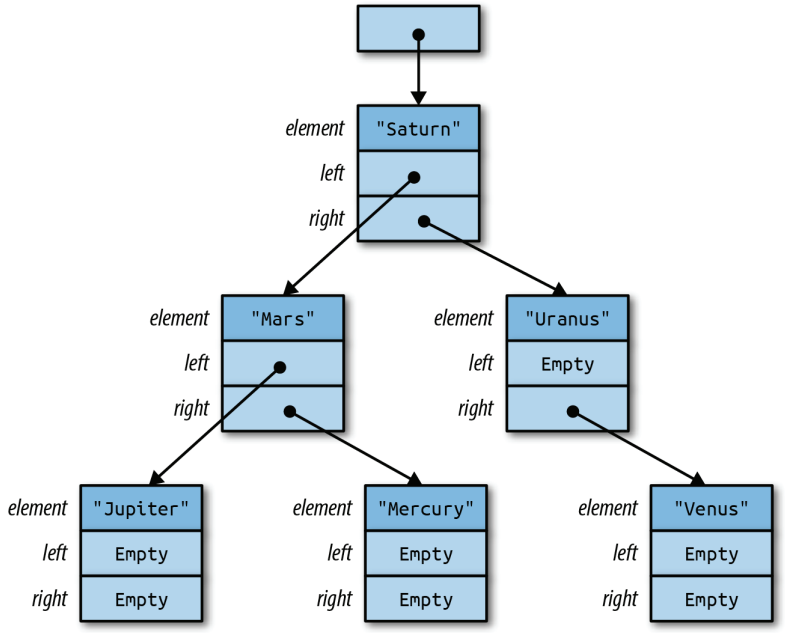


Рис. 10.3 ❖ Двоичное дерево BinaryTree, содержащее шесть строк

Остальные части дерева устроены так же. Корневой узел ничем не отличается от прочих:

```
let tree = NonEmpty(Box::new(TreeNode {
    element: "Saturn",
    left: mars_tree,
    right: uranus_tree
}));
```

Ниже в этой главе мы покажем, как реализовать метод `add` типа `BinaryTree`, чтобы можно было вместо этого написать:

```
let mut tree = BinaryTree::Empty;
for planet in planets {
    tree.add(planet);
}
```

Создание структур данных, подобных `BinaryTree`, в Rust выглядит так же, как в других языках. Поначалу неочевидно, куда поместить боксы `Box`. Один из способов найти работоспособный дизайн – нарисовать картинку, как на рис. 10.3, на которой показано размещение в памяти. А затем, отталкиваясь от картинки, написать код. Каждому набору прямоугольников соответствует структура или кортеж, а каждой стрелке – `Box` или другой интеллектуальный указатель. Выбрать тип каждого поля – задачка, но вполне разрешимая. А наградой за ее решение будет контроль над использованием памяти программой.

А теперь поговорим о «цене», упомянутой во введении. Поле тега, являющее частью перечисления, занимает место в памяти, в худшем случае 8 байтов, но обычно эти накладные расходы пренебрежимо малы. Настоящий недостаток

перечислений (если это можно так назвать) состоит в том, что Rust не может отбросить предосторожность и сделать попытку обратиться к полям, не проверяя, существуют ли они в данном значении.

```
let r = shape.radius;  // ошибка: в типе `Shape` нет поля `radius`
```

Единственный способ безопасно получить доступ к данным в перечислении – воспользоваться образцами.

Образцы

Напомним определение типа `RoughTime`:

```
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32)
}
```

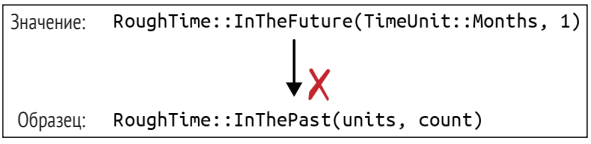
Предположим, что мы хотим отобразить на веб-странице значение типа `RoughTime`. Нам нужен доступ к его полям типа `TimeUnit` и `u32`. Rust не позволяет обратиться к ним напрямую, написав `rough_time.0` и `rough_time.1`, потому что вполне возможно, что это вариант `RoughTime::JustNow`, в котором таких полей нет. Но как же тогда получить данные?

Необходимо выражение `match`:

```
1  fn rough_time_to_english(rt: RoughTime) -> String {
2      match rt {
3          RoughTime::InThePast(units, count) =>
4              format!("{} {} ago", count, units.plural()),
5          RoughTime::JustNow =>
6              format!("just now»),
7          RoughTime::InTheFuture(units, count) =>
8              format!("{} {} from now", count, units.plural())
9      }
10 }
```

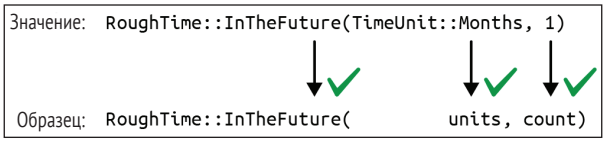
Выражение `match` производит сопоставление с образцом; в данном случае *образцами* являются части, предшествующие символу `=>` в строках 3, 5 и 7. Образцы, с которыми сравниваются значения `RoughTime`, ведут себя в точности как выражения, с помощью которых такие значения создаются. И это не случайность. Выражения *порождают* значения; образцы *потребляют* значения. Синтаксис в обоих случаях очень похож.

Проследим за тем, что происходит при выполнении выражения `match`. Пусть `rt` имеет значение `RoughTime::InTheFuture(TimeUnit::Months, 1)`. Сначала Rust пытается сопоставить это значение с образцом в строке 3. Они не совпадают:



Сопоставление с образцом для перечисления, структуры или кортежа работает так, будто Rust производит простой просмотр слева направо, проверяя совпадение с каждой компонентой образца. Если совпадение не найдено, Rust переходит к следующему образцу.

Для образцов в строках 3 и 5 совпадение отсутствует. Но для образца в строке 7 сравнение увенчалось успехом:



Если образец содержит несколько простых идентификаторов, например `units` и `count`, то они становятся локальными переменными в следующем за ним коде. Поля, присутствующие в значении, копируются или передаются в новые переменные. Rust записывает `TimeUnit::Months` в `units` и `1` в `count`, выполняет строчку 8 и возвращает строку `"1 months from now"`.

В этой строке есть грамматическая ошибка[1], которую можно исправить, добавив в `match` еще одну ветвь:

```
RoughTime::InTheFuture(unit, 1) =>
    format!("a {} from now", unit.singular()),
```

Эта ветвь выбирается, только если поле `count` равно 1. Отметим, что новый код следует поместить перед строчкой 7. Если добавить его в конец, то Rust никогда не доберется до него, потому что с образцом в строчке 7 сопоставляются все значения варианта `InTheFuture`. В случае такой ошибки компилятор Rust выдаст предупреждение о «недостижимом образце» (unreachable pattern).

К сожалению, и в новом коде есть проблема: для `RoughTime::InTheFuture(TimeUnit::Hours, 1)` выдается результат `"a hour from now"`, неверный с точки зрения грамматики английского языка[2]. Ее тоже можно исправить, добавив ветви в выражение `match`.

Как видно из этого примера, сопоставление с образцом работает рука об руку с перечислениями и способно даже проверить содержащиеся в них данные, так что `match` оказывается мощной и гибкой заменой предложению `switch` в C.

До сих пор мы видели только образцы, сопоставляемые со значениями перечислений. Но этим они не исчерпываются. Образцы в Rust образуют отдельный мини-язык, основные черты которого сведены в табл. 10.1. Оставшаяся часть главы будет в основном посвящена возможностям, описанным в этой таблице.

---

[1] После числа 1 существительное должно быть написано в единственном числе, т. е. «month», а не «months». – *Прим. перев.*

[2] Перед словом «hour» (час) ставится неопределенный артикль «an», а не «a». – *Прим. перев.*

Таблица 10.1. Образцы

Тип образца	Пример	Примечания
Литерал	`100` `"name"`	Сопоставляется с точным значением; допустимо также имя константы
Диапазон	`0 … 100` `'a' ... 'k'`	Сопоставляется с любым значением в диапазоне, включая и последнее
Метасимвол	`_`	Сопоставляется с любым значением и игнорирует его
Переменная	`name` `mut count`	Как _, но передает или копирует значение в новую локальную переменную
`ref` переменная	`ref field` `ref mut field`	Заимствует ссылку на сопоставленное значение, вместо того чтобы передавать или копировать его
Связывание с подобразцом	`val @ 0 ... 99` `ref circle @ Shape::Circle { .. }`	Сопоставляется с образцом справа от `@`, используя имя переменной слева

Перечислительный образец	`Some(value)`
	`None`
	`Pet::Orca`	

Кортежный образец	`(key, value)` `(r, g, b)`	

Структурный образец	`Color(r, g, b)`
	`Point { x, y }`
	`Card { suit: Clubs, rank: n }`
	`Account { id, name, .. }`	

Ссылка	`&value` `&(k, v)`	Сопоставляется только со значениями ссылок
Несколько образцов	`'a' | 'A'`	Только в выражении `match` (недопустимо в `let` и др.)
Охранное выражение	`x if x * x <= r2`	Только в выражении `match` (недопустимо в `let` и др.)

Литералы, переменные и метасимволы в образцах

До сих пор мы показывали только, как выражения `match` используются совместно с перечислениями. Но сопоставление можно производить и для других типов. Если необходимо что-то вроде предложения `switch` в C, используйте `match` с целым значением. Целочисленные литералы, например `0` и `1`, могут выступать в роли образцов:

```
match meadow.count_rabbits() {
    0 => {}  // ничего не говорить
    1 => println!("Кролик что-то вынюхивает в клевере."),
    n => println!("На лугу прыгает {} кроликов", n)
}
```

С образцом `0` сопоставление производится, если на лугу нет кроликов, с образцом `1` – если кролик ровно один, а если кроликов два или больше, то мы доходим до третьего образца, `n`. Этот образец представляет собой имя переменной. Он сопоставляется с любым значением, и это значение передается или копируется в новую локальную переменную. В данном случае значение `meadow.count_rabbits()` сохраняется в локальной переменной `n`, которую мы затем и печатаем.

В качестве образцов можно использовать и другие литералы, в т. ч. булевы величины, символы и даже строки:

```
let calendar =
    match settings.get_string("calendar") {
        "gregorian" => Calendar::Gregorian,
        "chinese" => Calendar::Chinese,
        "ethiopian" => Calendar::Ethiopian,
        other => return parse_error("calendar", other)
    };
```

Здесь `other` служит универсальным образцом, как `n` в предыдущем примере. Тот и другой образцы играют ту же роль, что ветвь `default` в предложении `switch`, т. е. сопоставляются с любыми значениями, которые не сопоставились с другими образцами.

Если необходим универсальный образец, но само сопоставленное значение не интересно, то можно использовать знак подчеркивания _:

```
let caption =
    match photo.tagged_pet() {
        Pet::Tyrannosaur => "RRRAAAAAHHHHHH",
        Pet::Samoyed => "*dog thoughts*",
        _ => "I'm cute, love me" // универсальная надпись, подходит любому домашнему животному
    };
```

Универсальный образец сопоставляется с любым значением, но нигде ничего не сохраняется. Поскольку Rust требует, чтобы всякое выражение `match` покрывало все возможные значения, то универсальный образец зачастую необходим и должен быть последним. Даже если вы уверены, что нерассмотренных случаев не существует, все равно включите ветвь «все остальное», и пусть она паникует:

```
// Существует много геометрических фигур типа Shape, но мы поддерживаем выбор
// только текста или прямоугольной области. Выбрать эллипс или трапецию невозможно.
match document.selection() {
    Shape::TextSpan(start, end) => paint_text_selection(start, end),
    Shape::Rectangle(rect) => paint_rect_selection(rect),
    _ => panic!("неожиданный тип выбранного объекта")
}
```

Отметим, что уже существующие переменные нельзя использовать в образцах. Допустим, что мы реализуем настольную игру с шестиугольными клетками и игрок щелкнул мышью, чтобы передвинуть фишку. Для проверки правильности хода нужно сделать что-то в таком роде:

```
fn check_move(current_hex: Hex, click: Point) -> game::Result<Hex> {
    match point_to_hex(click) {
        None =>
            Err("Вне игрового поля."),
        Some(current_hex) =>  // смотрим, пришелся ли щелчок на current_hex
                              // (не работает: см. объяснение ниже)
            Err("Вы уже тут! Щелкните где-нибудь в другом месте."),
        Some(other_hex) =>
            Ok(other_hex)
    }
}
```

Этот код не работает, потому что идентификаторам в образцах соответствуют *новые* переменные. Образец `Some(current_hex)` порождает новую локальную переменную `current_hex`, маскирующую аргумент `current_hex`. Rust выдает несколько предупреждений для такого кода, в частности о том, что последняя ветвь `match` недостижима. Чтобы исправить ошибку, воспользуемся выражением `if`:

```
Some(hex) =>
    if hex == current_hex {
        Err("Вы уже тут! Щелкните где-нибудь в другом месте.")
    } else {
        Ok(hex)
    }
```

Через несколько страниц мы рассмотрим охранные выражения, предлагающие другой способ решения этой проблемы.

Кортежные и структурные образцы

Кортежные образцы сопоставляются с кортежами. Они полезны, когда нужно выделить несколько элементов данных в одном `match`:

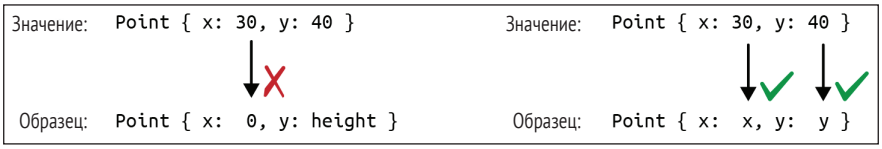
```
fn describe_point(x: i32, y: i32) -> &'static str {
    use std::cmp::Ordering::*;
    match (x.cmp(&0), y.cmp(&0)) {
        (Equal, Equal) => "в начале координат",
        (_, Equal) => "на оси x",
        (Equal, _) => "на оси y",
        (Greater, Greater) => "в первом квадранте",
        (Less, Greater) => "во втором квадранте",
        _ => "где-то еще"
    }
}
```

В структурных образцах используются фигурные скобки, как в выражениях `struct`. Для каждого поля имеется отдельный подобразец:

```
match balloon.location {
    Point { x: 0, y: height } =>
        println!("точно над головой на высоте {} метров", height),
    Point { x: x, y: y } =>
        println!("в точке ({}m, {}m)", x, y)
}
```

Здесь, если произошло сопоставление с первой ветвью, то `balloon.location.y` сохраняется в новой локальной переменной `height`.

Но предположим, что воздушный шар (`balloon.location`) находится в точке `Point { x: 30, y: 40 }`. Как всегда, Rust по очереди проверяет каждую компоненту образца:



Сопоставляется вторая ветвь, поэтому будет выведено «в точке (30m, 40m)».

Образцы вида `Point { x: x, y: y }` часто встречаются при сопоставлении со структурами, а повторяющиеся имена только затрудняют восприятие, потому в Rust допустима сокращенная запись: `Point {x, y}`. Смысл тот же самый. При совпадении с таким образцом поле `x` структуры `Point` по-прежнему сохраняется в новой локальной переменной `x`, а поле `y` – в переменной `y`.

Но даже при использовании такого сокращения сравнение с большой структурой, в которой нас интересует всего несколько полей, выглядит громоздко:

```
match get_account(id) {
    ...
    Some(Account {
            name, language,  // <--- два интересующих нас поля
            id: _, status: _, address: _, birthday: _, eye_color: _,
            pet: _, security_question: _, hashed_innermost_secret: _,
            is_adamantium_preferred_customer: _ }) =>
        language.show_custom_greeting(name)
}
```

Чтобы избежать такого многословия, поставьте `..` – тогда Rust поймет, что остальные поля вас не интересуют:

```
Some(Account { name, language, .. }) =>
    language.show_custom_greeting(name)
```

Ссылочные образцы

Образцы в Rust поддерживают два средства для работы со ссылками. `ref`-образцы заимствуют части сопоставленного значения, а `&`-образцы сопоставляются со ссылками. Сначала рассмотрим `ref`-образцы.

При сопоставлении с некопируемым значением это значение передается. Поэтому следующий код некорректен:

```
match account {
    Account { name, language, .. } => {
        ui.greet(&name, &language);
        ui.show_settings(&account);  // ошибка: использование переданного значения `account`
    }
}
```

При сопоставлении с образцом владение сопоставленным значением часто передается. В данном случае владение полями `account.name` и `account.language` передано локальным переменным `name` и `language`, а остальная часть `account` уничтожена. Потому-то мы и не можем после этого вызывать методы `account`.

Если бы `name` и `language` были копируемыми значениями, то Rust скопировал бы поля, а не передавал их, так что код был бы правильным. Но предположим, что это строки `String`. Что тогда можно сделать?

Нам нужен образец, который *заимствовал* бы сопоставленные значения, а не передавал их. Именно для этого предназначено ключевое слово `ref`:

```
match account {
    Account { ref name, ref language, .. } => {
        ui.greet(name, language);
```

```
        ui.show_settings(&account);  // ok
    }
}
```

Теперь локальные переменные `name` и `language` являются ссылками на соответствующие поля `account`. Поскольку переменная `account` всего лишь заимствуется, а не потребляется, то можно и дальше вызывать ее методы.

Для заимствования изменяемых ссылок пишите `ref mut`:

```
match line_result {
    Err(ref err) => log_error(err),  // `err` имеет тип &Error (разделяемая ссылка)
    Ok(ref mut line) => {            // `line` имеет тип &mut String (изменяемая ссылка)
        trim_comments(line);         // модифицировать String на месте
        handle(line);
    }
}
```

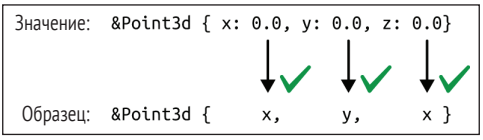
Образец `Ok(ref mut line)` сопоставляется с любым успешным результатом и заимствует изменяемую ссылку на хранящееся в нем значение.

Прямо противоположен `&`-образец, который начинается знаком `&` и сопоставляется со ссылкой.

```
match sphere.center() {
    &Point3d { x, y, z } => ...
}
```

Предположим, что `sphere.center()` возвращает ссылку на закрытое поле структуры `sphere`, как часто бывает в Rust. Возвращенное значение – это адрес значения типа `Point3d`. Если центр сферы совпадает с началом координат, то `sphere.center()` возвращает `&Point3d { x: 0.0, y: 0.0, z: 0.0 }`.

Поэтому сопоставление с образцом происходит так:



Это не сразу понятно, потому что Rust здесь следует по указателю – действие, которое мы привыкли ассоциировать с оператором `*`, а не с оператором `&`. Надо запомнить, что образцы и выражения – естественные противоположности. Выражение `(x, y)` объединяет два значения в кортеж, а образец `(x, y)` делает ровно противоположное: разделяет сопоставленный кортеж на два значения. То же самое и с оператором `&`. В выражении `&` создает ссылку. А в образце – сопоставляется со ссылкой.

Сопоставление со ссылкой подчиняется всем обычным правилам. Времена жизни контролируются. Изменить объект разделяемой ссылки невозможно. Передать владение значением, на которое указывает ссылка, тоже невозможно, даже если эта ссылка изменяемая. При сопоставлении с `&Point3d { x, y, z }` в переменные `x`, `y` и `z` записываются копии координат, а исходное значение `Point3d` остается неизменным. Это работает, потому что указанные поля копируемые. Если бы мы

попробовали проделать то же самое со структурой, содержащей некопируемые поля, то получили бы ошибку:

```
match friend.borrow_car() {
    Some(&Car { engine, .. }) =>  // ошибка: нельзя передать значение, на которое
                                  // заимствована ссылка
        ...
    None => {}
}
```

Разбирать одолженную машину на запчасти некрасиво, и Rust такое не похвалит. Можно воспользоваться `ref`-образцом, чтобы позаимствовать ссылку на деталь. Но владеть ей вы не будете.

```
    Some(&Car { ref engine, .. }) =>  // ok, engine - ссылка
```

Рассмотрим еще один пример `&`-образца. Пусть имеется итератор `chars` для обхода символов строки, и у него есть метод `chars.peek()`, который возвращает `Option<&char>`: ссылку на следующий символ, если таковой имеется (на самом деле, как мы увидим в главе 15, итераторы с методом `peek()` возвращают значение типа `Option<&ItemType>`).

Программа может применить `&`-образец для получения символа, на который указывает ссылка:

```
match chars.peek() {
    Some(&c) => println!("следующим будет: {:?}", c),
    None => println!("больше символов нет")
}
```

Сопоставление с несколькими возможностями

Вертикальная черта (`'|'`) позволяет объединить несколько образцов в одну ветвь `match`.

```
let at_end =
    match chars.peek() {
        Some(&'\r') | Some(&'\n') | None => true,
        _ => false
    };
```

В этом выражении `|` – оператор поразрядного ИЛИ, но по смыслу он больше похож на символ `|` в регулярном выражении. Переменной `at_end` присваивается значение `true`, если `chars.peek()` сопоставляется с одним из трех образцов.

Для сопоставления с целым диапазоном значений служит оператор `...`. Диапазонный образец включает начало и конец, т. е. `'0' ... '9'` сопоставляется с любой ASCII-цифрой.

```
match next_char {
    '0' ... '9' =>
        self.read_number(),
    'a' ... 'z' | 'A' ... 'Z' =>
        self.read_word(),
    ' ' | '\t' | '\n' =>
        self.skip_whitespace(),
```

```
    _ =>
        self.handle_punctuation()
}
```

Диапазоны в образцах *включительные*, поэтому `'0'` и `'9'` сопоставляются с образцом `'0' ... '9'`. Напротив, диапазонные выражения (записываемые с помощью двух точек, как в `for n in 0..100`) являются полуоткрытыми, или *исключительными* (`0` входит, а `100` – нет). Такая несогласованность объясняется тем, что исключительные диапазоны полезнее в циклах и срезках, а включительные – при сопоставлении с образцом.

Охранные выражения

Ключевое слово `if` добавляет охранное выражение в ветвь `match`. Сопоставление будет успешным, только если охранное выражение равно `true`:

```
match robot.last_known_location() {
    Some(point) if self.distance_to(point) < 10 =>
        short_distance_strategy(point),
    Some(point) =>
        long_distance_strategy(point),
    None =>
        searching_strategy()
}
```

Если образец передает владение какими-то значениями, то охранное выражение в нем недопустимо. Если охранное выражение принимает значение `false`, то Rust должен перейти к следующему образцу. Но это невозможно, если владение сопоставляемым значением уже кому-то передано. Поэтому приведенный выше код работает, только если значение `point` копируемое. В противном случае будет выдана ошибка:

```
error[E0008]: cannot bind by-move into a pattern guard
  --> enums_move_into_guard.rs:19:18
   |
19 |             Some(point) if self.distance_to(point) < 10 =>
   |                  ^^^^^ moves value into pattern guard
```

Чтобы обойти ее, нужно изменить образец, так чтобы он заимствовал значение `point`, а не передавал его: `Some(ref point)`.

@-образцы

И наконец, *x* `@` *pattern* сопоставляется в точности как с образцом *pattern*, но в случае успеха создаются не переменные для частей сопоставленного значения, а одна переменная *x*, в которую передается или копируется значение целиком. Рассмотрим такой код:

```
match self.get_selection() {
    Shape::Rect(top_left, bottom_right) =>
        optimized_paint(&Shape::Rect(top_left, bottom_right)),
    other_shape =>
        paint_outline(other_shape.get_outline()),
}
```

В первом случае значение `Shape::Rect` распаковывается только для того, чтобы в следующей же строке создать точно такое же значение `Shape::Rect`. Вместо этого можно воспользоваться `@`-образцом и написать:

```
    rect @ Shape::Rect(..) =>
        optimized_paint(&rect),
```

`@`-образцы полезны и при работе с диапазонами:

```
match chars.next() {
    Some(digit @ '0' ... '9') => read_number(digit, chars),
    ...
}
```

Где еще используются образцы

Конечно, чаще всего образцы встречаются в выражениях `match`, но они разрешены еще в нескольких местах, обычно вместо идентификатора. Семантика всегда одинакова: вместо того чтобы просто сохранить значение в одной переменной, Rust применяет сопоставление с образцом для выделения частей значения.

Следовательно, образцы могут использоваться для решения следующих задач:

```
// распаковать структуру в три локальные переменные
let Track { album, track_number, title, .. } = song;

// распаковать аргумент функции, являющийся кортежем
fn distance_to((x, y): (f64, f64)) -> f64 { ... }

// перебрать ключи и значения HashMap
for (id, document) in &cache_map {
    println!("Document #{}: {}", id, document.title);
}

// автоматически разыменовать аргумент замыкания
// (удобно, потому что иногда внешний код передает ссылку,
// а мы хотели бы иметь копию)
let sum = numbers.fold(0, |a, &num| a + num);
```

В каждом из этих примеров мы экономим две или три строки трафаретного кода. Та же идея встречается и в других языках: в JavaScript она называется «деструктуризацией», а в Python «распаковкой».

Отметим, что во всех четырех примерах образцы гарантированно сопоставляются. Образец `Point3d { x, y, z }` сопоставляется с любым возможным значением структурного типа `Point3d`; образец `(x, y)` сопоставляется с любой парой `(f64, f64)` и т. д. Образцы, которые всегда сопоставляются, занимают особое место в Rust. Они называются *неопровержимыми* (irrefutable), и только такие образцы допускаются в четырех показанных выше местах (после `let`, в аргументах функций, после `for` и в аргументах замыкания).

Опровержимым называется образец, который может не сопоставляться, как `Ok(x)` в случае ошибочного результата или `'0' ... '9'`, не сопоставляемый с символом `'Q'`. Опровержимые образцы можно использовать в ветвях `match`, потому что выражение `match` специально для этой цели и проектировалось: если один образец не сопоставляется, то ясно, что делать дальше. Все четыре приведенных выше

примера – это те места, где применение образцов удобно, но язык не допускает возможности несопоставления.

Опровержимые образцы разрешены также в выражениях `if let` и `while let`, которые используются в следующих случаях:

```
// обработать только один вариант перечисления специальным образом
if let RoughTime::InTheFuture(_, _) =
    user.date_of_birth() { user.set_time_traveler(true);
}

// выполнить некоторый код, если поиск в таблице завершился успешно
if let Some(document) = cache_map.get(&id) {
    return send_cached_response(document);
}

// повторять попытки, пока результат не окажется успешным
while let Err(err) = present_cheesy_anti_robot_task() {
    log_robot_attempt(err);
    // дать пользователю еще одну попытку (это может быть и человек)
}

// ручное итерирование в цикле
while let Some(_) = lines.peek() {
    read_paragraph(&mut lines);
}
```

Подробнее об этих выражениях см. разделы «if let» и «Циклы» главы 6.

Построение двоичного дерева

Выше мы обещали показать, как реализуется метод `BinaryTree::add()`, который добавляет узел в двоичное дерево, описываемое следующими типами:

```
enum BinaryTree<T> {
    Empty,
    NonEmpty(Box<TreeNode<T>>)
}

struct TreeNode<T> {
    element: T,
    left: BinaryTree<T>,
    right: BinaryTree<T>
}
```

Теперь мы знаем об образцах достаточно, чтобы написать этот метод. Объяснение принципов работы двоичных деревьев выходит за рамки этой книги, но читателям, знакомым с данной темой, будет интересно посмотреть, как это делается в Rust.

```
1  impl<T: Ord> BinaryTree<T> {
2      fn add(&mut self, value: T) {
3          match *self {
4              BinaryTree::Empty =>
5                  *self = BinaryTree::NonEmpty(Box::new(TreeNode {
6                      element: value,
7                      left: BinaryTree::Empty,
```

```
 8                      right: BinaryTree::Empty
 9                  })),
10              BinaryTree::NonEmpty(ref mut node) =>
11                  if value <= node.element {
12                      node.left.add(value);
13                  } else {
14                      node.right.add(value);
15                  }
16          }
17      }
18  }
```

В строке 1 объявляется метод типа `BinaryTree`, параметризованного упорядоченным типом. Синтаксис такой же, как при объявлении методов универсальных структур, описанных в разделе «Определение методов с помощью ключевого слова `impl`» главы 9.

Случай, когда существующее дерево `*self` пустое, прост. Выполняются строки 5–9, в которых дерево `Empty` становится `NonEmpty` (непустым). Вызов `Box::new()` выделяет новый узел `TreeNode` из кучи. По завершении дерево содержит один элемент, а левое и правое поддеревья пусты.

Если дерево `*self` не пусто, то происходит сопоставление в образцом в строке 10:

```
BinaryTree::NonEmpty(ref mut node) =>
```

Этот образец заимствует изменяемую ссылку на `Box<TreeNode<T>>`, так что мы можем читать и модифицировать данные в этом узле. Ссылка названа `node`, а ее область видимости занимает строки с 11 по 15. Поскольку в этом узле уже есть элемент, программа должна рекурсивно вызвать метод `.add()`, чтобы добавить новый элемент в левое или правое поддерево.

Написанный метод можно использовать так:

```
let mut tree = BinaryTree::Empty;
tree.add("Mercury");
tree.add("Venus");
...
```

Общая картина

Перечисления в Rust, возможно, и являются новшеством в системном программировании, но вообще-то идея не нова. Под разными «научными» именами вроде «алгебраических типов данных» они используются в функциональном программировании уже свыше сорока лет. Не ясно, почему они так слабо распространены в других языках программирования, берущих начало от C. Быть может, потому что для проектировщика языка слишком трудно соединить в единое целое варианты, ссылки, изменяемость и безопасное использование памяти. Функциональное программирование распростилось с изменяемостью. Напротив, в объединениях C сочетаются варианты, указатели и изменяемость, но они столь очевидно небезопасны, что даже в C считаются крайней мерой. Средства контроля заимствования в Rust – это та магия, благодаря которой удается объединить все четыре свойства и не идти на компромисс.

Программирование – это обработка данных. Выбор правильной формы данных отличает небольшую, быструю и элегантную программу от медленного гигантского переплетения вызовов виртуальных методов, на скорую руку склеенных скотчем.

Именно эту проблему призваны решить перечисления. Это инструмент для приведения данных к нужной форме. В случаях, когда значение может быть таким, сяким или вовсе никаким, перечисления лучше, чем иерархии классов, по любому критерию: быстрее, безопаснее, меньше кода, проще документировать.

Лимитирующим фактором является гибкость. Конечный пользователь перечисления не может расширить его, добавив новые варианты. Чтобы добавить вариант, нужно изменить определение перечисления. А если это сделать, то перестанет работать существующий код. Придется подвергнуть ревизии все выражения `match`, в которых производится сопоставление с вариантами, и в каждое добавить новую ветвь. В некоторых случаях обмен гибкости на простоту диктуется здравым смыслом. В конце концов, структуру языка JSON изменять никто не собирается. А иногда ревизия всех случаев использования перечисления после его изменения – как раз то, что нам и нужно. Например, если `enum` используется в компиляторе для представления различных операторов языка программирования, то после добавления нового оператора мы *обязаны* просмотреть весь код, где операторы обрабатываются.

Но иногда нужна большая гибкость. На такой случай в Rust имеются характеристики, их мы и рассмотрим в следующей главе.

Глава 11

Характеристики и универсальные типы

> Специалист по информатике обычно имеет дело с неоднородными структурами: случай 1, случай 2, случай 3, – тогда как математик хочет иметь единственную объединяющую аксиому, определяющую поведение всей системы.
>
> *— Дональд Кнут*

Одно из основных открытий в программировании – возможность писать код, работающий для значений разных типов, *даже еще не изобретенных*. Вот два примера:

- `Vec<T>` – универсальный тип: мы можем создать вектор значений любого типа, в т. ч. определенного в вашей программе, о котором авторы типа `Vec` и не подозревали;
- во многих типах есть метод `.write()`, например в типах `File` и `TcpStream`. Ваш код может принять по ссылке любого писателя и отправить ему данные. При этом совершенно необязательно знать, что это за писатель. Если впоследствии кто-то добавит новый тип писателя, то ваш код будет поддерживать и его тоже.

Разумеется, эта идея в Rust не нова. Она называется *полиморфизмом*, и новостью в технологии языков программирования была в 1970-х годах. Теперь же используется практически повсеместно. В Rust для поддержки полиморфизма применяются два механизма: характеристики и универсальные типы. Эти концепции знакомы многим программистам, но в Rust используется необычный подход, навеянный классами типов из Haskell.

Характеристики – это аналог интерфейсов или абстрактных базовых классов в Rust. На первый взгляд, они выглядят в точности как интерфейсы в Java или C#. Характеристика для записи байтов называется `std::io::Write`, ее определение в стандартной библиотеке начинается следующим образом:

```
trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;
    fn write_all(&mut self, buf: &[u8]) -> Result<()> { ... }
    ...
}
```

В этой характеристике определено несколько методов, мы показали только первые три.

Стандартные типы `File` и `TcpStream` реализуют характеристику `std::io::Write`. Как и тип `Vec<u8>`. Все три типа предоставляют методы `.write()`, `.flush()` и т. д. В следующем коде писатель используется без уточнения типа:

```
use std::io::Write;

fn say_hello(out: &mut Write) -> std::io::Result<()> {
    out.write_all(b"hello world\n")?;
    out.flush()
}
```

Аргумент `out` имеет тип `&mut Write`, т. е. «изменяемая ссылка на любое значение, реализующее характеристику `Write`».

```
use std::fs::File;
let mut local_file = File::create("hello.txt")?;
say_hello(&mut local_file)?;  // работает

let mut bytes = vec![];
say_hello(&mut bytes)?;  // тоже работает
assert_eq!(bytes, b"hello world\n");
```

Мы начнем эту главу с демонстрации того, как характеристики используются, как они работают и как определить характеристику самостоятельно. Но всем вышеупомянутым характеристики не исчерпываются. С их помощью мы можем добавлять методы расширения в существующие типы, даже встроенные, как, например, `str` и `bool`. Мы объясним, почему добавление характеристики в тип не требует дополнительной памяти и каким образом использование характеристик позволяет избежать накладных расходов, сопровождающих виртуальные методы. Мы увидим, что встроенные характеристики – это тот механизм, который в Rust применяется для перегрузки операторов и других языковых средств. Мы рассмотрим также тип `Self`, ассоциированные методы и ассоциированные типы – три позаимствованных у Haskell средства, дающих элегантные решения проблем, которые в других языках решаются с помощью различных трюков и обходных путей.

Универсальные типы – еще одно проявление полиморфизма в Rust. Как шаблоны в C++, универсальную функцию или тип можно применять к значениям различных типов.

```
/// Выбрать меньшее из двух значений.
fn min<T: Ord>(value1: T, value2: T) -> T {
    if value1 <= value2 {
        value1
    } else {
        value2
    }
}
```

В этой функции `<T: Ord>` означает, что аргументы `min` могут иметь любой тип `T`, реализующий характеристику `Ord`, т. е. любой упорядоченный тип. Компилятор генерирует машинный код, специализированный для конкретного использованного типа `T`.

Универсальные типы и характеристики тесно связаны. Rust заставляет нас объявлять требование `T: Ord` (называемое *ограничением* – bound) заранее, еще до использования оператора `<=` для сравнения значений типа `T`. Мы остановимся на сходствах и различиях `&mut Write` и `<T: Write>` и расскажем, как выбирать тот или иной способ использования характеристик.

Использование характеристик

Характеристика – это свойство, которое может поддерживаться или не поддерживаться заданным типом. Чаще всего характеристика описывает некоторую способность: то, что тип может делать.

- Значение, реализующее `std::io::Write`, может записывать байты.
- Значение, реализующее `std::iter::Iterator`, может порождать последовательность значений.
- Значение, реализующее `std::clone::Clone`, может создавать собственные клоны в памяти.
- Значение, реализующее `std::fmt::Debug`, можно распечатать макросом `println!()` со спецификатором формата `{:?}`.

Все эти характеристики входят в стандартную библиотеку Rust и реализуются различными стандартными типами.

- Тип `std::fs::File` реализует характеристику `Write`; он записывает байты в локальный файл. Тип `std::net::TcpStream` записывает байты в сетевое соединение. Тип `Vec<u8>` также реализует `Write`. Каждый вызов `.write()` для вектора байтов добавляет данные в конец вектора.
- Тип `Range<i32>` (таков, например, тип диапазона `0..10`) реализует характеристику `Iterator`, как и некоторые итераторные типы, ассоциированные со срезками, хеш-таблицами и т. д.
- Большинство типов из стандартной библиотеки реализует характеристику `Clone`. Исключения составляют в основном типы наподобие `TcpStream`, которые представляют не только данные в памяти.
- Аналогично большинство библиотечных типов реализует характеристику `Debug`.

К методам, объявленным в характеристиках, применимо одно необычное правило: сама характеристика должна находиться в области видимости. Иначе все ее методы будут скрыты.

```
let mut buf: Vec<u8> = vec![];
buf.write_all(b"hello")?;  // ошибка: метод `write_all` не найден
```

В данном случае компилятор выводит сообщение об ошибке, любезно предлагая добавить предложение `use std::io::Write;`. Это действительно решает проблему.

```
use std::io::Write;

let mut buf: Vec<u8> = vec![];
buf.write_all(b"hello")?;  // ok
```

Это правило введено в Rust, потому что, как мы увидим ниже, характеристики можно использовать для добавления новых методов в любой тип – даже в стан-

дартный, например `u32` или `str`. Это могут делать и сторонние крейты. Очевидно, что это может привести к конфликту имен! Но поскольку Rust заставляет импортировать характеристики, которые мы планируем использовать, то крейты могут обратить это мощное средство к своей выгоде, и на практике конфликты встречаются очень редко.

Методы характеристик `Clone` и `Iterator` работают даже без импорта, потому что по умолчанию они всегда находятся в области видимости, т. к. являются частью стандартной прелюдии – множества имен, которые Rust автоматически импортирует в каждый модуль. Вообще, прелюдия по большей части состоит из тщательно подобранных характеристик, мы рассмотрим многие из них в главе 13.

Программисты на C++ и C#, наверное, уже заметили, что методы характеристик напоминают виртуальные методы. Однако вызовы вроде показанного выше по скорости не уступают вызовам обычных методов. Проще говоря, никакого полиморфизма здесь нет. Очевидно, что `buf` – это вектор, а не файл и не сетевое соединение. Компилятор может сгенерировать простой вызов `Vec<u8>::write()`. Он может даже встроить этот метод. (В C++ и C# такое тоже можно сделать, но иногда этому препятствует возможность наличия подклассов.) И лишь обращение по ссылке `&mut Write` влечет за собой накладные расходы на вызов виртуального метода.

Объекты характеристик

Есть два способа использования характеристик для написания полиморфного кода в Rust: объекты характеристик и универсальные типы. Начнем с объектов характеристик.

В Rust нельзя объявить переменную типа `Write`.

```
use std::io::Write;

let mut buf: Vec<u8> = vec![];
let writer: Write = buf;  // ошибка: размер `Write` не постоянный
```

Размер переменной должен быть известен на этапе компиляции, а типы, реализующие `Write`, могут быть любого размера.

Программисту, работавшему на C# или Java, это может показаться удивительным, но причина проста. В Java переменная типа `OutputStream` (стандартный интерфейс Java, аналогичный `std::io::Write`) является ссылкой на объект, реализующий `OutputStream`. Ничем другим, кроме ссылки, она и быть не может. Точно так же обстоит дело в C# и большинстве других языков.

В Rust мы хотим добиться того же, но ссылки в Rust объявляются явно.

```
let mut buf: Vec<u8> = vec![];
let writer: &mut Write = &mut buf;  // ok
```

Ссылка на тип характеристики, например `writer`, называется *объектом характеристики*. Как и любая другая ссылка, объект характеристики указывает на некоторое значение, имеет время жизни и может быть изменяемым (`mut`) или разделяемым.

Отличительная особенность объекта характеристики состоит в том, что Rust обычно не знает типа значения, на которое указывает ссылка, во время компиляции. Поэтому объект характеристики содержит кое-какую дополнительную

информацию о типе объекта ссылки. Она нужна только для внутренних надобностей Rust: для выполнения вызова `writer.write(data)` Rust должен иметь информацию о типе, чтобы динамически определить нужный метод `write` в зависимости от типа `*writer`. Запросить информацию о типе напрямую невозможно, и Rust не поддерживает понижающего приведения типа от объекта характеристики `&mut Write` к конкретному типу, скажем `Vec<u8>`.

Размещение объекта характеристики в памяти

В памяти объект характеристики представляется как толстый указатель, содержащий значение и указатель на таблицу, представляющую тип этого значения. Поэтому каждый объект характеристики занимает два машинных слова, как показано на рис. 11.1.

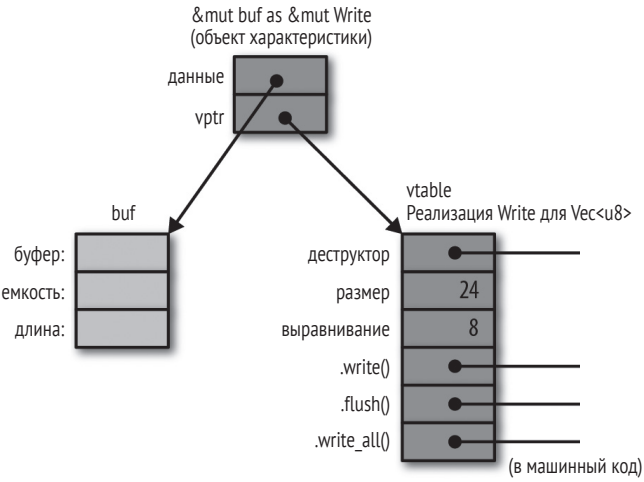


Рис. 11.1 ❖ Объект характеристики в памяти

В C++ тоже есть подобная информация, используемая на этапе выполнения. Она называется *таблицей виртуальных функций*, или *vtable*. В Rust, как и в C++, vtable генерируется один раз во время компиляции и совместно используется всеми объектами одного типа. Все, что закрашено серым цветом на рис. 11.1, включая и vtable, является закрытой деталью реализации Rust. К этим полям и структурам данных нет прямого доступа. Видя вызов метода объекта характеристики, язык автоматически использует vtable, чтобы найти, какую реализацию вызывать.

Опытные программисты на C++, вероятно, заметили, что в Rust и в C++ память используется немного по-разному. В C++ в структуре хранится указатель на vtable, или *vptr*. В Rust вместо этого используется толстый указатель, а сама структура не содержит ничего, кроме своих полей. Поэтому одна структура может реализовывать десятки характеристик, не храня десятков указателей vptr. Даже такие типы, как `i32`, в которых не хватает места для vptr, могут реализовывать характеристики.

Rust автоматически преобразует обычные ссылки в объекты характеристик по мере необходимости. Именно поэтому в следующем примере мы можем передать `&mut local_file` функции `say_hello`:

```
let mut local_file = File::create("hello.txt")?;
say_hello(&mut local_file)?;
```

Переменная `&mut local_file` имеет тип `&mut File`, а аргумент `say_hello` имеет тип `&mut Write`. Поскольку `File` – разновидность писателя, то Rust допускает такой вызов и автоматически преобразует простую ссылку в объект характеристики.

Аналогично Rust с радостью преобразует `Box<File>` в `Box<Write>` – значение, которое владеет писателем в куче.

```
let w: Box<Write> = Box::new(local_file);
```

`Box<Write>`, как и `&mut Write`, является толстым указателем: он содержит адрес самого писателя и адрес таблицы vtable. То же относится и к другим типам указателей, например `Rc<Write>`.

Подобное преобразование – единственный способ создать объект характеристики. Выполняемые компьютером действия очень просты. В момент преобразования Rust знает истинный тип объекта ссылки (в данном случае `File`), поэтому просто добавляет адрес соответствующей vtable, превращая обычный указатель в толстый.

Универсальные функции

В начале этой главы мы привели код функции `say_hello()`, принимающей объект характеристики в качестве аргумента. Переделаем ее в универсальную функцию:

```
fn say_hello<W: Write>(out: &mut W) -> std::io::Result<()> {
    out.write_all(b"hello world\n")?;
    out.flush()
}
```

Изменилась только сигнатура типа:

```
fn say_hello(out: &mut Write)        // простая функция

fn say_hello<W: Write>(out: &mut W)  // универсальная функция
```

Часть `<W: Write>` – это и есть то, что делает функцию универсальной. Это *параметрический тип*, и означает он, что в теле функции `W` обозначает некоторый тип, реализующий характеристику `Write`. По соглашению имена параметрических типов записываются большими буквами.

Какой тип подставляется вместо `W`, зависит от того, как универсальная функция используется.

```
say_hello(&mut local_file)?;  // вызывается say_hello::<File>
say_hello(&mut bytes)?;       // вызывается say_hello::<Vec<u8>>
```

Когда универсальной функции `say_hello()` передается аргумент `&mut local_file`, вызывается `say_hello::<File>()`. Rust генерирует для этой функции машинный код, который вызывает методы `File::write_all()` и `File::flush()`. Если же передается `&mut bytes`, то будет вызвана функция `say_hello::<Vec<u8>>()`. Для нее Rust генерирует другую версию машинного кода, из которой вызываются соответствующие методы `Vec<u8>`. В обоих случаях Rust выводит тип `W` из типа аргумента. Параметрические типы всегда можно указать явно:

```
say_hello::<File>(&mut local_file)?;
```

но это редко бывает необходимо, потому что Rust обычно может вывести их, анализируя аргументы. В данном случае универсальная функция `say_hello` ожидает получить аргумент типа `&mut W`, а мы передали `&mut File`, поэтому Rust заключает, что `W = File`.

Если у вызываемой универсальной функции нет аргументов, дающих полезную информацию, то параметрический тип придется указать явно:

```
// вызвается универсальный метод collect<C>() без аргументов
let v1 = (0 .. 1000).collect();  // ошибка: не удается вывести тип
let v2 = (0 .. 1000).collect::<Vec<i32>>(); // ok
```

Иногда параметрический тип должен обладать несколькими свойствами. Например, если мы хотим напечатать десять значений, чаще всего встречающихся в векторе, то сами значения должны допускать вывод на печать:

```
use std::fmt::Debug;

fn top_ten<T: Debug>(values: &Vec<T>) { ... }
```

Но этого недостаточно. Как мы собираемся искать чаще всего встречающиеся значения? Обычно строится хеш-таблица, в которой значения являются ключами. А это значит, что значения должны поддерживать операции `Hash` и `Eq`. Ограничения на тип `T` должны включать и эти характеристики, помимо `Debug`. Синтаксически это записывается с помощью знака `+`:

```
fn top_ten<T: Debug + Hash + Eq>(values: &Vec<T>) { ... }
```

Одни типы реализуют `Debug`, другие – `Hash`, третьи – `Eq`, и лишь немногие, например `u32` и `String`, поддерживают все три характеристики.

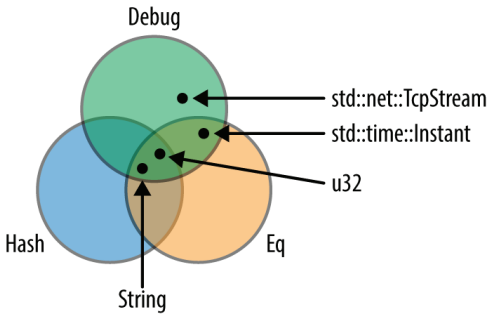


Рис. 11.2 ❖ Характеристики как множества типов

На параметрический тип можно вообще не накладывать никаких ограничений, но с таким значением мало что можно сделать. Нельзя передать владение им. Нельзя поместить его в бокс или в вектор. Ну и так далее.

Универсальная функция может иметь несколько параметрических типов:

```
/// Выполнить запрос для большого набора данных, разбитого на части.
/// См. <http://research.google.com/archive/mapreduce.html>.
fn run_query<M: Mapper + Serialize, R: Reducer + Serialize>(
    data: &DataSet, map: M, reduce: R) -> Results
{ ... }
```

Как видно из этого примера, ограничения могут быть такими длинными, что код становится трудно читать. На такой случай Rust предлагает альтернативный синтаксис с ключевым словом `where`:

```
fn run_query<M, R>(data: &DataSet, map: M, reduce: R) -> Results
    where M: Mapper + Serialize,
          R: Reducer + Serialize
{ ... }
```

Параметрические типы `M` и `R` по-прежнему объявляются в начале, но ограничения вынесены в отдельные строчки. Конструкция `where` допустима также для универсальных структур, перечислений, псевдонимов типов и методов – всюду, где могут встречаться ограничения.

Разумеется, альтернативой `where` является простота: придумайте, как написать программу с минимальным использованием универсальных типов и функций.

В разделе «Получение ссылок в качестве параметров» главы 5 мы познакомились с синтаксисом параметрического времени жизни. Универсальная функция может иметь как параметрическое время жизни, так и параметрические типы. Сначала задаются параметрические времена жизни.

```
/// Возвращает ссылку на точку в множестве `candidates`, ближайшую
/// к точке `target`.
fn nearest<'t, 'c, P>(target: &'t P, candidates: &'c [P]) -> &'c P
    where P: MeasureDistance
{
    ...
}
```

Эта функция принимает два аргумента: `target` и `candidates`. Оба являются ссылками, и им назначается разное время жизни: `'t` и `'c`. Кроме того, функция работает с любым типом `P`, реализующим характеристику `MeasureDistance`, так что ее можно применить к значению типа `Point2d` в одной программе и к значению типа `Point3d` в другой.

Время жизни никогда не оказывает влияния на машинный код. Два вызова `nearest()` с одним и тем же типом `P`, но разными временами жизни, приводят к вызову одной и той же откомпилированной функции. Лишь различие в типах заставляет Rust генерировать несколько версий универсальной функции.

Разумеется, функциями универсализм в Rust не исчерпывается.

- Мы уже встречались с универсальными типами в разделах «Универсальные структуры» и «Универсальные перечисления» главы 9.
- Отдельный метод может быть универсальным, даже если тип, в котором он определен, универсальным не является:

  ```
  impl PancakeStack {
      fn push<T: Topping>(&mut self, goop: T) -> PancakeResult<()> {
          ...
      }
  }
  ```

- Псевдонимы типов также могут быть универсальными:

  ```
  type PancakeResult<T> = Result<T, PancakeError>;
  ```

- Универсальные характеристики рассматриваются ниже в этой главе.

Все рассмотренные в этом разделе свойства – ограничения, фраза `where`, параметрическое время жизни и т. д. – применимы к любым универсальным конструкциям, а не только к функциям.

Что использовать

Вопрос о том, что использовать: объекты характеристик или универсальный код не так просты, как кажется. Поскольку оба средства основаны на характеристиках, у них много общего.

Объекты характеристик стоит предпочесть, если нужна коллекция значений разных типов. Технически возможно настрогать универсальный салат:

```
trait Vegetable {
    ...
}

struct Salad<V: Vegetable> {
    veggies: Vec<V>
}
```

но это уж больно суровый подход. Любой салат будет состоять только из овощей одного вида. Не каждый готов так питаться. Один из авторов этой книги однажды заплатил 14 долларов за `Salad<IcebergLettuce>` и до сих пор не может забыть этот печальный опыт.

А как приготовить салат получше? Поскольку значения типа `Vegetable` могут иметь разные размеры, мы не можем попросить у Rust вектора `Vec<Vegetable>`:

```
struct Salad {
    veggies: Vec<Vegetable>  // ошибка: размер `Vegetable` не постоянный
}
```

Решение дают объекты характеристик:

```
struct Salad {
    veggies: Vec<Box<Vegetable>>
}
```

Каждый бокс `Box<Vegetable>` может владеть овощем любого вида, но размер самого бокса постоянный – два указателя, – что позволяет хранить его в векторе. Если не обращать внимания на неудачную метафору – коробки (Box) в еде, то это именно то, что доктор прописал. Та же идея будет работать для геометрических фигур в программе для рисования, для монстров в игре, для сменных алгоритмов маршрутизации в сетевом маршрутизаторе и т. д.

Еще одна возможная причина использовать объекты характеристик – уменьшить размер откомпилированного кода. У Rust может возникнуть необходимость компилировать универсальную функцию несколько раз – по разу для каждой комбинации параметрических типов. В результате возможно явление, которое в сообществе C++ называют «разбуханием кода». В наше время памяти хватает, и обычно мы можем позволить себе роскошь не обращать внимания на размер кода, однако же среды с ограниченными ресурсами все-таки существуют.

Если отвлечься от салатов и микроконтроллеров, то универсальные функции обладают двумя преимуществами над объектами характеристик, благодаря которым программисты на Rust выбирают их чаще.

Первое преимущество – скорость. При генерации машинного кода универсальной функции компилятору известны все используемые типы, поэтому он знает, какой метод `write` вызывать. Динамическая диспетчеризация не нужна.

Показанная во введении функция `min()` работает так же быстро, как если бы мы написали отдельные функции `min_u8`, `min_i64`, `min_string` и т. д. Компилятор может встроить ее, как и любую другую функцию, так что в выпускной версии вызов `min::<i32>` обойдется в две или три команды. А вызов с постоянными аргументами, например `min(5, 3)`, будет еще быстрее: Rust может вычислить результат на этапе компиляции, так что во время выполнения накладные расходы окажутся нулевыми.

Или рассмотрим такой вызов универсальной функции:

```
let mut sink = std::io::sink();
say_hello(&mut sink)?;
```

`std::io::sink()` возвращает писателя типа `Sink`, который просто отбрасывает все записанные в него байты.

Компилятор мог бы сгенерировать машинный код, который вызывает метод `Sink::write_all`, проверяет отсутствие ошибок и затем вызывает `Sink::flush`. Именно так предписывает делать тело универсальной функции.

Но Rust мог бы вместо этого проанализировать указанные методы и понять, что:

- метод `Sink::write_all()` не делает ничего;
- метод `Sink::flush()` не делает ничего;
- ни тот, ни другой методы никогда не возвращают ошибку.

Короче говоря, у Rust имеется вся необходимая информация, чтобы вообще исключить эту функцию из оптимизированного кода.

Сравним это поведение с объектами характеристик. Rust до момента выполнения не знает, на значение какого типа указывает объект характеристики. Поэтому даже если мы передадим `Sink`, избежать накладных расходов на вызов виртуальных методов и проверку ошибок не удастся.

Второе преимущество универсальных типов состоит в том, что не каждая характеристика может поддержать объект характеристики. Характеристики поддерживают несколько средств, например статические методы, которые рассчитаны только на универсальные типы и полностью исключают объекты характеристик. Ниже мы еще остановимся на этих средствах.

Определение и реализация характеристик

Определить характеристику просто. Достаточно придумать имя и перечислить сигнатуры методов. Например, в игре могла бы пригодиться такая характеристика:

```
/// Характеристика для персонажей, предметов и обстановки -
/// всего, что видно на экране в мире игры.
trait Visible {
    /// Отрисовывает этот объект на заданном холсте.
    fn draw(&self, canvas: &mut Canvas);

    /// Возвращает true, если щелчок мышью в точке (x, y) должен
```

```
    /// выбрать этот объект.
    fn hit_test(&self, x: i32, y: i32) -> bool;
}
```

Для реализации характеристики используется синтаксис `impl` *`TraitName`* `for` *`Type`*.

```
impl Visible for Broom {
    fn draw(&self, canvas: &mut Canvas) {
        for y in self.y - self.height - 1 .. self.y {
            canvas.write_at(self.x, y, '|');
        }
        canvas.write_at(self.x, self.y, 'M');
    }

    fn hit_test(&self, x: i32, y: i32) -> bool {
        self.x == x
        && self.y - self.height - 1 <= y
        && y <= self.y
    }
}
```

Отметим, что блок `impl` содержит реализации всех методов характеристики `Visible` и больше ничего. Все, что определено внутри `impl`, должно быть свойством характеристики; если для поддержки `Broom::draw()` понадобится вспомогательный метод, то его нужно будет определить в отдельном блоке `impl`:

```
impl Broom {
    /// Вспомогательная функция, вызываемая из Broom::draw().
    fn broomstick_range(&self) -> Range<i32> {
        self.y - self.height - 1 .. self.y
    }
}

impl Visible for Broom {
    fn draw(&self, canvas: &mut Canvas) {
        for y in self.broomstick_range() {
            ...
        }
        ...
    }
    ...
}
```

Методы по умолчанию

Для реализации вышеупомянутого типа писателя `Sink` достаточно нескольких строчек. Сначала определяем тип:

```
/// Писатель, который игнорирует передаваемые ему данные.
pub struct Sink;
```

`Sink` – пустая структура, поскольку никакие данные в ней не хранятся. Далее мы предоставляем реализацию характеристики `Write` для `Sink`:

```
use std::io::{Write, Result};

impl Write for Sink {
```

```
    fn write(&mut self, buf: &[u8]) -> Result<usize> {
        // Сообщить, что весь буфер успешно записан.
        Ok(buf.len())
    }

    fn flush(&mut self) -> Result<()> {
        Ok(())
    }
}
```

До сих пор все очень похоже на характеристику `Visible`. Но мы видели, что у характеристики `Write` есть также метод `write_all`:

```
out.write_all(b"hello world\n")?;
```

Так почему же Rust не настаивает на определении этого метода в блоке `impl Write for Sink`? Дело в том, что определение характеристики `Write` в стандартной библиотеке содержит *реализацию* `write_all` *по умолчанию*:

```
trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()> {
        let mut bytes_written = 0;
        while bytes_written < buf.len() {
            bytes_written += self.write(&buf[bytes_written..])?;
        }
        } Ok(())
    }

    ...
}
```

`write` и `flush` – базовые методы, которые должен реализовать любой писатель. Кроме них, писатель может реализовать метод `write_all`, а если он этого не сделает, то будет использована показанная выше реализация по умолчанию.

В свои собственные характеристики вы тоже можете включать реализации по умолчанию.

Крайним примером использования методов по умолчанию в стандартной библиотеке является характеристика `Iterator`, в которой имеются один обязательный метод (`.next()`) и десятки методов по умолчанию. Почему так сделано, объясняется в главе 15.

Характеристики и сторонние типы

Rust позволяет реализовать любую характеристику в любом типе при условии, что либо тип, либо характеристика определены в текущем крейте.

Это означает, что всякий раз, как вы захотите добавить метод в какой-то тип, это можно будет сделать с помощью характеристики.

```
trait IsEmoji {
    fn is_emoji(&self) -> bool;
}

/// Реализовать IsEmoji для встроенного типа символа.
```

```
impl IsEmoji for char {
    fn is_emoji(&self) -> bool {
        ...
    }
}

assert_eq!('$'.is_emoji(), false);
```

Как и любой метод характеристики, метод `is_emoji` виден, только если определение `IsEmoji` находится в области видимости.

Единственная цель этой характеристики – добавить метод в существующий тип `char`. Это называется *расширяющей характеристикой* (extension trait). Конечно, эту характеристику можно реализовать и в других типах, включив, например, блок `impl IsEmoji for str { ... }` и т. п.

Можно даже использовать универсальный блок `impl`, чтобы добавить расширяющую характеристику сразу к целому семейству типов. Следующая расширяющая характеристика добавляет метод во все писатели:

```
use std::io::{self, Write};

/// Характеристика для значений, которым можно послать HTML.
trait WriteHtml {
    fn write_html(&mut self, &HtmlDocument) -> io::Result<()>;
}

/// HTML-код можно записать в любой писатель std::io.
impl<W: Write> WriteHtml for W {
    fn write_html(&mut self, html: &HtmlDocument) -> io::Result<()> {
        ...
    }
}
```

Строка `impl<W: Write> WriteHtml for W` означает: «для любого типа `W`, реализующего `Write`, ниже приведена реализация метода `WriteHtml`».

Библиотека `serde` дает прекрасный пример того, как полезно бывает определять свои характеристики для стандартных типов. `serde` – это библиотека сериализации. Она позволяет записывать структуры данных Rust на диск и впоследствии считывать их. В библиотеке определена характеристика `Serialize`, которая реализуется для всех поддерживаемых библиотекой типов. Так, в исходном коде `serde` есть реализации `Serialize` для типов `bool`, `i8`, `i16`, `i32`, массивов, кортежей и т. д., а также для всех стандартных структур данных, включая `Vec` и `HashMap`.

Мораль сей басни в том, что `serde` добавляет метод `.serialize()` ко всем этим типам. Используется он следующим образом:

```
use serde::Serialize;
use serde_json;

pub fn save_configuration(config: &HashMap<String, String>)
    -> std::io::Result<()>
{
    // Создать сериализатор JSON для записи данных в файл.
    let writer = File::create(config_filename())?;
    let mut serializer = serde_json::Serializer::new(writer);
```

```
    // Все остальное сделает метод serde `.serialize()`.
    config.serialize(&mut serializer)?;

    Ok(())
}
```

Ранее мы сказали, что при реализации характеристики либо сама характеристика, либо тип должны определяться в текущем крейте. Это так называемое *правило сцепленности* (coherence rule). Оно помогает Rust проверить уникальность реализации. Вы не можете включить в свою программу блок `impl Write for u8`, поскольку и `Write`, и `u8` определены в стандартной библиотеке. Если бы Rust позволил включать такие реализации в крейты, то характеристика `Write` могла бы иметь несколько реализаций для типа `u8` в различных крейтах и Rust не смог бы решить, какую из них использовать для данного вызова метода.

(В C++ есть аналогичное ограничение уникальности: правило одного определения. Как часто бывает в C++, компилятор проверяет его выполнение только в простейших случаях, и при нарушении имеет место неопределенное поведение.)

Употребление Self в характеристиках

Ключевое слово `Self` может употребляться в характеристиках в роли типа. Например, стандартная характеристика `Clone` выглядит так (с некоторыми упрощениями):

```
pub trait Clone {
    fn clone(&self) -> Self;
    ...
}
```

Здесь использование `Self` в качестве типа возвращаемого значения означает, что `x.clone()` имеет такой же тип, как `x`. Если `x` имеет тип `String`, то и `x.clone()` будет иметь тип `String` – а не `Clone` или еще какой-то клонируемый тип.

Аналогично, если определить такую характеристику:

```
pub trait Spliceable {
    fn splice(&self, other: &Self) -> Self;
}
```

с двумя реализациями:

```
impl Spliceable for CherryTree {
    fn splice(&self, other: &Self) -> Self {
        ...
    }
}

impl Spliceable for Mammoth {
    fn splice(&self, other: &Self) -> Self {
        ...
    }
}
```

то внутри первого блока `impl Self` является псевдонимом `CherryTree`, а внутри второго – псевдонимом `Mammoth`. Это означает, что мы можем скрестить два вишневых

дерева или двух мамонтов, но не можем создать гибрид вишни с мамонтом. Типы `self` и `other` должны совпадать.

Характеристика, в которой используется тип `Self`, несовместима с объектами характеристик:

```
// ошибка: характеристику `Spliceable` нельзя преобразовать в объект
fn splice_anything(left: &Spliceable, right: &Spliceable) {
    let combo = left.splice(right);
    ...
}
```

С причиной этого мы будем сталкиваться снова и снова при изучении продвинутых свойств характеристик. Rust отклоняет этот код, потому что не может проверить с точки зрения типов вызов `left.splice(right)`. Идея объектов характеристик в том и состоит, что тип неизвестен до момента выполнения. У Rust нет никакого способа на этапе компиляции узнать, совпадают ли типы `left` и `right`, как то требуется.

Объекты характеристик предназначены для простейших характеристик, которые можно было бы реализовать с помощью интерфейсов в Java или абстрактных базовых классов в C++. Более сложные возможности характеристик полезны, но не могут сосуществовать с объектами характеристик, потому что при переходе к объекту характеристики теряется информация о типе. А Rust обязан проверить корректность использования типов в программе.

Если бы мы захотели реализовать генетически невозможное скрещивание, то могли бы спроектировать характеристику, совместимую с объектами характеристик:

```
pub trait MegaSpliceable {
    fn splice(&self, other: &MegaSpliceable) -> Box<MegaSpliceable>;
}
```

У такого метода `.splice()` нет проблем с проверкой типов, потому что тип аргумента `other` не обязан совпадать с типом `self`, нужно лишь, чтобы оба реализовывали `MegaSpliceable`.

Подхарактеристики

Мы можем объявить характеристику, расширяющую другую характеристику:

```
/// Нечто в игровом мире: игрок или какой-нибудь эльф, горгулья, белка, орг и т. д.
trait Creature: Visible {
    fn position(&self) -> (i32, i32);
    fn facing(&self) -> Direction;
    ...
}
```

Фраза `trait Creature: Visible` означает, что все создания видимые. Любой тип, реализующий `Creature`, должен также реализовать характеристику `Visible`:

```
impl Visible for Broom {
    ...
}

impl Creature for Broom {
    ...
}
```

Мы можем реализовывать эти две характеристики в любом порядке, но будет ошибкой включить для какого-то типа реализацию `Creature`, опустив реализацию `Visible`.

Подхарактеристики аналогичны подынтерфейсам в Java или C#. Это способ описать характеристику, которая расширяет уже имеющуюся характеристику посредством добавления новых методов. В данном случае любой код, работающий с созданиями (значениями типа, реализующего `Creature`), может пользоваться методами, объявленными в характеристике `Visible`.

Статические методы

В большинстве объектно-ориентированных языков интерфейсы не могут содержать статических методов или конструкторов. Для характеристик в Rust это возможно:

```
trait StringSet {
    /// Возвращает новое пустое множество.
    fn new() -> Self;

    /// Возвращает множество, содержащее все строки из `strings`.
    fn from_slice(strings: &[&str]) -> Self;

    /// Определяет, содержит ли это множество конкретное значение `value`.
    fn contains(&self, string: &str) -> bool;

    /// Добавляет строку в это множество.
    fn add(&mut self, string: &str);
}
```

Любой тип, реализующий характеристику `StringSet`, должен реализовать эти четыре функции. Первые две, `new()` и `from_slice()`, не принимают аргумента `self`. Они играют роль конструкторов.

В неуниверсальном коде эти функции можно вызывать с помощью синтаксиса `::`, как любой другой статический метод:

```
// Создать множества двух гипотетических типов, реализующих StringSet:
let set1 = SortedStringSet::new();
let set2 = HashedStringSet::new();
```

В универсальном коде все обстоит так же, только в качестве типа часто употребляется параметрический тип, как при вызове `S::new()` ниже:

```
/// Возвращает множество слов в `document`, не входящих в `wordlist`.
fn unknown_words<S: StringSet>(document: &Vec<String>, wordlist: &S) -> S {
    let mut unknowns = S::new();
    for word in document {
        if !wordlist.contains(word) {
            unknowns.add(word);
        }
    }
    unknowns
}
```

Как интерфейсы в Java и C#, объекты характеристик не поддерживают статических методов. Чтобы использовать объекты характеристик `&StringSet`, необходимо

изменить характеристику, добавив ограничение «where Self: Sized» в каждый статический метод:

```
trait StringSet {
    fn new() -> Self
        where Self: Sized;

    fn from_slice(strings: &[&str]) -> Self
        where Self: Sized;

    fn contains(&self, string: &str) -> bool;

    fn add(&mut self, string: &str);
}
```

Это ограничение говорит Rust, что объектам характеристики разрешается не поддерживать этот метод. Теперь объекты характеристики StringSet допустимы; они, правда, не поддерживают двух статических методов, но их можно создавать и использовать для вызова методов .contains() и .add(). Этот прием работает и для всех прочих методов, несовместимых с объектами характеристик. (Мы опустим довольно запутанное техническое объяснение того, почему это работает, но саму характеристику Sized рассмотрим в главе 13.)

Полностью квалифицированные вызовы методов

Метод – это просто частный случай функции. Следующие два вызова эквивалентны:

```
"hello".to_string()

str::to_string("hello")
```

Вторая форма выглядит в точности как вызов статического метода. Это работает, несмотря на то что метод to_string принимает аргумент self. Просто передайте self в качестве первого аргумента функции.

Поскольку to_string – метод стандартной характеристики ToString, его можно вызвать еще двумя способами:

```
ToString::to_string("hello")

<str as ToString>::to_string("hello")
```

Все четыре формы вызова в точности эквивалентны. Чаще всего мы пишем просто value.method(). Остальные формы называются *квалифицированными* вызовами метода. В них задается тип или характеристика, с которыми ассоциирован данный метод. В последней форме (с угловыми скобками) указано то и другое, она называется *полностью квалифицированным* вызовом метода.

Когда мы пишем "hello".to_string(), применяя оператор ., мы не говорим точно, какой метод to_string вызывается. В Rust имеется алгоритм поиска метода, который определяет нужный метод, опираясь на типы, применяя Deref-преобразования и т. д. Употребляя полностью квалифицированный вызов, мы точно указываем, какой метод имели в виду, это может помочь в некоторых неоднозначных ситуациях.

- Когда есть два одноименных метода. Классический искусственный пример – тип Outlaw (бандит) с двумя методами .draw(), взятыми из двух разных

характеристик: один рисует бандита на экране, а другой служит для выхватывания пистолета[1].

```
outlaw.draw(); // ошибка: нарисовать на экране или выхватить пистолет?

Visible::draw(&outlaw);   // ok: нарисовать на экране
HasPistol::draw(&outlaw); // ok: загнан в угол
```

Вообще-то, лучше один из методов назвать по-другому, но иногда такой возможности нет.

- Когда невозможно вывести тип `self`:

```
let zero = 0; // тип не указан; может быть `i8`, `u8`, ...

zero.abs(); // ошибка: не найден метод `abs`
i64::abs(zero); // ok
```

- Когда в качестве значения используется сама функция:

```
let words: Vec<String> =
    line.split_whitespace() // итератор порождает значения &str
        .map(<str as ToString>::to_string) // ok
        .collect();
```

Здесь полностью квалифицированный вызов `<str as ToString>::to_string` – это просто способ поименовать конкретную функцию, передаваемую `.map()`.

- При вызове методов характеристик в макросах. Мы вернемся к этому вопросу в главе 20.

Полностью квалифицированными могут быть также вызовы статических методов. В предыдущем разделе мы писали `S::new()` для создания нового множества внутри универсальной функции. Но можно было бы написать также `StringSet::new()` или `<S as StringSet>::new()`.

Характеристики, определяющие связи между типами

Все рассмотренные до сих пор характеристики были автономными: характеристика – это набор методов, которые могут быть реализованы типами. Но характеристики можно использовать также в ситуациях, когда имеется несколько типов, которые должны работать совместно.

- Характеристика `std::iter::Iterator` связывает тип итератора с типом порождаемого им значения.
- Характеристика `std::ops::Mul` связывает типы, которые можно перемножать. В выражении `a * b` значения `a` и `b` могут быть как одного, так и разных типов.
- Крейт `rand` включает характеристику для генераторов случайных чисел (`rand::Rng`) и характеристику для типов, которые могут генерироваться случайным образом (`rand::Rand`). Характеристики точно определяют, как должны совместно работать эти типы.

Вам вряд ли придется создавать такие характеристики ежедневно, но они встречаются в стандартной библиотеке и сторонних крейтах. В этом разделе мы покажем, как реализованы вышеупомянутые примеры, и попутно познакомимся

[1] В английском языке оба действия обозначаются глаголом draw. – *Прим. перев.*

с некоторыми дополнительными свойствами языка Rust. Главное здесь – умение читать характеристики и сигнатуры методов и понимать, какую информацию они несут об используемых типах.

Ассоциированные типы, или Как работают итераторы

Начнем с итераторов. В наши дни в любой объектно-ориентированный язык в каком-то виде встроена поддержка итераторов – объектов, представляющих обход последовательности значений.

В Rust имеется стандартная характеристика `Iterator`, определенная следующим образом:

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
    ...
}
```

В начале этой характеристики мы видим *ассоциированный тип* – `type Item;`. Каждый тип, реализующий `Iterator`, должен указать тип порождаемых элементов.

Следующий далее метод `next()` использует ассоциированный тип в типе возвращаемого значения. Он возвращает значение типа `Option<Self::Item>`: либо `Some(item)` – следующее значение в последовательности, либо `None`, если значений не осталось. Тип записывается в виде `Self::Item`, а не просто `Item`, поскольку `Item` – принадлежность каждого типа итератора, а не самостоятельный тип. Как всегда, `self` и тип `Self` явно указываются во всех местах, где используются их поля, методы и т. п.

Вот как реализуется характеристика `Iterator` некоторым типом:

```
// (код взят из стандартного библиотечного модуля std::env)
impl Iterator for Args {
    type Item = String;

    fn next(&mut self) -> Option<String> {
        ...
    }
    ...
}
```

`std::env::Args` – это тип итератора, возвращаемого стандартной библиотечной функцией `std::env::args()`, которой мы пользовались в главе 2 для доступа к аргументам командной строки. Итератор порождает значения типа `String`, поэтому в блоке `impl` объявлено `type Item = String;`.

В универсальном коде можно использовать ассоциированные типы:

```
/// Обход итератора в цикле с сохранением значений в новом векторе.
fn collect_into_vector<I: Iterator>(iter: I) -> Vec<I::Item> {
    let mut results = Vec::new();
    for value in iter {
        results.push(value);
    }
    results
}
```

В теле этой функции Rust самостоятельно выводит тип `value`, и это хорошо; но мы должны явно указать тип значения, возвращаемого `collect_into_vector`, а единственный способ сделать это – воспользоваться ассоциированным типом `Item` (`Vec<I>` было бы откровенно неправильно: это означало бы, что мы возвращаем вектор итераторов!).

Самостоятельно мы не стали бы писать такой код, поскольку из главы 15 узнаем, что у итераторов уже имеется стандартный метод, который делает в точности то же самое: `iter.collect()`. Поэтому, прежде чем двигаться дальше, рассмотрим еще один пример.

```
/// Распечатать все значения, порождаемые итератором
fn dump<I>(iter: I)
    where I: Iterator
{
    for (index, value) in iter.enumerate() {
        println!("{}: {:?}", index, value); // error
    }
}
```

Чтобы этот код можно было назвать корректным, не хватает одной малости: может оказаться, что тип `value` не допускает печати.

```
error[E0277]: the trait bound `<I as std::iter::Iterator>::Item:
              std::fmt::Debug` is not satisfied
  --> traits_dump.rs:10:37
   |
10 | println!("{}: {:?}", index, value); // error
   |                              ^^^^^ the trait `std::fmt::Debug` is not implemented
   |                                    for `<I as std::iter::Iterator>::Item`
   |
   = help: consider adding a `where <I as std::iter::Iterator>::Item:
           std::fmt::Debug` bound
   = note: required by `std::fmt::Debug::fmt`
```

Сообщение об ошибке выглядит несколько запутанным из-за использования конструкции `<I as std::iter::Iterator>::Item`, но это не более чем длинный, максимально явный способ записи `I::Item`. Синтаксически все правильно, но вам редко придется прибегать к такой форме записи.

Смысл же сообщения в том, что, для того чтобы эта функция компилировалась, мы должны гарантировать, что `I::Item` реализует характеристику `Debug`, которая позволяет форматировать значения по спецификатору `{:?}`. Для этого следует наложить на `I::Item` ограничение.

```
use std::fmt::Debug;

fn dump<I>(iter: I)
    where I: Iterator, I::Item: Debug
{
    ...
}
```

Или можно было бы написать, что «`I` должно быть итератором для обхода значений типа `String`».

```
fn dump<I>(iter: I)
    where I: Iterator<Item=String>
{
    ...
}
```

`Iterator<Item=String>` – это тоже характеристика. Если рассматривать `Iterator` как множество всех итераторных типов, то `Iterator<Item=String>` – подмножество `Iterator`, а точнее множество таких итераторных типов, которые порождают значения типа `String`. Этот синтаксис можно использовать всюду, где допустимо имя характеристики, в т. ч. в типах объектов характеристик:

```
fn dump(iter: &mut Iterator<Item=String>) {
    for (index, s) in iter.enumerate() {
        println!("{}: {:?}", index, s);
    }
}
```

Характеристики с ассоциированными типами, в частности `Iterator`, совместимы с наличием методов, но только в том случае, когда все ассоциированные типы указаны явно, как в примере выше. В противном случае тип `s` мог бы быть любым, и Rust никаким способом не смог бы проверить корректность употребления типов в таком коде.

Мы привели немало примеров с итераторами. Да и куда же без них, ведь именно в итераторах ассоциированные типы употребляются чаще всего. Но, вообще говоря, ассоциированные типы полезны всюду, где характеристика нужна не только для определения методов.

- В библиотеке пула потоков с характеристикой `Task`, представляющей единицу работы, может быть ассоциирован тип `Output`.
- С характеристикой `Pattern`, представляющей способ поиска в строке, может быть ассоциирован тип `Match`, который представляет всю информацию, собранную в результате сопоставления строки с образцом.

```
trait Pattern {
type Match;

    fn search(&self, string: &str) -> Option<Self::Match>;
}

/// Можно искать конкретный символ в строке.
impl Pattern for char {
    /// "Результат сопоставления" — это просто позиция найденного
    /// символа.
    type Match = usize;

    fn search(&self, string: &str) -> Option<usize> {
        ...
    }

}
```

Если вы знакомы с регулярными выражениями, то легко поймете, что в блоке `impl Pattern for RegExp` должен был бы присутствовать более развитый тип `Match` – возможно, структура, содержащая начало и конец сопоставленного участка, позиции сопоставленных групп в круглых скобках и т. д.

- В библиотеке для работы с реляционными базами данных могла бы присутствовать характеристика `DatabaseConnection` с ассоциированными типами, представляющими транзакции, курсоры, подготовленные команды и т. д.

Ассоциированные типы идеально подходят для случаев, когда в каждой реализации есть *один* конкретный связанный тип: каждый тип `Task` порождает конкретный тип `Output`; каждый тип `Pattern` ищет конкретный тип `Match`. Однако, как мы увидим, не вся связи между типами имеют такой вид.

Универсальные характеристики, или Как работает перегрузка операторов

Для умножения в Rust используется следующая характеристика:

```
/// std::ops::Mul, характеристика для типов, поддерживающих `*`.
pub trait Mul<RHS> {
    /// Тип значения, получающегося в результате применения оператора `*`
    type Output;

    /// Метод, соответствующий оператору `*`
    fn mul(self, rhs: RHS) -> Self::Output;
}
```

`Mul` – это универсальная характеристика с параметрическим типом `RHS` (акроним «right-hand side» – правая часть).

Здесь параметрический тип означает то же самое, что в структуре или функции: `Mul` – универсальная характеристика, а ее экземпляры – `Mul<f64>`, `Mul<String>`, `Mul<Size>` и т. д. – это различные характеристики, точно так же, как `min::<i32>` и `min::<String>` – различные функции, а `Vec<i32>` и `Vec<String>` – различные типы.

Один тип, например `WindowSize`, может реализовывать сразу две характеристики, `Mul<f64>` и `Mul<i32>`, а то и намного больше. Тогда мы сможем умножать `WindowSize` на много разных типов. У каждой реализации будет свой собственный ассоциированный тип `Output`.

В показанной выше характеристике не хватает одной детали. В действительности характеристика `Mul` выглядит так:

```
pub trait Mul<RHS=Self> {
    ...
}
```

Синтаксис `RHS=Self` означает, что `RHS` по умолчанию совпадает с `Self`. Если бы я написал `impl Mul for Complex`, не задав явно параметрического типа `Mul`, то имелось бы в виду `impl Mul<Complex> for Complex`. Если бы я написал в ограничении `where T: Mul`, то это означало бы `where T: Mul<T>`.

В Rust выражение `lhs * rhs` – сокращенная запись для `Mul::mul(lhs, rhs)`. Поэтому для перегрузки оператора `*` достаточно просто реализовать характеристику `Mul`. Примеры будут приведены в следующей главе.

Парные характеристики, или Как работает rand::random()

Существует еще один способ использования характеристик для выражения связей между типами. Пожалуй, он самый простой из всех, потому что для его понимания не придется изучать никакие дополнительные языковые средства. Это так

называемые «парные характеристики» (buddy traits), которые проектируются для совместной работы.

Хороший пример имеется в крейте `rand`, предназначенном для генерации случайных чисел. Главное в этом крейте – функция `random()`, возвращающая случайное значение:

```
use rand::random;
let x = random();
```

Если, как часто бывает, Rust не может вывести тип случайного значения, то мы должны указать его сами:

```
let x = random::<f64>();   // число, 0.0 <= x < 1.0
let b = random::<bool>();  // true или false
```

Для многих программ достаточно одной этой универсальной функции. Но в крейте `rand` есть несколько разных, но совместимых между собой генераторов случайных чисел. Все они реализуют общую характеристику:

```
/// Генератор случайных чисел.
pub trait Rng {
    fn next_u32(&mut self) -> u32;
    ...
}
```

`Rng` – это просто штука, которая может по запросу выдавать целые числа. Библиотека `rand` предоставляет несколько реализаций, в т. ч. `XorShiftRng` (быстрый генератор псевдослучайных чисел) и `OsRng` (работает гораздо медленнее, но действительно порождает непредсказуемые числа, предназначен для использования в криптографии).

Парная характеристика называется `Rand`:

```
/// Тип, значения которого можно случайным образом сгенерировать с помощью `Rng`.
pub trait Rand: Sized {
    fn rand<R: Rng>(rng: &mut R) -> Self;
}
```

Эту характеристику реализуют, например, типы `f64` и `bool`. Передав любой генератор случайных чисел их методу `::rand()`, мы получим случайное значение соответствующего типа.

```
let x = f64::rand(rng);
let b = bool::rand(rng);
```

На самом деле `random()` – всего лишь тонкая обертка, которая передает глобально выделенный генератор `Rng` этому методу `rand`. Вот один из возможных способов реализации:

```
pub fn random<T: Rand>() -> T {
    T::rand(&mut global_rng())
}
```

Видя характеристики, которые используют другие характеристики в качестве ограничений, как `Rand::rand()` использует `Rng`, мы понимаем, что обе характеристики комбинируются: любой генератор `Rng` может генерировать значения любо-

го типа Rand. Поскольку задействованные методы универсальны, Rust генерирует оптимизированный машинный код для каждой комбинации Rng и Rand, встречающейся в программе.

Эти характеристики, кроме всего прочего, способствуют разделению обязанностей. Реализуя Rand для типа Monster или фантастически быстрый, но не особенно случайный Rng, мы не должны специально что-то делать, для того чтобы эти две части кода могли работать совместно.

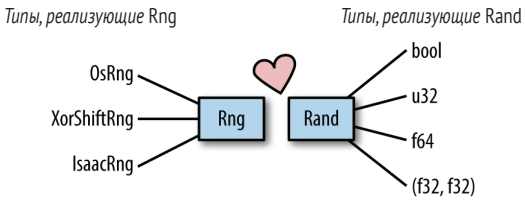


Рис. 11.3 ❖ Иллюстрация парных характеристик. Типы Rng слева – это настоящие генераторы случайных чисел, реализованные в крейте rand

Вычисление хеш-кодов в стандартной библиотеке – еще один пример использования парных характеристик. Типы, реализующие характеристику Hash, являются хешируемыми, т. е. их можно использовать в качестве ключей хеш-таблицы. А типы, реализующие характеристику Hasher, – это алгоритмы хеширования. Те и другие связаны так же, как Rand и Rng: в Hash определен универсальный метод Hash::hash(), который принимает любой тип Hasher в качестве аргумента.

Еще один пример – характеристика Serialize из библиотеки serde, с которой мы встречались в разделе «Характеристики и сторонние типы» выше в этой главе. У нее есть парная характеристика, о которой мы еще не говорили: Serializer, представляющая формат вывода. Библиотека serde поддерживает сменные форматы сериализации. Существуют реализации Serializer для JSON, YAML, двоичного формата CBOR и т. д. Благодаря тесной связи между обеими характеристиками любой формат автоматически поддерживает любой сериализуемый тип.

В трех предыдущих разделах мы показали три варианта описания связей между типами с помощью характеристик. Все их можно рассматривать как способы избежать понижающего приведения и накладных расходов, сопряженных с виртуальными методами, поскольку благодаря им Rust знает конкретные типы на этапе компиляции.

Обратное конструирование ограничений

Написание универсального кода может оказаться нудной и кропотливой работой, если нет одной готовой характеристики, делающей все, что нужно. Допустим, мы написали неуниверсальную функцию, выполняющую некоторое вычисление:

```
fn dot(v1: &[i64], v2: &[i64]) -> i64 {
    let mut total = 0;
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
```

```
    }
    total
}
```

А затем хотим использовать тот же код для чисел с плавающей точкой. Можно попробовать сделать следующее:

```
fn dot<N>(v1: &[N], v2: &[N]) -> N {
    let mut total: N = 0;
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

Увы – Rust ругается на использование `+` и `*` и тип значения `0`. Можно потребовать, чтобы тип `N` поддерживал операторы `+` и `*` с помощью характеристик `Add` и `Mul`. Но при этом нужно отказаться от использования `0`, поскольку в Rust `0` – всегда целое число, а соответствующее число с плавающей точкой записывается в виде `0.0`. По счастью, существует стандартная характеристика `Default` для тех типов, у которых есть «значение по умолчанию». Для числовых типов значение по умолчанию всегда равно 0.

```
use std::ops::{Add, Mul};

fn dot<N: Add + Mul + Default>(v1: &[N], v2: &[N]) -> N {
    let mut total = N::default();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

Уже ближе, но еще не работает:

```
error[E0308]: mismatched types
  --> traits_generic_dot_2.rs:11:25
    |
11 | total = total + v1[i] * v2[i];
   |                 ^^^^^^^^^^^^^ expected type parameter, found associated type
   |
   = note: expected type `N`
              found type `<N as std::ops::Mul>::Output`
```

В новом варианте кода предполагается, что умножение двух значений типа `N` порождает новое значение типа `N`. Но это вовсе не обязательно. Оператор умножения можно перегрузить, так что он будет возвращать значение любого типа. Необходимо как-то сказать Rust, что эта универсальная функция работает только с типами, для которых умножение определено «естественным» способом, т. е. результатом `N * N` является значение типа `N`. Для этого заменим `Mul` на `Mul<Output=N>` и так же поступим для `Add`.

```
fn dot<N: Add<Output=N> + Mul<Output=N> + Default>(v1: &[N], v2: &[N]) -> N
{
    ...
}
```

Как видим, ограничений становится все больше, из-за чего код трудно читать. Перенесем ограничения во фразу `where`:

```
fn dot<N>(v1: &[N], v2: &[N]) -> N
    where N: Add<Output=N> + Mul<Output=N> + Default
{
    ...
}
```

Отлично. Но Rust все равно ругается на следующую строчку:

```
error[E0508]: cannot move out of type `[N]`, a non-copy array
  --> traits_generic_dot_3.rs:7:25
   |
7  | total = total + v1[i] * v2[i];
   |                 ^^^^^ cannot move out of here
```

Вот уж действительно загадка, хотя мы уже знакомы со всей терминологией. Ну да, передавать владение значением `v1[i]` от срезки кому-то еще было бы неправильно, но ведь числа допускают копирование. Так в чем же дело?

В том, что *Rust не знает*, что `v1[i]` – число. На самом деле это и не число вовсе – `N` может быть любым типом, удовлетворяющим наложенным ограничениям. Если мы хотим, чтобы тип `N` был копируемым, то должны так и сказать:

```
where N: Add<Output=N> + Mul<Output=N> + Default + Copy
```

Вот теперь код компилируется и работает. Окончательный вариант выглядит так:

```
use std::ops::{Add, Mul};

fn dot<N>(v1: &[N], v2: &[N]) -> N
    where N: Add<Output=N> + Mul<Output=N> + Default + Copy
{
    let mut total = N::default();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}

#[test]
fn test_dot() {
    assert_eq!(dot(&[1, 2, 3, 4], &[1, 1, 1, 1]), 10);
    assert_eq!(dot(&[53.0, 7.0], &[1.0, 5.0]), 88.0);
}
```

Такое иногда случается в Rust: после длительных пререканий с компилятором код наконец приобретает элегантность, так что даже странно, почему же мы сразу так не написали.

Проделанное выше можно назвать обратным конструированием ограничений на `N`, когда компилятор используется как наставник и контролер. Причина, по которой возиться пришлось так долго, состоит в том, что в стандартной библиотеке нет одной характеристики `Number`, которая включала бы все необходимые нам операторы и методы. Зато есть популярный крейт с открытым исходным кодом `num`,

в котором такая характеристика имеется! Знай мы о нем, могли бы добавить `num` в файл `Cargo.toml` и написать:

```
use num::Num;

fn dot<N: Num + Copy>(v1: &[N], v2: &[N]) -> N {
    let mut total = N::zero();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

В объектно-ориентированном программировании для получения хорошего кода нужен правильный интерфейс, а в универсальном – правильная характеристика.

Но все же – к чему все эти сложности? Почему бы проектировщикам Rust не сделать универсальные типы и функции больше похожими на шаблоны C++, когда ограничения остаются в коде неявно, как в случае «утиной типизации»?

Одно из преимуществ подхода Rust – прямая совместимость универсального кода. Мы можем изменить реализацию открытой универсальной функции или метода, и если сигнатура осталась прежней, то весь ранее написанный код будет работать.

Другое преимущество ограничений состоит в том, что в случае ошибки компилятор хотя бы может сказать, в чем проблема. В C++ сообщения об ошибках при работе с шаблонами гораздо длиннее, чем в Rust, и указывают на много разных строк кода, потому что компилятор не может сказать, кто виноват: сам шаблон, или вызвавший его код, который тоже может быть шаблоном, или код, вызвавший *тот* код...

Но, пожалуй, самое главное достоинство явного выписывания характеристик заключается просто в том, что они присутствуют – в коде и в документации. Достаточно взглянуть на сигнатуру универсальной функции в Rust, как мы сразу понимаем, какие аргументы она принимает. О шаблонах такого не скажешь. Для полного документирования типов аргументов в таких библиотеках на C++, как Boost, придется поработать еще *усерднее*, чем показано выше. У разработчиков Boost нет компилятора, который стал бы проверять их действия.

Заключение

Характеристики – один из основных механизмов организации кода в Rust, и тому есть основательные причины. Нет ничего лучше, чем проектировать программу или библиотеку вокруг хорошего интерфейса.

В этой главе мы столкнулись с круговертью синтаксических конструкций, правил и объяснений. Но теперь, заложив фундамент, мы можем приступить к разговору о многочисленных способах использования характеристик и универсальных типов и функций в программах на Rust. Пока что мы прошлись только по верхам. В следующих двух главах рассматриваются наиболее употребительные характеристики, включенные в стандартную библиотеку. Мы обсудим замыкания, ввод-вывод и конкурентность. Характеристики, а также универсальные типы и функции во всех этих случаях играют главенствующую роль.

Глава 12

Перегрузка операторов

> Среди математиков и философов существуют разные мнения о предмете и определении математики. ...Ни одна точка зрения не является безупречной, ни одна не получила всеобщего признания, и не видно никаких перспектив для примирения.
>
> — *Википедия «Математика»*

В программе рисования множества Мандельброта, показанной в главе 2, мы воспользовались типом `Complex` из крейта `num`, чтобы представить число на комплексной плоскости:

```
#[derive(Clone, Copy, Debug)]
struct Complex<T> {
    /// Вещественная часть комплексного числа
    re: T,

    /// Мнимая часть комплексного числа
    im: T
}
```

Мы смогли складывать и умножать комплексные числа так же, как числа встроенных типов, применяя операторы `+` и `*`.

```
z = z * z + c;
```

Никто не мешает нам поддержать арифметические и другие операторы в своих собственных типах, для этого нужно только реализовать несколько встроенных характеристик. Это называется *перегрузкой операторов* и по достигнутому эффекту очень напоминает перегрузку операторов в C++, C#, Python и Ruby.

Характеристики, предназначенные для перегрузки операторов, разбиты на категории в зависимости от того, какую часть языка поддерживают (см. табл. 12.1). В последующих разделах мы поочередно рассмотрим все категории.

Таблица 12.1. Характеристики, предназначенные для перегрузки операторов

Категория	Характеристика	Оператор
Унарные операторы	`std::ops::Neg` `std::ops::Not`	`-x` `!x`

Арифметические операторы	`std::ops::Add`
	`std::ops::Sub`
	`std::ops::Mul`
	`std::ops::Div`
	`std::ops::Rem`
		`x + y`
		`x - y`
		`x * y`
		`x / y`
		`x % y`

Окончание табл. 12.1

Категория	Характеристика	Оператор

Поразрядные операторы	`std::ops::BitAnd`
	`std::ops::BitOr`
	`std::ops::BitXor`
	`std::ops::Shl`
	`std::ops::Shr`
		`x & y`
		`x | y`
		`x ^ y`
		`x << y`
		`x >> y`

Арифметические составные операторы присваивания	`std::ops::AddAssign`
	`std::ops::SubAssign`
	`std::ops::MulAssign`
	`std::ops::DivAssign`
	`std::ops::RemAssign`
		`x += y`
		`x -= y`
		`x *= y`
		`x /= y`
		`x %= y`

Поразрядные составные операторы присваивания	`std::ops::BitAndAssign`
	`std::ops::BitOrAssign`
	`std::ops::BitXorAssign`
	`std::ops::ShlAssign`
	`std::ops::ShrAssign`
		`x &= y`
		`x |= y`
		`x ^= y`
		`x <<= y`
		`x >>= y`

Сравнение	`std::cmp::PartialEq` `std::cmp::PartialOrd`	`x == y, x != y` `x < y, x <= y, x > y, x >= y`
Индексирование	`std::ops::Index` `std::ops::IndexMut`	`x[y], &x[y]` `x[y] = z, &mut x[y]`

Арифметические и поразрядные операторы

В Rust выражение `a + b` на самом деле является сокращенной формой записи `a.add(b)` – вызова метода `add`, определенного в стандартной библиотечной характеристике `std::ops::Add`. Все стандартные числовые типы Rust реализуют характеристику `std::ops::Add`. Чтобы выражение `a + b` работало и для комплексных чисел, в крейте `num` эта характеристика реализована также для типа `Complex`. Аналогичные характеристики существуют и для других операторов: `a * b` – сокращенная форма записи `a.mul(b)`, метода характеристики `std::ops::Mul`; характеристика `std::ops::Neg` связана с унарным оператором `-` и т. д.

Если вы захотите написать `z.add(c)`, то должны будете ввести характеристику `Add` в область видимости, чтобы ее метод стал видимым. После этого все арифметические операции сложения можно будет переписать в виде вызовов функций[1]:

```
use std::ops::Add;

assert_eq!(4.125f32.add(5.75), 9.875);
assert_eq!(10.add(20), 10 + 20);
```

А вот как выглядит определение `std::ops::Add`:

```
trait Add<RHS=Self> {
    type Output;
    fn add(self, rhs: RHS) -> Self::Output;
}
```

Иными словами, характеристика `Add<T>` выражает способность прибавить к себе значение типа `T`. Например, если мы захотим складывать значение своего типа со значениями типов `i32` и `u32`, то должны будем реализовать в своем типе харак-

---

[1] Возрадуйтесь, пишущие на Lisp! Выражение `<i32 as Add>::add` – это оператор `+` над типом `i32`, запомненный в качестве значения функции.

теристики `Add<i32>` и `Add<u32>`. Параметрический тип `RHS` по умолчанию совпадает с `Self`, так что при реализации сложения двух значений одного типа можно писать просто `Add`. Ассоциированный тип `Output` описывает результат сложения.

Например, чтобы можно было складывать значения типа `Complex<i32>`, тип `Complex<i32>` должен реализовать характеристику `Add<Complex<i32>>`. А поскольку мы складываем значения одного типа, достаточно написать просто `Add`:

```
use std::ops::Add;

impl Add for Complex<i32> {
    type Output = Complex<i32>;
    fn add(self, rhs: Self) -> Self {
        Complex { re: self.re + rhs.re, im: self.im + rhs.im }
    }
}
```

Разумеется, нет необходимости реализовывать `Add` отдельно для `Complex<i32>`, `Complex<f32>`, `Complex<f64>` и т. д. Все эти определения не отличались бы ничем, кроме типа, поэтому можно написать единственную универсальную реализацию при условии, что тип компонента комплексного числа поддерживает сложение:

```
use std::ops::Add;

impl<T> Add for Complex<T>
    where T: Add<Output=T>
{
    type Output = Self;
    fn add(self, rhs: Self) -> Self {
        Complex { re: self.re + rhs.re, im: self.im + rhs.im }
    }
}
```

Написав `where T: Add<Output=T>`, мы ограничили `T` типами, для которых определено сложение с собой, так что получается результат того же типа. Это разумное ограничение, но его можно было еще ослабить: характеристика `Add` не требует, чтобы оба операнда `+` были одного типа, и не налагает ограничений на тип результата. Поэтому в максимально общей реализации левый и правый операнды могли бы изменяться независимо, а тип результата определялся бы тем, что получается при сложении компонентов:

```
use std::ops::Add;

impl<L, R, O> Add<Complex<R>> for Complex<L>
    where L: Add<R, Output=O>
{
    type Output = Complex<O>;
    fn add(self, rhs: Complex<R>) -> Self::Output {
        Complex { re: self.re + rhs.re, im: self.im + rhs.im }
    }
}
```

Но на практике Rust обычно не поддерживает операций над смешанными типами. Поскольку параметрический тип `L` должен реализовывать `Add<R, Output=O>`, отсюда обычно следует, что типы `L`, `R` и `O` совпадают: просто для `L` можно подобрать

не так уж много типов, реализующих что-то иное. Так что в итоге максимально универсальная версия может оказаться не намного полезнее, чем более простая, показанная выше.

Встроенные в Rust характеристики для арифметических и поразрядных операторов можно разбить на три группы: унарные операторы, бинарные операторы и составные операторы присваивания. Внутри каждой группы характеристики и их методы по форме одинаковы, поэтому мы рассмотрим только по одному примеру из каждой группы.

Унарные операторы

Если не считать оператора разыменования `*`, который будет рассмотрен отдельно в разделе «Deref и DerefMut», то в Rust есть два допускающих настройку унарных оператора, они показаны в табл. 12.2.

Таблица 12.2. Встроенные характеристики для унарных операторов

Имя характеристики	Выражение	Эквивалентное выражение
`std::ops::Neg`	`-x`	`x.neg()`
`std::ops::Not`	`!x`	`x.not()`

Все числовые типы в Rust реализуют характеристику `std::ops::Neg`, соответствующему унарному оператору отрицания `-`; целые типы и тип `bool` реализуют еще характеристику `std::ops::Not`, соответствующему унарному оператору дополнения `!`. Существуют также реализации для ссылок на эти типы.

Отметим, что оператор `!` вычисляет логическое отрицание для значений типа `bool` и поразрядное дополнение (т. е. обращение отдельных битов) для целых чисел. Он выполняет функции сразу двух операторов, `!` and `~`, имеющихся в C и C++.

Определения этих характеристик просты:

```
trait Neg {
    type Output;
    fn neg(self) -> Self::Output;
}

trait Not {
    type Output;
    fn not(self) -> Self::Output;
}
```

Отрицание комплексного числа сводится к отрицанию его вещественной и мнимой частей. Ниже показано, как можно было бы написать универсальную реализацию отрицания для комплексных чисел:

```
use std::ops::Neg;

impl<T, O> Neg for Complex<T>
    where T: Neg<Output=O>
{
    type Output = Complex<O>;
    fn neg(self) -> Complex<O> {
        Complex { re: -self.re, im: -self.im }
    }
}
```

Бинарные операторы

В табл. 12.3 перечислены бинарные арифметические и поразрядные операторы в Rust, а также соответствующие им характеристики.

Таблица 12.3. Встроенные характеристики для бинарных операторов

Категория	Имя характеристики	Выражение	Эквивалентное выражение

Арифметические операторы	`std::ops::Add`
	`std::ops::Not`
	`std::ops::Mul`
	`std::ops::Div`
	`std::ops::Rem`
		`x + y`
		`x - y`
		`x * y`
		`x / y`
		`x % y`
			`x.add()`
			`x.sub()`
			`x.mul()`
			`x.div()`
			`x.rem()`

Поразрядные операторы	`std::ops::BitAnd`
	`std::ops::BitOr`
	`std::ops::BitXor`
	`std::ops::Shl`
	`std::ops::Shr`
		`x & y`
		`x | y`
		`x ^ y`
		`x << y`
		`x >> y`
			`x.bitand(y)`
			`x.bitor(y)`
			`x.bitxor(y)`
			`x.shl(y)`
			`x.shr(y)`

Все числовые типы в Rust реализуют арифметические операторы. Целые типы и тип `bool` реализуют поразрядные операторы. Существуют также реализации, которые принимают ссылки на эти типы в качестве одного или обоих операндов.

Все характеристики из этой группы имеют одинаковую форму. Ниже приведено определение характеристики `std::ops::BitXor`, соответствующей оператору `^`:

```
trait BitXor<RHS=Self> {
    type Output;
    fn bitxor(self, rhs: RHS) -> Self::Output;
}
```

В начале этой главы мы привели также определение характеристики `std::ops::Add` из этой категории и несколько примеров ее реализации.

Характеристики `Shl` и `Shr` немного отходят от этой схемы: параметрический тип `RHS` в них не совпадает по умолчанию с `Self`, поэтому тип правого операнда всегда надо указывать явно. Правый операнд операторов `<<` или `>>` – это величина сдвига в битах, она никак не связана с типом сдвигаемого значения.

Оператор `+` можно использовать для конкатенации строки `String` со срезкой `&str` или другой строкой. Однако Rust запрещает левому операнду оператора `+` иметь тип `&str`, чтобы воспрепятствовать построению длинных строк путем повторного присоединения небольших кусочков слева (время работы такого алгоритма квадратично зависит от конечной длины строки, что неприемлемо). В общем случае макрос `write!` лучше подходит для построения строки по частям; как это делается, мы рассмотрим в разделе «Дописывание и вставка текста» главы 17.

Составные операторы присваивания

Составное выражение присваивания имеет вид `x += y` или `x &= y`: оно принимает два операнда, производит над ними некоторую операцию, например сложение или поразрядное И, и сохраняет результат на месте левого операнда. В Rust значением составного выражения присваивания всегда является `()`, а не сохраненное значение.

Такие операторы есть во многих языках, и обычно они определяются как сокращенная запись выражений вида `x = x + y` или `x = x & y`. Но в Rust принят другой подход: `x += y` – это сокращенная запись вызова метода `x.add_assign(y)`, где `add_assign` – единственный метод характеристики `std::ops::AddAssign`:

```
trait AddAssign<RHS=Self> {
    fn add_assign(&mut self, RHS);
}
```

В табл. 12.4 перечислены все составные операторы присваивания в Rust, а также соответствующие им встроенные характеристики.

Таблица 12.4. Встроенные характеристики для составных операторов присваивания

Категория	Имя характеристики	Выражение	Эквивалентное выражение
Арифметические операторы	std::ops::AddAssign std::ops::NotAssign std::ops::MulAssign std::ops::DivAssign std::ops::RemAssign	x += y x -= y x *= y x /= y x %= y	x.add_assign() x.sub_assign() x.mul_assign() x.div_assign() x.rem_assign()
Поразрядные операторы	std::ops::BitAndAssign std::ops::BitOrAssign std::ops::BitXorAssign std::ops::ShlAssign std::ops::ShrAssign	x &= y x |= y x ^= y x <<= y x >>= y	x.bitand_assign(y) x.bitor_assign(y) x.bitxor_assign(y) x.shl_assign(y) x.shr_assign(y)

Все числовые типы в Rust реализуют арифметические составные операторы присваивания. Целые типы и тип `bool` реализуют поразрядные составные операторы присваивания.

Универсальная реализация характеристики `AddAssign` для типа `Complex` не вызывает затруднений:

```
use std::ops::AddAssign;

impl<T> AddAssign for Complex<T>
    where T: AddAssign<T>
{
    fn add_assign(&mut self, rhs: Complex<T>) {
        self.re += rhs.re;
        self.im += rhs.im;
    }
}
```

Встроенная характеристика для составного оператора присваивания абсолютно не зависит от встроенной характеристики соответственного бинарного оператора. Реализация `std::ops::Add` не означает автоматической реализации `std::ops::AddAssign`; если вы хотите, чтобы Rust разрешил употребление вашего типа в левой части оператора `+=`, то должны реализовать характеристику `AddAssign` самостоятельно.

Как и в случае бинарных характеристик `Shl` и `Shr`, характеристики `ShlAssign` и `ShrAssign` несколько отклоняются от схемы, действующей для других составных операторов присваивания: параметрический тип `RHS` в них по умолчанию не совпадает с `Self`, так что тип правого операнда всегда нужно указывать явно.

Сравнение на равенство

Операторы == and != в Rust представляют собой сокращенную запись обращений к методам eq и ne характеристики std::cmp::PartialEq:

```
assert_eq!(x == y, x.eq(&y));
assert_eq!(x != y, x.ne(&y));
```

Вот определение std::cmp::PartialEq:

```
trait PartialEq<Rhs: ?Sized = Self> {
    fn eq(&self, other: &Rhs) -> bool;
    fn ne(&self, other: &Rhs) -> bool { !self.eq(other) }
}
```

Поскольку у метода ne есть определение по умолчанию, то для реализации характеристики PartialEq нужно определить только метод eq. Ниже приведена ее полная реализация для типа Complex:

```
impl<T: PartialEq> PartialEq for Complex<T> {
    fn eq(&self, other: &Complex<T>) -> bool {
        self.re == other.re && self.im == other.im
    }
}
```

Иными словами, для любого типа компонент комплексного числа T, допускающего сравнение на равенство, это определение реализует сравнение для типа Complex<T>. В предположении, что где-то уже реализована характеристика std::ops::Mul для типа Complex, мы можем теперь написать:

```
let x = Complex { re: 5, im: 2 };
let y = Complex { re: 2, im: 5 };
assert_eq!(x * y, Complex { re: 0, im: 29 });
```

Реализации PartialEq почти всегда имеют такой вид: каждое поле левого операнда сравнивается с соответственным полем правого операнда. Писать такой код скучно, а сравнение на равенство требуется часто, поэтому Rust по запросу может автоматически сгенерировать реализацию PartialEq. Нужно просто добавить PartialEq в атрибут derive определения типа:

```
#[derive(Clone, Copy, Debug, PartialEq)]
struct Complex<T> {
    ...
}
```

Автоматически сгенерированная реализация практически идентична написанному вручную коду, т. е. сравниваются все поля или элементы типа. Rust умеет также выводить реализации PartialEq для типов перечислений. Естественно, для этого все части типа (или перечисления) должны реализовывать PartialEq.

В отличие от арифметических и поразрядных характеристик, которые принимают операнды по значению, PartialEq принимает их по ссылке. Это означает, что при сравнении некопируемых значений, например String, Vec или HashMap, не происходит передачи владения, что было бы крайне неудобно:

```
let s = "d\x6fv\x65t\x61i\x6c".to_string();
let t = "\x64o\x76e\x74a\x69l".to_string();
assert!(s == t);  // s и t только заимствуются...

// ...поэтому сохраняют свои значения.
assert_eq!(format!("{} {}", s, t), "dovetail dovetail");
```

При этом на параметрический тип `Rhs` характеристики налагается ограничение, с которым мы еще не встречались:

```
where Rhs: ?Sized
```

Это ослабляет обычное требование Rust, согласно которому параметрические типы должны иметь известный размер, и позволяет определять характеристики вроде `PartialEq<str>` или `PartialEq<[T]>`. Методы `eq` и `ne` принимают параметров типа `&Rhs`, поэтому сравнение чего-то с `&str` или `&[T]` вполне разумно. Поскольку `str` реализует `PartialEq<str>`, то

```
assert!("ungula" != "ungulate");
```

Это эквивалентно вызову метода:

```
assert!("ungula".ne("ungulate"));
```

Здесь `Self` и `Rhs` имеют безразмерный тип `str`, поэтому параметры `self` и `rhs` метода `ne` оказываются значениями типа `&str`. Мы подробно обсудим размерные и безразмерные типы, а также характеристику `Sized` в разделе «Характеристика `Sized`» главы 13.

Почему эта характеристика называется `PartialEq`? Традиционное математическое определение «отношения эквивалентности», частным случаем которого является равенство, включает три условия:

- если `x == y` истинно, то `y == x` тоже должно быть истинно. Иными словами, результат не зависит от перестановки сторон сравнения на равенство;
- если `x == y` и `y == z`, то `x == z`. Если в цепочке значений каждое равно следующему, то все значения равны. Отношение равенства транзитивно;
- всегда `x == x`.

Последнее требование может показаться настолько очевидным, что и формулировать-то его не стоит, но именно здесь-то и зарыта собака. Типы Rust `f32` и `f64` – это значения с плавающей точкой, отвечающие стандарту IEEE. Согласно этому стандарту, выражения вида `0.0/0.0` и другие, которым нельзя сопоставить разумного значения, должны порождать специальное значение «не число», обычно обозначаемое `NAN`. Более того, согласно стандарту, значение `NAN` не должно быть равно никакому другому – в том числе себе самому. Так, стандарт диктует следующее поведение:

```
assert!(f64::is_nan(0.0/0.0));
assert_eq!(0.0/0.0 == 0.0/0.0, false);
assert_eq!(0.0/0.0 != 0.0/0.0, true);
```

Кроме того, любое сравнение на больше-меньше с `NAN` должно возвращать `false`:

```
assert_eq!(0.0/0.0 < 0.0/0.0, false);
assert_eq!(0.0/0.0 > 0.0/0.0, false);
```

```
assert_eq!(0.0/0.0 <= 0.0/0.0, false);
assert_eq!(0.0/0.0 >= 0.0/0.0, false);
```

Таким образом, оператор == в Rust удовлетворяет первым двум условиям в определении отношения эквивалентности, но, очевидно, не удовлетворяет третьему в применении к числам с плавающей точкой в стандарте IEEE. Это называется «частичным отношением эквивалентности», поэтому соответствующая встроенная характеристика и называется PartialEq. При написании универсального кода, для которого параметрические типы реализуют только характеристику PartialEq, мы можем предполагать, что первые два условия выполняются, но не должны считать, что всякое значение равно самому себе.

Это противоречит интуиции и может стать причиной ошибок, если потерять бдительность. Если вы предпочитаете потребовать в универсальном коде соблюдения всех условий эквивалентности, то можете наложить ограничение std::cmp::Eq: если тип реализует характеристику Eq, то для любого значения x этого типа гарантируется, что x == x. На практике почти все типы, реализующие PartialEq, реализуют и Eq; типы f32 и f64 – единственные исключения в стандартной библиотеке, которые реализуют PartialEq, но не Eq.

В стандартной библиотеке Eq определена как расширение PartialEq без добавления новых методов:

```
trait Eq: PartialEq<Self> { }
```

Если ваш тип реализует PartialEq и вы хотите, чтобы он реализовывал также Eq, то должны реализовать Eq явно, хотя определять новые функции или типы для этого не понадобится. Так, реализовать Eq для типа Complex можно очень быстро:

```
impl<T: Eq> Eq for Complex<T> { }
```

Мы могли бы сделать даже еще лаконичнее, просто включив Eq в атрибут derive в определении типа Complex:

```
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
struct Complex<T> {
    ...
}
```

Выведенные реализации универсального типа могут зависеть от параметрических типов. В случае атрибута derive выше тип Complex<i32> смог бы реализовать характеристику Eq, потому что ее реализует тип i32, но тип Complex<f32> реализовал бы только PartialEq, поскольку f32 не реализует Eq.

Когда мы реализуем std::cmp::PartialEq самостоятельно, Rust не может проверить, что наши определения методов eq и ne действительно ведут себя, как того требует полное или частичное отношение эквивалентности. Они могут делать что угодно. Rust верит нам на слово, что сравнение на равенство реализовано так, как ожидают пользователи характеристики.

Хотя в определении PartialEq имеется определение метода ne по умолчанию, при желании мы можем предоставить собственную реализацию. Но при этом надо гарантировать, что ne и eq всегда дают в точности противоположные результаты. Пользователи характеристики PartialEq предполагают именно такое поведение.

Сравнение на больше-меньше

Rust определяет поведение операторов <, >, <= и >= в терминах единственной характеристики, std::cmp::PartialOrd:

```
trait PartialOrd<Rhs = Self>: PartialEq<Rhs> where Rhs: ?Sized {
    fn partial_cmp(&self, other: &Rhs) -> Option<Ordering>;

    fn lt(&self, other: &Rhs) -> bool { ... }
    fn le(&self, other: &Rhs) -> bool { ... }
    fn gt(&self, other: &Rhs) -> bool { ... }
    fn ge(&self, other: &Rhs) -> bool { ... }
}
```

Отметим, что PartialOrd<Rhs> расширяет PartialEq<Rhs>: сравнение на больше-меньше возможно только для типов, которые можно сравнивать на равенство.

Единственный метод PartialOrd, который вы должны реализовать самостоятельно, – partial_cmp. Если partial_cmp возвращает Some(o), то o определяет, как self соотносится с other:

```
enum Ordering {
    Less,       // self < other
    Equal,      // self == other
    Greater,    // self > other
}
```

Если же partial_cmp возвращает None, значит, self и other несравнимы: ни одно не больше другого, и они не равны. Из всех примитивных типов Rust только сравнение значений с плавающей точкой может вернуть None: точнее, None возвращается при сравнении NAN («не число») с любым другим значением. О значениях NAN было рассказано в разделе «Сравнение на равенство» выше.

Как и для всех бинарных операторов, чтобы значения типов Left и Right можно было сравнивать, тип Left должен реализовать характеристику PartialOrd<Right>. Выражения вида x < y или x >= y представляют собой сокращенную форму вызова методов PartialOrd (см. табл. 12.5).

Таблица 12.5. Операторы сравнения на больше-меньше и методы PartialOrd

Выражение	Эквивалентный вызов метода	Определение по умолчанию
x < y	x.lt(y)	x.partial_cmp(&y) == Some(Less)
x > y	x.gt(y)	x.partial_cmp(&y) == Some(Greater)

x <= y	x.le(y)	match x.partial_cmp(&y) {
		  Some(Less) | Some(Equal) => true,
		  _ => false,
		}

x >= y	x.ge(y)	match x.partial_cmp(&y) {
		  Some(Greater) | Some(Equal) => true,
		  _ => false,
		}

Как и в предыдущих примерах, код в столбце «Эквивалентный вызов метода» подразумевает, что std::cmp::PartialOrd и std::cmp::Ordering находятся в области видимости.

Если известно, что значения двух типов всегда сравнимы между собой, то можно реализовать более строгую характеристику `std::cmp::Ord`:

```
trait Ord: Eq + PartialOrd<Self> {
    fn cmp(&self, other: &Self) -> Ordering;
}
```

Здесь метод `cmp` возвращает просто `Ordering`, а не `Option<Ordering>`, как метод `partial_cmp`: `cmp` либо говорит, что оба аргумента равны, либо сообщает, какой больше, а какой меньше. Почти все типы, реализующие `PartialOrd`, реализуют также и `Ord`. В стандартной библиотеке единственными исключениями из этого правила являются типы `f32` и `f64`.

Поскольку на множестве комплексных чисел не существует естественного порядка, мы не сможем воспользоваться типом `Complex` для демонстрации простой реализации `PartialOrd`. Вместо этого предположим, что мы работаем со следующим типом, который представляет множество чисел, попадающих в заданный полуоткрытый интервал:

```
#[derive(Debug, PartialEq)]
struct Interval<T> {
    lower: T, // включается
    upper: T  // не включается
}
```

Мы хотим ввести на множестве значений этого типа отношение частичного порядка: один интервал считается меньше другого, если он полностью содержится в нем. Два пересекающихся интервала несравнимы: в каждом из них существует элемент, меньший какого-то элемента другого интервала. Два одинаковых интервала равны. Эти правила поддерживает следующая реализация характеристики `PartialOrd`:

```
use std::cmp::{Ordering, PartialOrd};

impl<T: PartialOrd> PartialOrd<Interval<T>> for Interval<T> {
    fn partial_cmp(&self, other: &Interval<T>) -> Option<Ordering> {
        if self == other { Some(Ordering::Equal) }
        else if self.lower >= other.upper { Some(Ordering::Greater) }
        else if self.upper <= other.lower { Some(Ordering::Less) }
        else { None }
    }
}
```

При такой реализации мы можем написать:

```
assert!(Interval { lower: 10, upper: 20 } < Interval { lower: 20, upper: 40 });
assert!(Interval { lower: 7, upper: 8 } >= Interval { lower: 0, upper: 1 });
assert!(Interval { lower: 7,  upper: 8  } <= Interval { lower: 7,  upper: 8  });

// Пересекающиеся интервалы несравнимы между собой.
assert!(!(Interval { lower: 10, upper: 30 } < Interval { lower: 20, upper: 40 });
assert!(!(Interval { lower: 10, upper: 30 } >= Interval { lower: 20, upper: 40 });
```

Index и IndexMut

Мы можем указать, как индексное выражение вида `a[i]` работает для некоторого типа, реализовав характеристики `std::ops::Index` и `std::ops::IndexMut`. Массивы

поддерживают оператор [] непосредственно, но для всех прочих типов выражение a[i] обычно является сокращенной записью вызова *a.index(i), где index – метод характеристики std::ops::Index. Однако если такому выражению производится присваивание или заимствуется изменяемая ссылка на него, то a[i] интерпретируется как сокращенная запись вызова метода *a.index_mut(i) характеристики std::ops::IndexMut.

Ниже приведены определения обеих характеристик:

```
trait Index<Idx> {
    type Output: ?Sized;
    fn index(&self, index: Idx) -> &Self::Output;
}

trait IndexMut<Idx>: Index<Idx> {
    fn index_mut(&mut self, index: Idx) -> &mut Self::Output;
}
```

Отметим, что эти характеристики принимают в качестве параметра тип индексного выражения. Индексировать срезку можно с помощью одного значения типа usize, ссылающегося на единственный элемент, поскольку срезки реализуют характеристику Index<usize>. А на подсрезку можно сослаться с помощью выражения вида a[i..j], поскольку она реализует также характеристику Index<Range<usize>>. Это выражение является сокращенной записью такого вызова:

```
*a.index(std::ops::Range { start: i, end: j })
```

Коллекции HashMap и BTreeMap позволяют использовать в качестве индекса произвольную хеш-таблицу или упорядоченный тип. Следующий код работает, потому что тип HashMap<&str, i32> реализует характеристику Index<&str>:

```
use std::collections::HashMap;
let mut m = HashMap::new();
m.insert("十", 10);
m.insert("百", 100);
m.insert("千", 1000);
m.insert("万", 1_0000);
m.insert("億", 1_0000_0000);

assert_eq!(m["十"], 10);
assert_eq!(m["千"], 1000);
```

Эти индексные выражения эквивалентны таким:

```
use std::ops::Index;
assert_eq!(*m.index("十"), 10);
assert_eq!(*m.index("千"), 1000);
```

Ассоциированный с характеристикой Index тип Output определяет тип значения, порождаемого индексным выражением: в примере HashMap выше тип Output в реализации Index совпадает с i32.

Характеристика IndexMut расширяет Index, добавляя метод index_mut, который принимает изменяемую ссылку на self и возвращает изменяемую ссылку на значение типа Output. Rust автоматически выбирает index_mut, если индексное выражение встречается в требующем того контексте:

```
let mut desserts = vec!["Howalon".to_string(),
                        "Soan papdi".to_string()];
desserts[0].push_str(" (fictional)");
desserts[1].push_str(" (real)");
```

Поскольку метод `push_str` применяется к `&mut self`, последние две строки эквивалентны такому коду:

```
use std::ops::IndexMut;
(*desserts.index_mut(0)).push_str(" (fictional)");
(*desserts.index_mut(1)).push_str(" (real)");
```

У характеристики `IndexMut` имеется проектное ограничение: она должна возвращать изменяемую ссылку на некоторое значение. Поэтому мы не можем использовать выражение вида `m["十"] = 10;` для вставки значения в `HashMap m`: сначала таблица должна была бы создать запись с ключом `"十"` и каким-нибудь значением по умолчанию и вернуть изменяемую ссылку на нее. Но не у всех типов имеются значения по умолчанию, которые можно создать с минимальными накладными расходами, а иногда уничтожение значения не тривиально. И было бы расточительно создавать такое значение только для того, чтобы сразу уничтожить его вследствие присваивания. (В будущих версиях языка планируется улучшить эту ситуацию.)

Чаще всего индексирование применяется при работе с коллекциями. Пусть, например, мы имеем дело с растровыми изображениями наподобие созданных программой рисования множества Мандельброта в главе 2. Напомним, что в этой программе был такой код:

```
pixels[row * bounds.0 + column] = ...;
```

Было бы хорошо, если бы тип `Image<u8>` вел себя как двумерный массив, т. к. это позволило бы обращаться к пикселям, не занимаясь нудной арифметикой:

```
image[row][column] = ...;
```

Для этого следует объявить структуру:

```
struct Image<P> {
    width: usize,
    pixels: Vec<P>
}

impl<P: Default + Copy> Image<P> {
    /// Создать новое изображение заданного размера.
    fn new(width: usize, height: usize) -> Image<P> {
        Image {
            width,
            pixels: vec![P::default(); width * height]
        }
    }
}
```

А ниже приведены соответствующие реализации характеристик `Index` и `IndexMut`:

```
impl<P> std::ops::Index<usize> for Image<P> {
type Output = [P];
```

```
    fn index(&self, row: usize) -> &[P] {
        let start = row * self.width;
        &self.pixels[start .. start + self.width]
    }
}

impl<P> std::ops::IndexMut<usize> for Image<P> {
    fn index_mut(&mut self, row: usize) -> &mut [P] {
        let start = row * self.width;
        &mut self.pixels[start .. start + self.width]
    }
}
```

Результатом индексированного доступа к изображению является срезка пикселей, а индексирование срезки дает один конкретный пиксель.

Отметим, что если в выражении `image[row][column]` величина `row` выходит за границы массива, то наш метод `.index()` попытается проиндексировать `self.pixels` за границами диапазона, что вызовет панику. Именно так и должны вести себя реализации `Index` и `IndexMut`: доступ за границами диапазона обнаруживается и вызывает панику – точно так же, как при попытке доступа к элементу за границами массива, срезки или вектора.

Прочие операторы

Не все операторы в Rust можно перегружать. В версии Rust 1.17 оператор проверки ошибок `?` работает только со значениями типа `Result`. Аналогично логические операторы `&&` и `||` могут работать только с булевыми значениями. Оператор `..` всегда создает значения типа `Range`, оператор `&` заимствует ссылки, а оператор `=` всегда копирует значения или передает владение ими. Ни один из них не допускает перегрузки.

Оператор разыменования `*val` и оператор «точка», применяемый для доступа к полям и вызова методов, как в выражениях `val.field` и `val.method()`, можно перегружать посредством характеристик `Deref` и `DerefMut`, рассматриваемых в следующей главе. (Мы не включили их в эту главу, поскольку их назначение не сводится к одной лишь перегрузке нескольких операторов.)

Rust не поддерживает перегрузку оператора вызова функции, `f(x)`. Когда требуется значение, допускающее вызов, мы обычно просто пишем замыкание. В главе 1 мы расскажем, как это работает, и рассмотрим специальные характеристики `Fn`, `FnMut` и `FnOnce`.

Глава 13

Вспомогательные характеристики

> Наука – это не что иное, как стремление выявить единство в огромном разнообразии природы – или, точнее, в разнообразии нашего опыта. Поэзия, живопись и прочие искусства также занимаются, по словам Кольриджа, поиском единства в многообразии.
>
> *— Якоб Броновски*

Помимо перегрузки операторов, рассмотренной в предыдущей главе, встроенные характеристики позволяют модифицировать также работу других частей языка Rust и стандартной библиотеки.

- Характеристика `Drop` используется для очистки значений, покидающих область видимости, как деструкторы в C++.
- Интеллектуальные указатели, например `Box<T>` и `Rc<T>`, могут реализовать характеристику `Deref`, чтобы указатель повторял методы обернутого значения.
- Посредством реализации характеристик `From<T>` и `Into<T>` мы можем подсказать Rust, как преобразовывать значение из одного типа в другой.

Эта глава представляет собой сборную солянку из различных полезных характеристик, включенных в стандартную библиотеку Rust. В табл. 13.1 перечислены все характеристики, которые мы собираемся рассмотреть.

В стандартной библиотеке есть и другие важные характеристики. В главе 5 мы рассмотрим характеристики `Iterator` и `IntoIterator`, в главе 16 – характеристику `Hash`, применяемую для вычисления хеш-кодов, а в главе 19 – пару характеристик `Send` и `Sync`, помечающих потокобезопасные типы.

Таблица 13.1. Вспомогательные характеристики

Характеристика	Описание
`Drop`	Деструкторы. Код очистки, который Rust автоматически выполняет при уничтожении значения
`Sized`	Маркерная характеристика для типов, размер которых известен на этапе компиляции, – в противоположность типам (например, срезкам), размер которых определяется динамически
`Clone`	Типы, поддерживающие клонирование значений
`Copy`	Маркерная характеристика для типов, клонирование которых сводится к побайтовому копированию области памяти

Окончание табл. 13.1

Характеристика	Описание
`Deref` и `DerefMut`	Характеристики для типов интеллектуальных указателей
`Default`	Типы, имеющие разумное «значение по умолчанию»
`AsRef` и `AsMut`	Преобразовательные характеристики для заимствования одного типа ссылки у другого
`Borrow` и `BorrowMut`	Преобразовательные характеристики, похожие на `AsRef` и `AsMut`, но дополнительно гарантирующие согласованное хеширование, упорядочение и равенство
`From` и `Into`	Преобразовательные характеристики для преобразования значений из одного типа в другой
`ToOwned`	Преобразовательная характеристика для преобразования ссылки в значение, имеющее владельца

Характеристика Drop

Когда владелец значения перестает существовать, мы говорим, что Rust *уничтожает* (drop) значение. В процессе уничтожения освобождаются другие значения, память в куче и системные ресурсы, принадлежащие данному значению. Уничтожение происходит по различным причинам: когда переменная выходит из области видимости, когда значение выражения отбрасывается оператором `;`, при усечении вектора в результате удаления последних элементов и т. д.

Как правило, Rust обрабатывает уничтожение значений автоматически. Рассмотрим, к примеру, такой тип:

```
struct Appellation {
    name: String,
    nicknames: Vec<String>
}
```

Значение типа `Appellation` владеет памятью в куче, выделенной для хранения содержимого строк и буфера вектора. Rust позаботится об очистке этой памяти при уничтожении `Appellation`, и вам для этого ничего писать не надо. Но при желании можно настроить процесс уничтожения значений определенного вами типа, реализовав в нем характеристику `std::ops::Drop`:

```
trait Drop {
    fn drop(&mut self);
}
```

Реализация `Drop` аналогична написанию деструктора в C++ или финализатора в других языках. Если уничтожаемое значение реализует `std::ops::Drop`, то Rust вызывает его метод `drop`, до того как приступить к уничтожению значений, которыми владеют его поля или элементы. Такое неявное обращение к `drop` – единственный способ вызвать этот метод; любую попытку вызвать его явно Rust считает ошибкой.

Поскольку Rust вызывает метод `Drop::drop` для значения, перед тем как уничтожить его поля или элементы, значение, которое получает этот метод, все еще полностью инициализировано. Поэтому в реализации характеристики `Drop` для нашего типа `Appellation` можно пользоваться всеми его полями:

```
impl Drop for Appellation {
    fn drop(&mut self) {
```

```
        print!("Уничтожается {}", self.name);
        if !self.nicknames.is_empty() {
            print!(" (АКА {})", self.nicknames.join(", "));
        }
        println!("");
    }
}
```

Имея эту реализацию, мы можем написать такой код:

```
{
    let mut a = Appellation { name: "Зевс".to_string(),
                              nicknames: vec!["тучегонитель".to_string(),
                                              "царь богов".to_string()] };

    println!("до присваивания");
    a = Appellation { name: "Гера".to_string(), nicknames: vec![] };
    println!("в конце блока");
}
```

Когда переменной `a` присваивается второе значение `Appellation`, первое уничтожается, а когда мы покидаем область видимости `a`, уничтожается и второе. Этот код печатает следующие сообщения:

```
до присваивания
Уничтожается Зевс (АКА тучегонитель, царь богов)
В конце блока
Уничтожается Гера
```

Поскольку наша реализация `std::ops::Drop` для типа `Appellation` не делает ничего, кроме печати сообщения, то как же очищается память? Тип `Vec` реализует характеристику `Drop`, при этом уничтожается каждый элемент вектора, а затем освобождается память, выделенная в куче для буфера. Тип `String` использует `Vec<u8>` для хранения текста, поэтому `String` может и не реализовывать `Drop` самостоятельно, а просто положиться на `Vec` для освобождения символов. Тот же принцип применим и к значениям типа `Appellation`: когда такое значение уничтожается, за освобождение содержимого всех строк и затем буфера, в котором хранятся элементы вектора, отвечает реализация `Drop` в типе `Vec`. Что же касается памяти, занятой самим значением `Appellation`, то у нее есть какой-то владелец, быть может, локальная переменная или структура данных, который несет ответственность за ее освобождение.

Если переменная кому-то передается, то она деинициализируется при выходе из области действия, поэтому Rust не будет пытаться ее уничтожить, ведь в ней нет никакого значения.

Этот принцип действует даже тогда, когда в зависимости от пути выполнения владение значением переменной иногда передается, а иногда нет. В таких случаях Rust отслеживает состояние переменной с помощью невидимого флага, показывающего, нужно ли уничтожать значение:

```
let p;
{
    let q = Appellation { name: "Cardamine hirsuta".to_string(),
                          nicknames: vec!["жеруха обыкновенная".to_string(),
                                          "водяной кресс".to_string()] };
```

```
    if complicated_condition() {
        p = q;
    }
}
println!("Блин! Что это было?");
```

В зависимости от того, что возвращает функция `complicated_condition` – `true` или `false`, владельцем `Appellation` будет одна из переменных `p` или `q`, а другая останется неинициализированной. От этого зависит, будет ли значение уничтожено до или после `println!`, поскольку `q` выходит из области видимости раньше `println!`, а `p` – позже. И хотя значение можно передавать из одного места в другое, уничтожает его Rust только один раз.

Обычно самостоятельно реализовывать `std::ops::Drop` имеет смысл только тогда, когда определяемый тип владеет ресурсами, о которых Rust ничего не знает. Например, в стандартной библиотеке для Unix-систем имеется следующий тип, представляющий файловый дескриптор в операционной системе:

```
struct FileDesc {
    fd: c_int,
}
```

Поле `fd` в этой структуре – просто номер дескриптора, который должен быть закрыт, когда программа заканчивает работу с ним; тип `c_int` – псевдоним `i32`. В стандартной библиотеке характеристика `Drop` реализована для типа `FileDesc` следующим образом:

```
impl Drop for FileDesc {
    fn drop(&mut self) {
        let _ = unsafe { libc::close(self.fd) };
    }
}
```

Здесь `libc::close` – имя, под которым Rust известна функция `close` из библиотеки C. Написанная на Rust программа может вызывать C-функции только внутри блоков `unsafe`, что мы здесь и наблюдаем.

Если некоторый тип реализует `Drop`, то он не может реализовывать характеристику `Copy`. По определению для копируемого типа независимую копию значения можно создать побайтовым копированием. Но обычно вызов одного и того же метода `drop` для одних и те же данных более одного раза является ошибкой.

Стандартная прелюдия включает функцию `drop` для уничтожения значения, но ее определение магическим никак не назовешь:

```
fn drop<T>(_x: T) { }
```

Иными словами, функция получает аргумент по значению, принимая на себя владение им от вызывающей стороны, а затем ничего с ним не делает. Rust уничтожает значение `_x`, когда оно покидает область видимости, как поступил бы для любой другой переменной.

Характеристика Sized

Размерным (sized) называется тип, все значения которого имеют одинаковый размер в памяти. Почти все типы в Rust размерные: любое значение типа `u64` за-

нимает восемь байтов, значение типа `(f32, f32, f32)` – двенадцать байтов. Даже перечисления размерные: какой бы вариант ни присутствовал в данный момент, перечисление всегда занимает столько места, сколько необходимо для хранения самого большого варианта. И хотя `Vec<T>` владеет выделенным в куче буфером, размер которого может изменяться, само значение `Vec` состоит из указателя на буфер, емкости и длины буфера, т. е. тип `Vec<T>` размерный.

Однако в Rust есть также *безразмерные* (unsized) типы, значения которых имеют разные размеры. Например, безразмерным является тип срезки строки `str` (обратите внимание на отсутствие `&`). Строковые литералы `"diminutive"` и `"big"` – ссылки на срезки `str`, занимающие десять и три байта соответственно. Оба показаны на рис. 13.1. Тип срезки массива `[T]` (снова без `&`) также безразмерный: разделяемая ссылка вида `&[u8]` может указывать на срезку `[u8]` любого размера. Поскольку типы `str` и `[T]` обозначают множества значений переменного размера, они являются безразмерными.

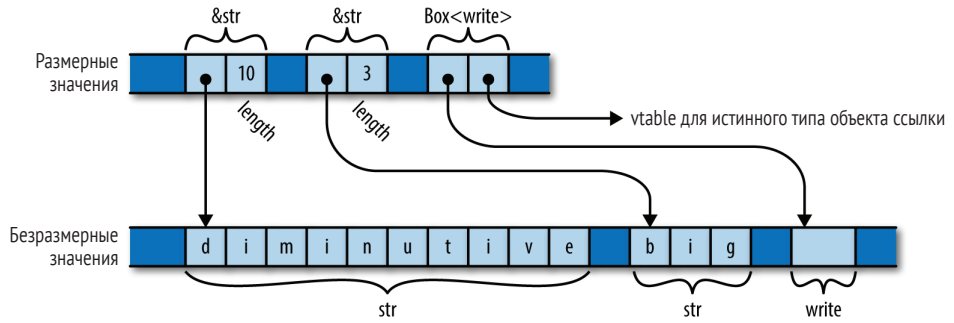


Рис. 13.1 ❖ Ссылки на безразмерные значения

Еще одна часто встречающаяся разновидность безразмерного типа – ссылка на объект характеристики. Как было сказано выше, объект характеристики – это указатель на некоторое значение, реализующее данную характеристику. Например, типы `&std::io::Write` и `Box<std::io::Write>` – указатели на некоторое значение, реализующее характеристику `Write`. Объектом ссылки может быть файл, сетевой сокет или ваш собственный тип, реализующий `Write`. Поскольку множество типов, реализующих `Write`, не фиксировано, характеристика `Write`, рассматриваемая как тип, безразмерна: ее значения имеют переменные размеры.

Rust не может хранить безразмерные значения в переменных или передавать их в качестве аргументов. Работать с ними можно только через указатели вида `&str` или `Box<Write>`, которые уже являются размерными. Как показано на рис. 13.1, указателем на безразмерное значение всегда является «толстый указатель» шириной два слова: указатель на срезку включает длину срезки, а объект характеристики – указатель на таблицу виртуальных методов.

Объекты характеристик и указатели на срезки в каком-то смысле симметричны. В обоих случаях типу недостает информации для полноценного использования: невозможно индексировать срезку `[u8]`, не зная ее длины, и невозможно вызвать метод `Box<Write>`, не зная реализации `Write`, соответствующей конкретному значению, на которое указывает бокс. И в обоих случаях толстый указатель

привносит информацию, отсутствующую в самом типе: длину или указатель на vtable. Недостающая статическая информация заменяется динамической.

Все размерные типы реализуют характеристику `std::marker::Sized`, в которой нет ни методов, ни ассоциированных типов. Rust автоматически реализует ее для всех типов, к которым она применима; реализовать ее самостоятельно невозможно. Единственное применение `Sized` – в качестве ограничения для параметрических типов: ограничение вида `T: Sized` означает, что размер типа `T` должен быть известен на этапе выполнения. Характеристики такого вида называются маркерными, поскольку сам язык использует их, для того чтобы пометить типы, обладающие некоторыми интересными свойствами.

Поскольку функциональность безразмерных типов слишком узкая, большинство параметрических типов должно реализовывать `Sized`. На самом деле это требуется так часто, что является неявным умолчанием в Rust: когда мы пишем `struct S<T> { ... }`, Rust считает, что имеется в виду `struct S<T: Sized> { ... }`. Если вы не хотите накладывать на `T` такое ограничение, то его необходимо отменить явно, написав `struct S<T: ?Sized> { ... }`. Синтаксис `?Sized` относится только к этому случаю и означает «необязательно `Sized`». Например, если написать `struct S<T: ?Sized> { b: Box<T> }`, то Rust позволит определить типы `S<str>` и `S<Write>`, для которых бокс становится толстым указателем, а также типы `S<i32>` и `S<String>`, для которых бокс – обычный указатель.

Несмотря на все ограничения безразмерных типов, благодаря им система типов в Rust работает более органично. В документации по стандартной библиотеке иногда можно встретить ограничение `?Sized` на переменную типа; почти всегда это означает, что на данный тип можно только указывать и что соответствующему коду позволено работать не только с обычными значениям, но и со срезками и объектами характеристик. Если на тип наложено ограничение `?Sized`, то говорят, что он «возможно, размерный»: может быть, `Sized`, а может быть, и нет.

Помимо срезок и объектов характеристик, есть еще одна разновидность безразмерных типов. Последнее поле структуры (но только последнее) может быть безразмерным, и такая структура сама является безразмерной. Например, тип указателя с подсчетом ссылок `Rc<T>` реализован как указатель на закрытый тип `RcBox<T>`, в котором хранятся счетчик ссылок и значение типа `T`. Вот упрощенное определение `RcBox`:

```
struct RcBox<T: ?Sized> {
    ref_count: usize,
    value: T,
}
```

Поле `value` – значение типа `T`, на которое `Rc<T>` подсчитывает ссылки; результатом разыменования `Rc<T>` является указатель на это поле. А в поле `ref_count` хранится счетчик ссылок.

Тип `RcBox` можно использовать вместе с размерными типами, например `RcBox<String>`, результатом будет размерный структурный тип. Но можно и с безразмерными типами, например `RcBox<std::fmt::Display>` (где `Display` – характеристика для типов, допускающих форматирование макросом `println!` и ему подобными); `RcBox<Display>` – безразмерный структурный тип.

Значение типа `RcBox<Display>` создать непосредственно невозможно. Вместо этого нужно сначала создать обыкновенный размерный `RcBox`, для которого `value` име-

ет тип, реализующий характеристику Display, например RcBox<String>. Затем Rust позволит преобразовать ссылку &RcBox<String> в толстую ссылку &RcBox<Display>:

```
let boxed_lunch: RcBox<String> = RcBox {
    ref_count: 1,
    value: "lunch".to_string()
};

use std::fmt::Display;
let boxed_displayable: &RcBox<Display> = &boxed_lunch;
```

Это преобразование производится неявно при передаче значений функциям, так что &RcBox<String> можно передать функции, ожидающей получить &RcBox<Display>:

```
fn display(boxed: &RcBox<Display>) {
    println!("For your enjoyment: {}", &boxed.value);
}

display(&boxed_lunch);
```

В результате будет напечатана строка:

```
For your enjoyment: lunch
```

Характеристика Clone

Характеристика std::clone::Clone предназначена для типов, которые могут создавать собственные копии. Она определена следующим образом:

```
trait Clone: Sized {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) {
        *self = source.clone()
    }
}
```

Метод clone должен сконструировать и вернуть независимую копию self. Поскольку метод возвращает значение типа Self, а функция не может возвращать значения безразмерного типа, то характеристика Clone расширяет Sized: по сути дела, это означает, что на типы Self в реализациях наложено ограничение Sized.

Клонирование значения обычно подразумевает выделение памяти для всего, чем это значение владеет, так что метод clone может оказаться накладным с точки зрения времени и памяти. Так, при клонировании Vec<String> копируется не только сам вектор, но и все его элементы типа String. Поэтому Rust не клонирует значения автоматически, а требует явного вызова метода. Типы указателей с подсчетом ссылок, например Rc<T> и Arc<T>, составляют исключения: клонирование такого указателя приводит только к увеличению счетчика ссылок.

Метод clone_from модифицирует self, превращая его в копию source. У него есть определение по умолчанию, которое просто клонирует source и передает результат *self. Это решение работает всегда, но для некоторых типов получить тот же эффект можно быстрее. Предположим, например, что s и t имеют тип String. Предложение s = t.clone(); должно клонировать t, уничтожить старое значение s

и затем передать в s клонированное значение. Получается одно выделение памяти из кучи и одно освобождение памяти. Но если буфер в куче, принадлежащий исходному s, достаточно велик и может вместить содержимое t, то выделять и освобождать память необязательно – можно просто скопировать текст t в буфер s и правильно установить длину. В универсальном коде следует всюду, где возможно, использовать метод clone_from, чтобы такие оптимизации не оставались втуне.

Если в вашей реализации Clone метод clone просто применяется к каждому полю или элементу типа, а затем из полученных клонов конструируется новое значение и определение clone_from по умолчанию приемлемо, то Rust готов реализовать Clone самостоятельно: просто снабдите определение типа атрибутом #[derive(Clone)].

Почти все стандартные библиотечные типы, которые имеет смысл копировать, реализуют характеристику Clone. К ним относятся как примитивные типы bool и i32, так и контейнерные типы String, Vec<T>, HashMap и т. д. Некоторые типы копировать бессмысленно, например std::sync::Mutex; они и не реализуют Clone. Такие типы, как std::fs::File, копировать, в принципе, можно, но операция копирования может завершиться неудачно, если у операционной системы недостаточно ресурсов. Подобные типы не реализуют Clone, потому что метод clone всегда должен завершаться успешно. Вместо этого тип std::fs::File предлагает метод try_clone, возвращающий значение типа std::io::Result<File>, способное сообщить об ошибке.

Характеристика Copy

В главе 4 мы говорили, что для большинства типов операция присваивания передает значение, а не копирует его. Передача значений упрощает отслеживание ресурсов, которыми они владеют. Но в разделе «Копируемые типы: исключения из правила передачи владения» мы отметили одно исключение: простые типы, не владеющие ресурсами, могут быть копируемыми, т. е. в процессе присваивания создается копия исходного значения, вместо того чтобы передавать его и оставлять исходное значение неинициализированным.

Тогда мы не стали уточнять, что такое копируемый тип, но теперь можем открыть тайну: тип называется копируемым, если реализует маркерную характеристику std::marker::Copy, определенную следующим образом:

```
trait Copy: Clone { }
```

Реализовать ее для собственных типов совсем просто:

```
impl Copy for MyType { }
```

но, поскольку Copy – маркерная характеристика, имеющая специальный смысл для языка, Rust разрешает типу реализовать ее, только если поверхностное побайтовое копирование – все, что необходимо. Типы, владеющие какими-то другими ресурсами, например буферами в куче или описателями объектов операционной системы, не могут реализовывать Copy.

Никакой тип, реализующий характеристику Drop, не может быть копируемым. Rust предполагает, что если типу необходим специальный код очистки, значит,

ему необходим также специальный код копирования, поэтому характеристика `Copy` для него не подходит.

Как и в случае `Clone`, мы можем попросить Rust реализовать `Copy` автоматически, добавив атрибут `#[derive(Copy)]`. Часто тип помечается обоими атрибутами: `#[derive(Copy, Clone)]`.

Хорошенько подумайте, прежде чем делать тип копируемым. Хотя использовать такие типы проще, на реализацию налагаются серьезные ограничения. Неявное копирование к тому же может обходиться дорого. Подробно эти факторы рассматриваются в разделе «Копируемые типы: исключения из правила передачи владения» главы 4.

Характеристики Deref и DerefMut

Чтобы описать поведение операторов `*` и `.` для своего типа, следует реализовать характеристики `std::ops::Deref` и `std::ops::DerefMut`. Указательные типы, такие как `Box<T>` и `Rc<T>`, реализуют их, так чтобы поведение было аналогично поведению встроенных указателей. Например, если имеется значение `b` типа `Box<Complex>`, то `*b` ссылается на то значение типа `Complex`, на которое указывает `b`, и `b.re` – на его вещественную часть. Если контекст подразумевает присваивание или заимствование изменяемой ссылки на объект ссылки, то Rust использует характеристику `DerefMut` («изменяемое разыменование»), в противном случае доступа для чтения достаточно и используется характеристика `Deref`.

Эти характеристики определены следующим образом:

```
trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}

trait DerefMut: Deref {
    fn deref_mut(&mut self) -> &mut Self::Target;
}
```

Методы `deref` и `deref_mut` принимают ссылку типа `&Self` и возвращают ссылку типа `&Self::Target`. `Target` должно быть чем-то, что `Self` содержит, чем владеет или на что ссылается: для типа `Box<Complex>` тип `Target` совпадает с `Complex`. Отметим, что `DerefMut` расширяет `Deref`: если мы можем что-то разыменовать и модифицировать, то, очевидно, должны также иметь возможность позаимствовать разделяемую ссылку на это. Поскольку методы возвращают ссылку с таким же временем жизни, как у `&self`, `self` продолжает оставаться заимствованной в течение всего времени жизни возвращенной ссылки.

У характеристик `Deref` и `DerefMut` есть и еще одна роль. Поскольку метод `deref` принимает ссылку типа `&Self`, а возвращает ссылку типа `&Self::Target`, то Rust использует его для автоматического преобразования ссылок из первого типа во второй. Иными словами, если вставка вызова `deref` способна предотвратить несоответствие типов, то Rust вставит его. Реализация `DerefMut` открывает возможность аналогичного преобразования для изменяемых ссылок. Это называется «Deref-преобразованием».

Хотя Deref-преобразования не являются чем-то таким, что мы не могли бы сами написать явно, они все же дают ряд удобств:

- если мы имеем значение `r` типа `Rc<String>` и хотим применить к нему метод `String::find`, то можно написать просто `r.find('?')`, а не `(*r).find('?')`: вызов метода неявно заимствует `r`, и `&Rc<String>` преобразуется в `&String`, поскольку `Rc<T>` реализует характеристику `Deref<Target=T>`;
- к значениям `String` можно применять метод `split_at` и ему подобные, хотя это метод типа срезки `str`, потому что тип `String` реализует характеристику `Deref<Target=str>`. Нет необходимости заново реализовывать все методы `str` для строк `String`, потому что существует Deref-преобразование из `&String` в `&str`;
- если имеется байтовый вектор `v` и мы хотим передать его функции, ожидающей получить байтовую срезку `[u8]`, то можно просто передать `&v` в качестве аргумента, потому что `Vec<T>` реализует характеристику `Deref<Target=[T]>`.

Если необходимо, Rust может применить несколько Deref-преобразований подряд. Например, благодаря описанным выше преобразованиям мы можем применить метод `split_at` непосредственно к `Rc<String>`, потому что `&Rc<String>` разыменуется в `&String`, которое разыменуется в `&str`, а у этого типа имеется метод `split_at`.

Рассмотрим, к примеру, следующий тип:

```
struct Selector<T> {
    /// Элементы, доступные в этом селекторе.
    elements: Vec<T>,

    /// Индекс текущего элемента вектора `elements`. `Selector` ведет себя
    /// как указатель на текущий элемент.
    current: usize
}
```

Чтобы тип `Selector` вел себя, как написано в комментариях, мы должны реализовать для него характеристики `Deref` и `DerefMut`:

```
use std::ops::{Deref, DerefMut};

impl<T> Deref for Selector<T> {
    type Target = T;
    fn deref(&self) -> &T {
        &self.elements[self.current]
    }
}

impl<T> DerefMut for Selector<T> {
    fn deref_mut(&mut self) -> &mut T {
        &mut self.elements[self.current]
    }
}
```

Имея такие реализации, мы можем использовать `Selector` следующим образом:

```
let mut s = Selector { elements: vec!['x', 'y', 'z'],
                       current: 2 };

    // Поскольку `Selector` реализует `Deref`, оператор `*` можно использовать
    // для ссылки на его текущий элемент.
    assert_eq!(*s, 'z');

    // Проверяем, что 'z' - буквы, применяя метод самого типа `char` прямо
    // к `Selector` благодаря Deref-преобразованию.
```

```
    assert!(s.is_alphabetic());

    // Заменяем 'z' на 'w', присваивая значение объекту ссылки `Selector`.
    *s = 'w';

assert_eq!(s.elements, ['x', 'y', 'w']);
```

Характеристики `Deref` и `DerefMut` проектировались для реализации типов интеллектуальных указателей, в частности `Box`, `Rc` и `Arc`, а также типов, которые могут выступать в роли владельцев чего-то, что часто используется по ссылке: например, `Vec<T>` и `String` владеют значениями типа `[T]` и `str`. Не следует реализовывать `Deref` и `DerefMut` в некотором типе только для того, чтобы автоматически наделить его методами типа `Target`, подобно тому как методы базового класса C++ видны в методах его подклассов. Это не всегда работает в соответствии с ожиданиями, а если что-то сорвется, то ждите неприятных сюрпризов.

С Deref-преобразованиями связан один подвох: Rust применяет их для разрешения конфликта типов, а не для того, чтобы удовлетворить ограничениям на переменные типов. Например, следующий код работает правильно:

```
let s = Selector { elements: vec!["good", "bad", "ugly"],
                   current: 2 };

fn show_it(thing: &str) { println!("{}", thing); }
show_it(&s);
```

При вызове `show_it(&s)` Rust видит аргумент типа `&Selector<&str>` и параметр типа `&str`, находит реализацию `Deref<Target=str>` и переписывает код в виде `show_it(s.deref())`, как и положено.

Но если превратить `show_it` в универсальную функцию, то Rust внезапно отказывается сотрудничать:

```
use std::fmt::Display;
fn show_it_generic<T: Display>(thing: T) { println!("{}", thing); }
show_it_generic(&s);
```

Rust ругается:

```
error[E0277]: the trait bound `Selector<&str>: Display` is not satisfied
    |
542 |         show_it_generic(&s);
    |         ^^^^^^^^^^^^^^^ trait `Selector<&str>: Display` not satisfied
    |
```

Чушь какая-то: стоило сделать функцию универсальной, как ни с того ни с сего появилась ошибка! Да, сам тип `Selector<&str>` не реализует характеристику `Display`, но ведь он разыменовывается в тип `&str`, а тот-то точно реализует.

Поскольку мы передаем аргумент типа `&Selector<&str>`, а параметрический тип функции равен `&T`, то переменная типа `T` должна совпадать с `Selector<&str>`. Тогда Rust проверяет, удовлетворяется ли ограничение `T: Display`: поскольку он не применяет Deref-преобразований, чтобы удовлетворить ограничениям на переменные-типы, то эта проверка завершается неудачно.

Чтобы обойти эту проблему, можно указать преобразование явно с помощью оператора `as`:

```
show_it_generic(&s as &str);
```

Характеристика Default

У некоторых типов имеется напрашивающееся «значение по умолчанию»: вектор или строка по умолчанию пусты, число по умолчанию равно нулю, значение типа `Option` по умолчанию равно `None` и т. д. Подобные типы могут реализовать характеристику `std::default::Default`:

```
trait Default {
    fn default() -> Self;
}
```

Метод `default` просто возвращает новое значение типа `Self`. Для типа `String` реализация `Default` тривиальна:

```
impl Default for String {
    fn default() -> String {
        String::new()
    }
}
```

Все типы коллекций Rust – `Vec`, `HashMap`, `BinaryHeap` и т. д. – реализуют характеристику `Default`, так что метод `default` возвращает пустую коллекцию. Это полезно, когда мы хотим построить коллекцию значений, но при этом предоставить вызывающей стороне решение о том, какой именно должна быть эта коллекция. Например, метод `partition` характеристики `Iterator` разбивает значения, порождаемые итератором, на две коллекции, применяя замыкание, чтобы решить, в какую коллекцию отправлять каждое значение:

```
use std::collections::HashSet;
let squares = [4, 9, 16, 25, 36, 49, 64];
let (powers_of_two, impure): (HashSet<i32>, HashSet<i32>)
    = squares.iter().partition(|&n| n & (n-1) == 0);

assert_eq!(powers_of_two.len(), 3);
assert_eq!(impure.len(), 4);
```

Замыкание `|&n| n & (n-1) == 0` посредством битовой операции распознает степени двойки, и метод `partition` с его помощью порождает две коллекции `HashSet`. Но, конечно, `partition` может работать не только с `HashSet`; его можно использовать для порождения любой коллекции, лишь бы только тип коллекции реализовывал характеристику `Default` (чтобы можно было вначале создать пустую коллекцию) и характеристику `Extend<T>` (чтобы можно было добавить в коллекцию значение типа `T`). Тип `String` реализует `Default` и `Extend<char>`, так что можно написать:

```
let (upper, lower): (String, String)
    = "Great Teacher Onizuka".chars().partition(|&c| c.is_uppercase());
assert_eq!(upper, "GTO");
assert_eq!(lower, "reat eacher nizuka");
```

Еще одно распространенное применение `Default` – порождение значений по умолчанию для структур, представляющих большую коллекцию параметров, большинство которых изменяется редко. Например, крейт `glium` содержит интерфейс из Rust к мощной и сложной графической библиотеке OpenGL. Структура `glium::DrawParameters` содержит 22 поля, каждое из которых управляет некоторым

аспектом отрисовки графики средствами OpenGL. Функция `draw` ожидает структуру `DrawParameters` в качестве аргумента. Поскольку тип `DrawParameters` реализует характеристику `Default`, то мы можем создать такую структуру для передачи `draw`, заполнив только поля, которые отличаются от подразумеваемых по умолчанию:

```
let params = glium::DrawParameters {
    line_width: Some(0.02),
    point_size: Some(0.02),
    .. Default::default()
};

target.draw(..., &params).unwrap();
```

Здесь метод `Default::default()` вызывается для создания значения типа `DrawParameters`, инициализированного значениями всех полей по умолчанию, а затем конструкция `..` используется, чтобы создать новую структуру, в которой отличаются только поля `line_width` и `point_size`. Созданная структура передается методу `target.draw`.

Если тип `T` реализует характеристику `Default`, то стандартная библиотека автоматически реализует `Default` для типов `Rc<T>`, `Arc<T>`, `Box<T>`, `Cell<T>`, `RefCell<T>`, `Cow<T>`, `Mutex<T>` и `RwLock<T>`. Например, значением по умолчанию для типа `Rc<T>` будет `Rc`, указывающее на значение по умолчанию для типа `T`.

Если типы всех элементов кортежного типа реализуют `Default`, то ее реализует и сам кортежный тип – кортеж по умолчанию состоит из элементов, имеющих значения по умолчанию.

Rust автоматически не реализует `Default` для структурных типов, но если все поля структуры реализуют `Default`, то можно запросить автоматическую реализацию `Default` для структуры, добавив атрибут `#[derive(Default)]`.

Значением по умолчанию для любого типа `Option<T>` является `None`.

Характеристики AsRef и AsMut

Если тип реализует характеристику `AsRef<T>`, значит, можно эффективно заимствовать от него ссылку `&T`. Аналогичная характеристика `AsMut` предназначена для заимствования изменяемых ссылок. Вот их определения:

```
trait AsRef<T: ?Sized> {
    fn as_ref(&self) -> &T;
}

trait AsMut<T: ?Sized> {
    fn as_mut(&mut self) -> &mut T;
}
```

Например, тип `Vec<T>` реализует `AsRef<[T]>`, а тип `String` реализует `AsRef<str>`. Мы можем также заимствовать содержимое строки в виде массива байтов, так что `String` реализует еще и `AsRef<[u8]>`.

`AsRef` обычно применяется, чтобы повысить гибкость функций с точки зрения принимаемых ими аргументов. Например, функция `std::fs::File::open` объявлена следующим образом:

```
fn open<P: AsRef<Path>>(path: P) -> Result<File>
```

Функции `open` в действительности нужен аргумент типа `&Path`, представляющего путь в файловой системе. Но при такой сигнатуре `open` принимает все, от чего можно заимствовать ссылку `&Path`, т. е. значение любого типа, реализующего `AsRef<Path>`. К таким типам относятся `String` и `str`, типы строк, составляющие интерфейс с операционной системой: `OsString` и `OsStr`, и, конечно, `PathBuf` и `Path`; полный перечень см. в документации по библиотеке. Это и позволяет передавать `open` строковые литералы:

```
let dot_emacs = std::fs::File::open("/home/jimb/.emacs")?;
```

Все функции доступа к файловой системе из стандартной библиотеки принимают путь именно таким образом. С точки зрения вызывающей стороны это напоминает перегруженные функции в C++, хотя подход Rust к установлению приемлемости аргументов принципиально отличается.

Но на этом история не заканчивается. Строковый литерал имеет тип `&str`, однако характеристику `AsRef<Path>` реализует тип `str`, без `&`. Как объяснялось в разделе «Характеристики Deref и DerefMut» выше, Rust не пытается выполнить Deref-преобразований, чтобы удовлетворить ограничения на переменные типов, поэтому здесь они не помогут.

К счастью, в стандартной библиотеке имеется всеобъемлющая реализация:

```
impl<'a, T, U> AsRef<U> for &'a T
    where T: AsRef<U>,
          T: ?Sized, U: ?Sized
{
    fn as_ref(&self) -> &U {
        (*self).as_ref()
    }
}
```

Иными словами, для любых типов `T` и `U` если `T: AsRef<U>`, то также `&T: AsRef<U>`: нужно просто проследовать по ссылке и продолжать, как и выше. В частности, поскольку `str: AsRef<Path>`, то `&str: AsRef<Path>`. В некотором смысле мы получили нечто похожее на Deref-преобразование при проверке ограничений `AsRef` на переменные типов.

Можно было бы подумать, что если тип реализует `AsRef<T>`, то он должен также реализовывать `AsMut<T>`. Однако есть случаи, когда это не так. Например, мы отмечали, что тип `String` реализует характеристику `AsRef<[u8]>`; это имеет смысл, поскольку у каждой строки имеется байтовый буфер, к которому может быть полезно обращаться, как к двоичным данным. Но тип `String` еще и гарантирует, что эти байты образуют корректный Юникод-текст в кодировке UTF-8; если бы `String` реализовывал `AsMut<[u8]>`, то вызывающая сторона была бы вправе изменять байты строки произвольным образом, и тогда пропала бы уверенность в том, что строка содержит допустимый текст в кодировке UTF-8. Реализация характеристики `AsMut<T>` некоторым типом имеет смысл, только если `T` не может нарушить инвариантов этого типа.

Характеристики `AsRef` и `AsMut` довольно просты, но наличие стандартных универсальных характеристик для преобразования ссылок позволяет избежать неограниченного размножения более специальных преобразовательных характеристик. Воздержитесь от определения собственных характеристик `AsFoo`, если достаточно просто реализовать `AsRef<Foo>`.

Характеристики Borrow и BorrowMut

Характеристика `std::borrow::Borrow` похожа на `AsRef`: если тип реализует `Borrow<T>`, то его метод `borrow` фактически заимствует ссылку `&T`. Но `Borrow` налагает дополнительные ограничения: реализация `Borrow<T>` некоторым типом возможна, лишь если ссылка `&T` хешируется и сравнивается точно так же, как значение, от которого она позаимствована. (Rust это не гарантирует; это всего лишь документированное назначение характеристики.) Поэтому `Borrow` особенно полезна при работе с ключами в хеш-таблицах и деревьях, а также со значениями, которые могут хешироваться или сравниваться по другим причинам.

Эта особенность начинает играть роль, например, при заимствовании у строк `String`: тип `String` реализует характеристики `AsRef<&str>`, `AsRef<[u8]>` и `AsRef<Path>`, но хеш-коды этих трех целевых типов обычно различаются. Гарантируется лишь, что срезка `&str` хешируется в то же значение, что эквивалентная строка `String`, так что `String` реализует только `Borrow<str>`.

Определение `Borrow` совпадает с определением `AsRef`, изменяются только имена:

```
trait Borrow<Borrowed: ?Sized> {
    fn borrow(&self) -> &Borrowed;
}
```

Характеристика `Borrow` предназначена для специальной ситуации, возникающей при работе с универсальными хеш-таблицами и другими типами ассоциативных коллекций. Рассмотрим тип `std::collections::HashMap<String, i32>`, отображающий строки в числа. Ключами этой таблицы являются строки `String`, каждая запись владеет своим ключом. Какова должна быть сигнатура метода, который ищет запись в такой таблице? Вот первая попытка:

```
impl HashMap<K, V> where K: Eq + Hash
{
    fn get(&self, key: K) -> Option<&V> { ... }
}
```

Разумно: чтобы найти запись, нужно задать ключ подходящего типа. Но в данном случае `K` – это `String`, и такая сигнатура заставила бы передавать `String` по значению при каждом вызове `get`, а это расточительно. На самом деле хватило бы и ссылки на ключ:

```
impl HashMap<K, V> where K: Eq + Hash
{
    fn get(&self, key: &K) -> Option<&V> { ... }
}
```

Немного лучше, но теперь мы должны передавать ключ как `&String`, поэтому для поиска константной строки пришлось бы написать:

```
hashtable.get(&"twenty-two".to_string())
```

Нелепость какая-то: мы выделяем буфер для строки в куче и копируем в него текст, только чтобы его можно было позаимствовать как `&String`, затем передаем ссылку `get`, после чего она уничтожается.

Было бы вполне достаточно передать нечто, что можно хешировать и сравнивать с типом нашего ключа; `&str` подошла бы идеально. И вот последний вариант, который и находится в стандартной библиотеке:

```
impl HashMap<K, V> where K: Eq + Hash
{
    fn get<Q: ?Sized>(&self, key: &Q) -> Option<&V>
        where K: Borrow<Q>,
              Q: Eq + Hash
    { ... }
}
```

Иными словами, если можно заимствовать ключ записи как `&Q` и результирующая ссылка хешируется и сравнивается так же, как сам ключ, то, очевидно, `&Q` будет приемлемым типом ключа. Поскольку `String` реализует `Borrow<str>` и `Borrow<String>`, то эта последняя версия `get` позволяет передавать в виде ключа как `&String`, так и `&str`, что и требуется.

Типы `Vec<T>` и `[T: N]` реализуют `Borrow<[T]>`. Любой подобный строке тип допускает заимствование соответствующего типа срезки: `String` реализует `Borrow<str>`, `PathBuf` реализует `Borrow<Path>` и т. д. И все стандартные библиотечные типы ассоциативных коллекций используют `Borrow`, когда нужно решить, какие типы передавать функциям поиска.

В стандартной библиотеке имеется всеобъемлющая реализация, чтобы любой тип мог заимствовать у себя же: `T: Borrow<T>`. Тем самым гарантируется, что `&K` всегда является допустимым типом для поиска записей в `HashMap<K, V>`.

В качестве дополнительного удобства всякий тип `&mut T` также реализует `Borrow<T>`, возвращая, как обычно, разделяемую ссылку `&T`. Это позволяет передавать изменяемые ссылки функциям поиска в коллекциях, не заставляя повторно заимствовать разделяемую ссылку; тем самым имитируется обычное для Rust неявное приведение изменяемых ссылок к разделяемым.

Характеристика `BorrowMut` – аналог `Borrow` для изменяемых ссылок:

```
trait BorrowMut<Borrowed: ?Sized>: Borrow<Borrowed> {
    fn borrow_mut(&mut self) -> &mut Borrowed;
}
```

Все, что мы ожидаем от `Borrow`, относится также к `BorrowMut`.

Характеристики From и Into

Характеристики `std::convert::From` и `std::convert::Into` представляют преобразования значения одного типа в другой. Если `AsRef` и `AsMut` заимствуют ссылку одного типа от другого, то `From` и `Into` принимают владение своим аргументом, преобразуют его и возвращают владение вызывающей стороне.

Их определения обладают приятной симметрией:

```
trait Into<T>: Sized {
    fn into(self) -> T;
}

trait From<T>: Sized {
    fn from(T) -> Self;
}
```

Стандартная библиотека автоматически реализует тривиальное преобразование каждого типа в себя же: любой тип `T` реализует характеристики `From<T>` и `Into<T>`.

Хотя эти характеристики дают два способа решить одну и ту же задачу, применяются они в разных ситуациях.

Характеристика `Into` обычно используется, чтобы сделать функцию более гибкой с точки зрения принимаемых аргументов. Например, приведенная ниже функция `ping`

```
use std::net::Ipv4Addr;
fn ping<A>(address: A) -> std::io::Result<bool>
    where A: Into<Ipv4Addr>
{
    let ipv4_address = address.into();
    ...
}
```

может принимать в качестве аргумента как `Ipv4Addr`, так и `u32` или массив `[u8; 4]`, поскольку последние два типа реализуют `Into<Ipv4Addr>`. (Иногда полезно рассматривать IPv4-адрес как одно 32-разрядное значение или как массив из четырех байтов.) Поскольку единственное, что `ping` знает об аргументе `address`, – это тот факт, что он реализует `Into<Ipv4Addr>`, нет необходимости указывать конечный тип при вызове `into`; существует лишь один подходящий тип, и механизм вывода типов его найдет.

Как и в случае характеристики `AsRef` из предыдущего раздела, достигаемый эффект очень напоминает перегрузку функций в C++. Показанное выше определение `ping` позволяет обращаться к ней любым из следующих способов:

```
println!("{:?}", ping(Ipv4Addr::new(23, 21, 68, 141))); // передать Ipv4Addr
println!("{:?}", ping([66, 146, 219, 98]));             // передать [u8; 4]
println!("{:?}", ping(0xd076eb94_u32));                 // передать u32
```

Но у характеристики `From` назначение иное. Метод `from` выступает в роли универсального конструктора для порождения экземпляра типа из некоторого другого значения. Например, вместо того чтобы заводить в типе `Ipv4Addr` два метода: `from_array` и `from_u32` – мы просто реализуем в этом типе характеристики `From<[u8;4]>` и `From<u32>`, что дает возможность писать:

```
let addr1 = Ipv4Addr::from([66, 146, 219, 98]);
let addr2 = Ipv4Addr::from(0xd076eb94_u32);
```

А какую реализацию применить, пусть решает механизм вывода типов.

При наличии подходящей реализации `From` стандартная библиотека автоматически реализует соответствующую характеристику `Into`. При определении собственного типа, имеющего конструкторы с одним аргументом, следует записывать их в виде реализаций `From<T>` для подходящих типов, тогда соответствующие реализации `Into` вы получите задаром.

Поскольку методы `from` и `into` принимают владение своими аргументами, преобразование может повторно воспользоваться ресурсами исходного значения для конструирования преобразованного. Рассмотрим такой код:

```
let text = "Beautiful Soup".to_string();
let bytes: Vec<u8> = text.into();
```

Реализация `Into<Vec<u8>>` для типа `String` просто берет буфер строки в куче и использует его в качестве буфера элементов возвращенного вектора, не внося ни-

каких изменений. Выделять новую память или копировать текст нет необходимости. Это еще один случай, когда передача владения открывает возможность для эффективной реализации.

Эти преобразования дают также изящный способ внести некоторую гибкость в значение ограниченного типа, не ослабляя гарантий, которые дают ограничения. Например, тип `String` гарантирует, что содержимое строки является допустимым текстом в кодировке UTF-8; изменяющие методы тщательно написаны так, чтобы ни в коем случае не получился недопустимый с точки зрения UTF-8 текст. Но в приведенном выше примере строка «понижается в ранге» до обычного байтового блока, с которым можно делать все что угодно, например сжать или скомбинировать с двоичными данными, не записанными в UTF-8. Поскольку `into` принимает аргумент по значению, `text` после преобразования оказывается неинициализированным, а это значит, что мы можем свободно обращаться к буферу бывшей строки, не опасаясь повредить существующую строку.

Однако низкие накладные расходы не являются частью контракта `Into` и `From`. Если от `AsRef` и `AsMut` мы ожидаем, что преобразование будет дешевым, то `From` и `Into` могут в процессе преобразования выделять память, копировать и обрабатывать содержимое значения любым другим способом. Например, тип `String` реализует характеристику `From<&str>`, которая копирует срезку строки в новый выделенный в куче буфер. А тип `std::collections::BinaryHeap<T>` реализует `From<Vec<T>>`, так что элементы сравниваются и переупорядочиваются в соответствии с требованиями алгоритма.

Отметим, что `From` и `Into` могут производить только такие преобразования, которые гарантированно завершаются успешно. Сигнатуры методов не позволяют сообщить об ошибке преобразования. Если преобразование в тип или из типа может завершиться ошибкой, то лучше написать функцию или метод, возвращающие значение типа `Result`.

До того как характеристики `From` и `Into` были добавлены в стандартную библиотеку, код на Rust был переполнен преобразовательными характеристиками и методами конструирования, рассчитанными на один конкретный тип. `From` и `Into` вводят соглашения, которым рекомендуется следовать, чтобы ваши типы было проще использовать, поскольку пользователи уже знакомы с этими соглашениями.

Характеристика ToOwned

Если имеется некоторая ссылка, то типичный способ породить принадлежащую вам копию ее объекта – вызвать метод `clone` в предположении, что тип реализует характеристику `std::clone::Clone`. Но что, если клонировать требуется `&str` или `&[i32]`? Вы, вероятно, хотите получить значение типа `String` или `Vec<i32>`, но определение `Clone` такого не допускает: по определению клонирование `&T` обязательно должно возвращать значение типа `T`, а типы `str` и `[u8]` безразмерные, так что никакая функция не может вернуть значение такого типа.

Характеристика `std::borrow::ToOwned` дает не столь ограничительный способ преобразования ссылки в принадлежащее вам значение:

```
trait ToOwned {
    type Owned: Borrow<Self>;
    fn to_owned(&self) -> Self::Owned;
}
```

В отличие от метода `clone`, который должен возвращать строго значение типа `Self`, метод `to_owned` может вернуть все, от чего можно заимствовать `&Self`: тип `Owned` должен реализовывать `Borrow<Self>`. Ссылку `&[T]` можно заимствовать от `Vec<T>`, поэтому `[T]` может реализовать `ToOwned<Owned=Vec<T>>`, коль скоро `T` реализует `Clone`, так что мы сможем скопировать элементы срезки в вектор. Аналогично `str` реализует `ToOwned<Owned=String>`, `Path` реализует `ToOwned<Owned=PathBuf>` и т. д.

Borrow и ToOwned за работой: скромное копирование при записи

Правильное использование Rust подразумевает обдумывание вопросов владения, например должна ли функция получить параметр по ссылке или по значению. Обычно можно остановиться на том или ином подходе, и тип параметра будет отражать принятое решение. Но в некоторых случаях невозможно решить, что делать – заимствовать или принимать во владение, – до этапа выполнения программы; тип `std::borrow::Cow` (акроним «**C**lone **o**n **w**rite» – копирование при записи) предлагает один из возможных подходов в таких случаях.

Вот его определение:

```
enum Cow<'a, B: ?Sized + 'a>
    where B: ToOwned
{
    Borrowed(&'a B),
    Owned(<B as ToOwned>::Owned),
}
```

Тип `Cow<B>` либо заимствует разделяемую ссылку на `B`, либо владеет значением, от которого можно заимствовать такую ссылку. Поскольку `Cow` реализует `Deref`, мы можем вызывать от его имени методы, как если бы это была разделяемая ссылка на `B`: если перечисление принимает значение `Owned`, значит, заимствована разделяемая ссылка на находящееся во владении значение, а если `Borrowed`, то просто отдается хранящаяся в перечислении ссылка.

Мы можем также получить изменяемую ссылку на значение `Cow`, вызвав метод `to_mut`, который возвращает `&mut B`. Если `Cow` принимает значение `Cow::Borrowed`, то `to_mut` просто вызывает метод ссылки `to_owned`, чтобы получить собственную копию ее объекта, затем заменяет значение `Cow` на `Cow::Owned` и заимствует изменяемую ссылку на новое владеемое значение. Это и есть поведение «копирования при записи», упоминаемое в имени типа.

В типе `Cow` есть также метод `into_owned`, который при необходимости получает из ссылки независимое значение, а затем возвращает его, передавая владение вызывающей стороне и потребляя по ходу дела `Cow`.

`Cow` часто применяется, когда нужно вернуть либо статически выделенную строковую константу, либо вычисленную строку. Предположим, к примеру, что требуется преобразовать перечисление, содержащее коды ошибок, в сообщение. В большинстве случаев сообщение является фиксированной строкой, но иногда содержит дополнительные данные. Тогда мы можем вернуть `Cow<'static, str>`:

```
use std::path::PathBuf;
use std::borrow::Cow;
```

```
fn describe(error: &Error) -> Cow<'static, str> {
    match *error {
        Error::OutOfMemory => "out of memory".into(),
        Error::StackOverflow => "stack overflow".into(),
        Error::MachineOnFire => "machine on fire".into(),
        Error::Unfathomable => "machine bewildered".into(),
        Error::FileNotFound(ref path) => {
            format!("file not found: {}", path.display()).into()
        }
    }
}
```

Здесь для конструирования значений используется реализация характеристики `Into` в типе `Cow`. В большинстве ветвей предложения `match` возвращается значение `Cow::Borrowed`, ссылающееся на статически выделенную строку. Но в ветви `FileNotFound` мы используем макрос `format!`, чтобы построить сообщение, включающее имя файла. Эта ветвь порождает значение `Cow::Owned`.

Если сторона, вызывающая `describe`, не собирается изменять значение, то может рассматривать `Cow` просто как `&str`:

```
println!("Случилась беда: {}", describe(&error));
```

Если же вызывающая сторона хочет получить значение во владение, то может легко это сделать:

```
let mut log: Vec<String> = Vec::new();
...
log.push(describe(&error).into_owned());
```

Использование `Cow` позволяет функции `describe` и вызывающим ее отложить выделение памяти до того момента, как в нем возникнет необходимость.

Глава 14

Замыкания

> Сохраните окружающую среду! Создайте замыкание сегодня же!
>
> — *Кормак Флэнаган*

Отсортировать вектор целых чисел просто.

```
integers.sort();
```

Но, увы, чаще приходится сортировать другие данные. Как правило, имеется множество каких-то записей, для которых встроенный метод `sort` не работает.

```
struct City {
    name: String,
    population: i64,
    country: String,
    ...
}

fn sort_cities(cities: &mut Vec<City>) {
    cities.sort();  // ошибка: и как их сортировать?
}
```

Rust жалуется, что тип `City` не реализует характеристику `std::cmp::Ord`. Мы должны задать порядок сортировки:

```
/// Вспомогательная функция для сортировки городов по численности населения.
fn city_population_descending(city: &City) -> i64 {
    -city.population
}

fn sort_cities(cities: &mut Vec<City>) {
    cities.sort_by_key(city_population_descending);  // ok
}
```

Вспомогательная функция `city_population_descending` принимает запись о городе `City` и извлекает из нее *ключ* – поле, по которому будут сортироваться данные. (Она возвращает отрицательное число, поскольку функция `sort` сортирует числа в порядке возрастания, а мы хотим сортировать в порядке убывания: сначала самый населенный город.) Метод `sort_by_key` принимает эту функцию в качестве параметра.

Такой подход работает, но получится короче, если записать вспомогательную функцию в виде *замыкания* – анонимной функции:

```
fn sort_cities(cities: &mut Vec<City>) {
    cities.sort_by_key(|city| -city.population);
}
```

Здесь замыканием является выражение `|city| -city.population`. Оно принимает в качестве аргумента `city` и возвращает `-city.population`. По контексту использования замыкания Rust выводит тип аргумента и тип возвращаемого значения.

Приведем еще примеры средств стандартной библиотеки, принимающих замыкания:

- методы характеристики `Iterator`, например `map` и `filter`, для работы с последовательными данными. Мы рассмотрим их в главе 15;
- потоковые API, например функция `thread::spawn`, запускающая новый системный поток. Смысл конкурентности состоит в том, чтобы передать работу другим потокам, а замыкания очень удобны для представления единиц работы. Эти средства рассматриваются в главе 19;
- некоторые методы, которые должны условно вычислять значение по умолчанию, например метод `or_insert_with` записей таблицы `HashMap`. Этот метод либо получает запись `HashMap`, либо создает ее; он используется, когда вычисление значения по умолчанию обходится дорого. Значение по умолчанию передается в виде замыкания, которое вычисляется только тогда, когда нужно создать новую запись.

В наши дни анонимные функции проникли повсюду, даже в языки Java, C#, Python и C++, где их первоначально не было. В дальнейшем мы будем предполагать, что вы с ними уже встречались, и сосредоточим внимание на особенностях замыканий в Rust. В этой главе вы научитесь применять замыкания вместе с методами из стандартной библиотеки, узнаете, как замыкание «запоминает» переменные из объемлющей области видимости, научитесь писать функции и методы, принимающие замыкания в качестве аргументов, и сохранять замыкания для последующего использования в качестве функций обратного вызова. Мы объясним, как работают замыкания в Rust и почему они быстрее, чем можно было бы ожидать.

Захват переменных

Замыкание может пользоваться данными, принадлежащими объемлющей функции, например:

```
/// Сортировать по одному из нескольких статистических показателей.
fn sort_by_statistic(cities: &mut Vec<City>, stat: Statistic) {
    cities.sort_by_key(|city| -city.get_statistic(stat));
}
```

Здесь в замыкании используется переменная `stat`, принадлежащая объемлющей функции `sort_by_statistic`. Мы говорим, что замыкание «захватывает» `stat`. Это одно из классических свойств замыканий, и, естественно, Rust его поддерживает, но в Rust к нему прилагается кое-что еще.

В большинстве языков, поддерживающих замыкания, важную роль играет сборка мусора. Рассмотрим, например, такой код на JavaScript:

```
// Начать анимацию, которая переупорядочивает строки в таблице городов.
function startSortingAnimation(cities, stat) {
```

```
    // Вспомогательная функция для сортировки таблицы.
    // Отметим, что она ссылается на stat.
    function keyfn(city) {
        return city.get_statistic(stat);
    }

    if (pendingSort)
        pendingSort.cancel();

    // Запускаем анимацию, передавая ей keyfn.
    // Алгоритм сортировки вызовет keyfn в нужный момент.
    pendingSort = new SortingAnimation(cities, keyfn);
}
```

Замыкание `keyfn` хранится в новом объекте `SortingAnimation`. Предполагается, что оно будет вызвано после возврата из `startSortingAnimation`. Обычно, когда функция возвращает управление, все локальные переменные и аргументы покидают область видимости и уничтожаются. Но в данном случае движок JavaScript должен как-то сохранить переменную `stat`, поскольку она используется в замыкании. В большинстве реализаций JavaScript для `stat` выделяется память в куче, а впоследствии ее освобождает сборщик мусора.

Но в Rust нет сборки мусора. И как же тогда это работает?

Чтобы ответить на этот вопрос, рассмотрим два разных примера.

Замыкания с заимствованием

Сначала повторим пример, открывавший этот раздел:

```
fn sort_by_statistic(cities: &mut Vec<City>, stat: Statistic) {
    cities.sort_by_key(|city| -city.get_statistic(stat));
}
```

В этом случае Rust, создавая замыкание, автоматически заимствует ссылку на `stat`. И это разумно: раз замыкание ссылается на аргумент `stat`, то оно должно иметь ссылку на него.

Остальное просто. Замыкание подчиняется всем правилам заимствования и времени жизни, которые были описаны в главе 5. В частности, поскольку замыкание содержит ссылку на `stat`, Rust не позволит ему пережить `stat`. Поскольку это замыкание используется только в процессе сортировки, то никаких трудностей не возникает.

Короче говоря, Rust гарантирует безопасность благодаря использованию времен жизни вместо сборки мусора. Это быстрее: даже самый быстрый сборщик мусора работает медленнее, чем сохранение `stat` в стеке, как делает Rust в этом случае.

Замыкания с кражей

Второй пример посложнее:

```
use std::thread;

fn start_sorting_thread(mut cities: Vec<City>, stat: Statistic)
    -> thread::JoinHandle<Vec<City>>
{
```

```
    let key_fn = |city: &City| -> i64 { -city.get_statistic(stat) };

    thread::spawn(|| {
        cities.sort_by_key(key_fn);
        cities
    })
}
```

Это больше похоже на то, что делается в примере на JavaScript: `thread::spawn` принимает замыкание и вызывает его в новом системном потоке. Отметим, что `||` обозначает пустой список аргументов замыкания.

Новый поток работает параллельно с вызывающим. Когда замыкание возвращает управление, новый поток завершается. (Возвращенное замыканием значение передается вызвавшему потоку в виде значения типа `JoinHandle`. Мы рассмотрим этот вопрос в главе 19.)

И снова замыкание `key_fn` содержит ссылку на `stat`. Но на этот раз Rust не может гарантировать безопасность использования ссылки и потому отклоняет программу:

```
error[E0373]: closure may outlive the current function, but it borrows `stat`,
              which is owned by the current function
  --> closures_sort_thread.rs:33:18
   |
33 | let key_fn = |city: &City| -> i64 { -city.get_statistic(stat) };
   |              ^^^^^^^^^^^^^^^^^^^^                       ^^^^
   |              | `stat` is borrowed here
   |              may outlive borrowed value `stat`
```

На самом деле здесь сразу две проблемы, потому что переменная `cities` тоже разделяется небезопасно. Все просто: нельзя ожидать, что новый поток, созданный методом `thread::spawn`, закончит работу раньше, чем переменные `cities` и `stat` будут уничтожены в конце функции.

Решение обеих проблем одно: потребовать, чтобы Rust *передал* переменные `cities` и `stat` замыканиям, в которых они используются, а не заимствовал ссылки на них.

```
fn start_sorting_thread(mut cities: Vec<City>, stat: Statistic)
    -> thread::JoinHandle<Vec<City>>
{
    let key_fn = move |city: &City| -> i64 { -city.get_statistic(stat) };

    thread::spawn(move || {
        cities.sort_by_key(key_fn);
        cities
    })
}
```

Изменилось только одно – мы добавили ключевое слово `move` перед обоими замыканиями. Это слово говорит Rust, что замыкание не заимствует используемые переменные, а крадет их.

Первое замыкание, `key_fn`, принимает владение переменной `stat`, а второе – владение переменными `cities` и `key_fn`.

Таким образом, Rust предлагает два способа получения замыканиями данных из объемлющей области видимости: передача и заимствование. И больше тут

сказать в общем-то нечего; замыкания следуют тем же правилам передачи и заимствования, которые рассматривались в главах 4 и 5. Отметим лишь несколько моментов:

- как и всюду в языке, если замыкание должно передать владение (с помощью `move`) значением копируемого типа, например `i32`, то вместо этого значение копируется. Так что если бы тип `Statistic` был копируемым, то мы могли бы использовать переменную `stat` даже после создания `move`-замыкания, в котором она захватывается;
- значения некопируемых типов, например `Vec<City>`, действительно передаются: приведенный выше код передает `cities` новому потоку, поскольку замыкание создано с ключевым словом `move`. После создания такого замыкания Rust не позволит обращаться к переменной `cities` по имени;
- в данном случае программе и не нужно использовать `cities` после передачи ее замыканию. Но если бы это было не так, то есть простой обходной путь: клонировать `cities` и сохранить копию в другой переменной. Замыкание украло бы только одну копию – ту, на которую ссылается.

Смирившись со строгими правилами Rust, мы приобрели нечто важное: потокобезопасность. Именно потому, что вектор передается другому потоку, а не разделяется двумя потоками, мы точно знаем, что старый поток не освободит вектора в процессе модификации его новым потоком.

Типы функций и замыканий

В этой главе мы не раз видели, как функции и замыкания используются в качестве значений. Следовательно, у них должен быть какой-то тип. Например:

```
fn city_population_descending(city: &City) -> i64 {
    -city.population
}
```

Эта функция принимает один аргумент (`&City`) и возвращает число типа `i64`. Ее тип: `fn(&City) -> i64`.

С функциями можно делать все то же, что с другими значениями. Их можно сохранять в переменных. Можно использовать обычный синтаксис Rust для вычисления значений функций.

```
let my_key_fn: fn(&City) -> i64 =
    if user.prefs.by_population {
        city_population_descending
    } else {
        city_monster_attack_risk_descending
    };

cities.sort_by_key(my_key_fn);
```

В структурах могут быть поля типа функций. Универсальные типы наподобие `Vec` могут содержать целые стада функций, лишь бы все они были одного типа. А значения типа функций занимают совсем немного места: это адрес машинного кода функции в памяти, совсем как указатель на функцию в C++.

Функция может принимать другую функцию в качестве аргумента, например:

```
/// Получив список городов и функцию проверки,
/// возвращает количество городов, прошедших проверку.
fn count_selected_cities(cities: &Vec<City>,
                         test_fn: fn(&City) -> bool) -> usize
{
    let mut count = 0;
    for city in cities {
        if test_fn(city) {
            count += 1;
        }
    }
    count
}

/// Пример функции проверки. Отметим, что эта функция имеет тип
/// `fn(&City) -> bool` - такой же, как функция, передаваемая
/// в аргументе `test_fn` функции `count_selected_cities`.
fn has_monster_attacks(city: &City) -> bool {
    city.monster_attack_risk > 0.0
}

// Сколько городов под угрозой атаки монстров?
let n = count_selected_cities(&my_cities, has_monster_attacks);
```

Те, кто знаком с указателями на функции в C/C++, понимают, что значения типа функций в Rust – в точности то же самое.

После этого кажется странным, что тип замыкания – *не* то же, что тип функции.

```
let limit = preferences.acceptable_monster_risk();
let n = count_selected_cities(
    &my_cities,
    |city| city.monster_attack_risk > limit);  // ошибка: несоответствие типов
```

Второй аргумент приводит к ошибке типизации. Для поддержки замыканий мы должны изменить сигнатуру этой функции на такую:

```
fn count_selected_cities<F>(cities: &Vec<City>, test_fn: F) -> usize
    where F: Fn(&City) -> bool
{
    let mut count = 0;
    for city in cities {
        if test_fn(city) {
            count += 1;
        }
    }
    count
}
```

Изменилась только сигнатура `count_selected_cities`, но не ее тело. Новая версия стала универсальной. Она принимает аргумент `test_fn` любого типа `F`, реализующего специальную характеристику `Fn(&City) -> bool`. Эта характеристика автоматически реализуется всеми функциями и замыканиями, которые принимают единственный аргумент типа `&City` и возвращают булево значение.

```
fn(&City) -> bool  // тип fn (только функции)
Fn(&City) -> bool  // характеристика Fn (функции и замыкания)
```

Этот специальный синтаксис встроен в язык. Символ `->` и тип возвращаемого значения необязательны; если они опущены, подразумевается, что возвращается значение типа `()`.

В новой версии функция `count_selected_cities` принимает как функцию, так и замыкание.

```
count_selected_cities(
    &my_cities,
    has_monster_attacks);  // правильно

count_selected_cities(
    &my_cities,
    |city| city.monster_attack_risk > limit);  // тоже правильно
```

Почему первая попытка оказалась неудачной? Потому что замыкание – это вызываемый объект, а не `fn`. У замыкания `|city| city.monster_attack_risk > limit` есть собственный тип, не совпадающий с типом `fn`.

На самом деле у любого написанного нами замыкания свой тип, поскольку замыкание может содержать данные: значения, заимствованные или украденные из объемлющей области видимости. Таких переменных может быть сколько угодно, с разными типами. Поэтому каждое замыкание имеет тип, созданный специально для него компилятором, достаточно большой для хранения всех данных. Не бывает двух замыканий с одинаковыми типами. Но всякое замыкание реализует характеристику `Fn`; в нашем примере замыкание реализует характеристику `Fn(&City) -> i64`.

Поскольку у каждого замыкания свой тип, код, предназначенный для работы с замыканиями, обычно должен быть универсальным, как функция `count_selected_cities`. Кажется, что всякий раз указывать универсальные типы, – корявое проектное решение, но у него есть преимущества. Читайте дальше.

Производительность замыканий

Замыкания в Rust проектировались максимально быстрыми: быстрее указателей на функции, настолько быстрыми, чтобы их можно было использовать даже в коде, который должен работать на пределе возможностей оборудования. Для знакомых с лямбда-функциями в C++ скажем, что замыкания в Rust такие же быстрые и компактные, но при этом более безопасные.

В большинстве языков замыкания выделяются в куче, диспетчеризуются динамически и уничтожаются сборщиком мусора. Поэтому на создание, вызов и уборку тратится дополнительное процессорное время. Хуже того, замыкания обычно исключают *встраивание* – технику, применяемую компиляторами, чтобы избежать накладных расходов на вызов, – а также ряд других оптимизаций. Из-за всего этого замыкания в таких языках оказываются настолько медленными, что во внутренних циклах их лучше удалять вручную.

В Rust замыкания свободны от всех этих недостатков. Они не убираются в мусор. Как и везде в Rust, память для замыкания не выделяется из кучи, если только не поместить его в `Box`, `Vec` или другой контейнер. А поскольку у каждого замыкания свой тип, то всякий раз, как компилятор Rust знает тип вызываемого замыка-

ния, он может встроить его код. Поэтому замыкания вполне можно использовать во внутренних циклах, и в программах на Rust это делается сплошь и рядом, как мы увидим в главе 15.

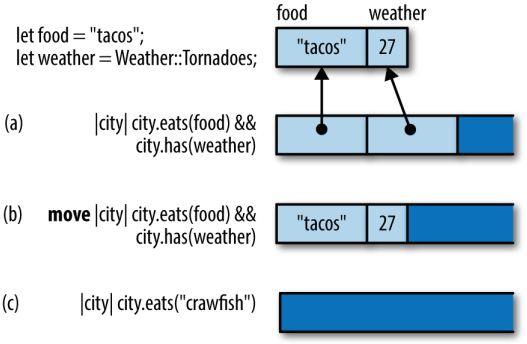


Рис. 14.1 ❖ Размещение замыканий в памяти

На рис. 14.1 показано, как замыкания размещаются в памяти. На верхнем рисунке мы видим две локальные переменные, на которые ссылается замыкание: строка `food` и простое перечисление `weather` с числовым значением 27.

В замыкании (a) используются обе переменные. Понятно, что мы ищем города, в которых имеются лепешки такос и торнадо. В памяти это замыкание выглядит как небольшая структура, содержащая ссылки на используемые переменные.

Обратите внимание, что никакого указателя на код нет! Это необязательно – зная тип замыкания, Rust знает, и какой код выполнять при его вызове.

Замыкание (b) в точности такое же, но ему предшествует ключевое слово `move`, а значит, оно содержит сами значения, а не ссылки.

В замыкании (c) переменные из окружения не используются вовсе. Структура пустая, так что это замыкание не занимает никакого места в памяти.

Как видно по рисунку, замыкания занимают немного места. Но даже эти несколько байтов на практике нужны не всегда. Зачастую компилятор может встроить все обращения к замыканию, и тогда оптимизатор удаляет даже те небольшие структуры, которые показаны на рисунке.

В разделе «Обратные вызовы» ниже мы покажем, как выделить память для замыканий в куче и вызывать их динамически с помощью объектов характеристик. Это несколько медленнее, но все же не уступает по скорости любому другому методу объектов характеристик.

Замыкания и безопасность

Далее мы завершим объяснение того, как замыкания взаимодействуют с системой безопасности Rust. Как было сказано выше, главное заключается в том, что при создании замыкания захваченные переменные либо копируются, либо передаются. Но некоторые последствия не вполне очевидны. В частности, мы поговорим о том, что происходит, когда замыкание уничтожает или модифицирует захваченное значение.

Замыкания, которые убивают

Мы видели замыкания, которые заимствуют значения и которые крадут их. И когда они свернут на дурную дорожку, лишь вопрос времени.

Конечно, «убивают» – не совсем правильная терминология. В Rust мы не убиваем, а *уничтожаем* значения. И самый простой способ сделать это – вызвать функцию `drop()`:

```
let my_str = "hello".to_string();
let f = || drop(my_str);
```

После вызова `f` значение `my_str` уничтожается.

А если вызвать ее дважды?

```
f();
f();
```

Давайте-ка поразмыслим. При первом вызове `f` значение `my_str` уничтожается, т. е. память, где хранилась строка, освобождается и возвращается системе. При втором вызове `f` происходит то же самое. Имеет место «двойное освобождение» – классическая ошибка при программировании на C++, приводящая к неопределенному поведению.

Двойное уничтожение строки в Rust ничем не лучше. По счастью, так просто обмануть Rust не удастся:

```
f();  // правильно
f();  // ошибка: использование переданного значения
```

Rust знает, что это замыкание нельзя вызывать дважды.

Казалось бы, замыкание, которое можно вызывать всего один раз, – вещь довольно необычная. Но мы уже не раз говорили в этой книге о владении и времени жизни. Идея об «израсходовании» (читай: передаче владения) переменной – одна из ключевых концепций Rust. В применении к замыканиям она работает так же, как ко всему остальному.

FnOnce

Предпримем еще одну попытку заставить Rust дважды уничтожить строку. На этот раз воспользуемся универсальной функцией:

```
fn call_twice<F>(closure: F) where F: Fn() {
    closure();
    closure();
}
```

Этой функции можно передать любое замыкание, которое реализует характеристику `Fn()`, т. е. не принимает аргументов и возвращает `()`. (Как и для функций, тип возвращаемого значения можно опускать, если он равен `()`, так что `Fn()` – сокращенная запись `Fn() -> ()`.)

А что произойдет, если передать этой универсальной функции наше небезопасное замыкание?

```
let my_str = "hello".to_string();
let f = || drop(my_str);
call_twice(f);
```

Как и раньше, замыкание уничтожает `my_str` при вызове. Двойной вызов – двойное уничтожение. Но одурачить Rust снова не удалось:

```
error[E0525]: expected a closure that implements the `Fn` trait, but
              this closure only implements `FnOnce`
  --> closures_twice.rs:12:13
   |
12 |     let f = || drop(my_str);
   |             ^^^^^^^^^^^^^^^
   |
note: the requirement to implement `Fn` derives from here
  --> closures_twice.rs:13:5
   |
13 |     call_twice(f);
   |     ^^^^^^^^^^
```

Это сообщение об ошибке дает нам дополнительную информацию о том, как Rust обрабатывает «замыкания, которые убивают». Можно было бы вообще запретить их употребление в языке, но иногда замыкания полезны для очистки. Поэтому Rust только налагает ограничения на их использование. Замыканиям, которые, как `f`, уничтожают значения, запрещено реализовывать характеристику `Fn`. Да они и не `Fn` вовсе, а реализуют менее мощную характеристику, `FnOnce`, помечающую замыкания, которые можно вызывать только один раз.

После первого обращения к замыканию типа `FnOnce` *оно считается израсходованным*. Для наглядности можно предполагать, что две разные характеристики, `Fn` и `FnOnce`, определены следующим образом:

```
// Псевдокод характеристик `Fn` и `FnOnce` без аргументов.
trait Fn() -> R {
    fn call(&self) -> R;
}

trait FnOnce() -> R {
    fn call_once(self) -> R;
}
```

Как арифметическое выражение вида `a + b` является сокращенной формой вызова метода `Add::add(a, b)`, так и `closure()` в Rust считается сокращенной записью метода одной из двух показанных выше характеристик. Для `Fn`-замыкания `closure()` превращается в `closure.call()`. Этот метод принимает `self` по ссылке, так что замыкание не передается. Но если замыкание безопасно вызывать только один раз, то `closure()` превращается в `closure.call_once()`. Этот метод принимает `self` по значению, так что замыкание оказывается израсходованным после вызова.

Разумеется, мы здесь сознательно создали проблему, вызвав `drop()`. На практике такая ситуация обычно возникает случайно. Это бывает не часто, но иногда все же случается написать код замыкания, в котором значение непреднамеренно уничтожается:

```
let dict = produce_glossary();
let debug_dump_dict = || {
    for (key, value) in dict {  // ой!
        println!("{:?} - {:?}", key, value);
    }
};
```

Теперь при повторном вызове `debug_dump_dict()` мы получим такое сообщение об ошибке:

```
error[E0382]: use of moved value: `debug_dump_dict`
  --> closures_debug_dump_dict.rs:18:5
   |
17 |     debug_dump_dict();
   |     ---------------- value moved here
18 |     debug_dump_dict();
   |     ^^^^^^^^^^^^^^^^ value used here after move
   |
   = help: closure was moved because it only implements `FnOnce`
```

Чтобы разобраться, в чем здесь дело, нужно понять, почему это замыкание реализует характеристику `FnOnce`. Какое значение здесь потребляется? Код ссылается только на `dict`. Ага, вот здесь и ошибка: мы потребляем словарь `dict`, обходя его напрямую. А надо было бы обходить `&dict`, а не просто `dict`, чтобы обращаться к значениям по ссылке:

```
let debug_dump_dict = || {
    for (key, value) in &dict {  // dict не потребляется
        println!("{:?} - {:?}", key, value);
    }
};
```

Ошибка исправлена, теперь замыкание реализует `Fn`, и вызывать его можно сколько угодно раз.

FnMut

Существует еще одна разновидность замыкания – содержащее изменяемые данные или ссылки.

Rust считает, что значения без модификатора `mut` безопасно разделять между потоками. Но небезопасно разделять замыкания без `mut`, содержащие изменяемые данные: вызов такого замыкания из нескольких потоков может привести к гонкам за данные, если несколько потоков одновременно пытается читать и записывать одни и те же данные.

Поэтому в Rust введена еще одна категория замыканий, `FnMut`, в которых данные записываются. `FnMut`-замыкания вызываются по `mut`-ссылке, как если бы были определены следующим образом:

```
// Псевдокод характеристик `Fn`, `FnMut` и `FnOnce`.
trait Fn() -> R {
    fn call(&self) -> R;
}

trait FnMut() -> R {
    fn call_mut(&mut self) -> R;
}

trait FnOnce() -> R {
    fn call_once(self) -> R;
}
```

Любое замыкание, требующее изменяющего доступа к значению, но при этом не уничтожающее никаких значений, является FnMut-замыканием. Например:

```
let mut i = 0;
let incr = || {
    i += 1;  // incr заимствует изменяемую ссылку на i
    println!("Ding! i is now: {}", i);
};
call_twice(incr);
```

Написанная ранее функция call_twice требует, чтобы аргумент реализовывал характеристику Fn. Поскольку incr относится к категории FnMut, а не Fn, этот код не компилируется. Но это легко поправить. Чтобы понять, как, отступим на шаг назад и подытожим все, что мы узнали о трех категориях замыканий в Rust:

- Fn – семейство замыканий и функций, которые можно вызывать несколько раз без ограничений. Эта категория включает все fn-функции;
- FnMut – семейство замыканий, которые можно вызывать несколько раз, если в объявлении самого замыкания присутствует модификатор mut;
- FnOnce – семейство замыканий, который можно вызывать только один раз, если вызывающая сторона владеет замыканием.

Любое замыкание семейства Fn удовлетворяет требованиям к FnMut, а любое замыкание семейства FnMut удовлетворяет требованиям к FnOnce. Как видно из рис. 14.2, это не три непересекающиеся категории.

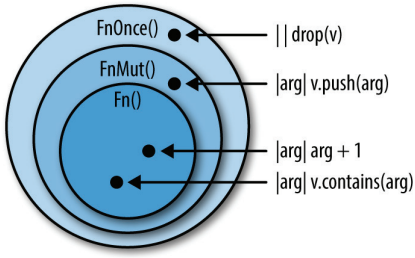


Рис. 14.2 ❖ Диаграмма Венна трех категорий замыканий

Напротив, Fn() – подхарактеристика FnMut(), которая сама является подхарактеристикой FnOnce(). Следовательно, Fn – самая узкая категория, не связанная никакими ограничениями, а FnMut и FnOnce – более широкие и включают замыкания с ограничениями на использование.

Теперь становится ясно, что максимально широкое множество замыканий, которые могла бы принять функция call_twice, – это категория FnMut, т. е. функцию следовало бы объявить так:

```
fn call_twice<F>(mut closure: F) where F: FnMut() {
    closure();
    closure();
}
```

Прежнее ограничение в первой строке, F: Fn(), заменено на F: FnMut(). Теперь call_twice, как и раньше, может принимать все Fn-замыкания, а вдобавок к ним еще и замыкания, в которых данные изменяются.

```
let mut i = 0;
call_twice(|| i += 1);  // ok!
assert_eq!(i, 2);
```

Обратные вызовы

Во многих библиотеках частью API являются *обратные вызовы*: функции, которые пользователь предоставляет, а библиотека вызывает позже. И мы даже встречались с такими API в этой книге. В главе 2 мы использовали каркас Iron для написания простого веб-сервера, который выглядел так:

```
fn main() {
let mut router = Router::new();

    router.get("/", get_form, "root");
    router.post("/gcd", post_gcd, "gcd");

    println!("Serving on http://localhost:3000...");
    Iron::new(router).http("localhost:3000").unwrap();
}
```

Маршрутизатор `router` направляет поступающие из Интернета запросы Rust-коду, обрабатывающему запросы одного вида. В данном случае `get_form` и `post_gcd` – имена функций, объявленных где-то в другом месте программы с помощью ключевого слова `fn`. Но можно было бы вместо них передать замыкания:

```
let mut router = Router::new();

router.get("/", |_: &mut Request| {
    Ok(get_form_response())
}, "root");
router.post("/gcd", |request: &mut Request| {
    let numbers = get_numbers(request)?;
    Ok(get_gcd_response(numbers))
}, "gcd");
```

Это возможно, потому что Iron написан так, что принимает любой потокобезопасный объект, реализующий характеристику `Fn`, в качестве аргумента. А как сделать то же самое в собственной программе? Напишем очень простой маршрутизатор с нуля, не пользуясь никаким кодом из Iron. Начнем с объявления типов для представления HTTP-запросов и ответов.

```
struct Request {
    method: String,
    url: String,
    headers: HashMap<String, String>,
    body: Vec<u8>
}

struct Response {
    code: u32,
    headers: HashMap<String, String>,
    body: Vec<u8>
}
```

Задача маршрутизатора состоит в том, чтобы просто сохранить таблицу, которая отображает URL-адреса на обратные вызовы, так чтобы можно было легко найти нужный обратный вызов. (Для простоты мы разрешаем создавать только маршруты, которым точно соответствует единственный URL.)

```
struct BasicRouter<C> where C: Fn(&Request) -> Response {
    routes: HashMap<String, C>
}

impl<C> BasicRouter<C> where C: Fn(&Request) -> Response {
    /// Создает пустой маршрутизатор.
    fn new() -> BasicRouter<C> {
        BasicRouter { routes: HashMap::new() }
    }

    /// Добавляет маршрут.
    fn add_route(&mut self, url: &str, callback: C) {
        self.routes.insert(url.to_string(), callback);
    }
}
```

К сожалению, мы допустили ошибку. Видите, какую?

Этот маршрутизатор правильно работает, если добавить в него только один маршрут:

```
let mut router = BasicRouter::new();
router.add_route("/", |_| get_form_response());
```

Такой код компилируется и выполняется. Но если добавить еще один маршрут:

```
router.add_route("/gcd", |req| get_gcd_response(req));
```

то компилятор выдаст ошибку:

```
error[E0308]: mismatched types
  --> closures_bad_router.rs:41:30
   |
41 |     router.add_route("/gcd", |req| get_gcd_response(req));
   |                              ^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |                              expected closure, found a different closure
   |
   = note: expected type `[closure@closures_bad_router.rs:40:27: 40:50]`
              found type `[closure@closures_bad_router.rs:41:30: 41:57]`
note: no two closures, even if identical, have the same type
help: consider boxing your closure and/or using it as a trait object
```

Ошибка – в определении типа `BasicRouter`:

```
struct BasicRouter<C> where C: Fn(&Request) -> Response {
    routes: HashMap<String, C>
}
```

Мы неосторожно объявили, что во всех значениях `BasicRouter` один и тот же тип обратного вызова `C` и что все обратные вызовы в `HashMap` имеют такой тип. В разделе «Что использовать» главы 11, рассматривая тип `Salad`, мы столкнулись с той же проблемой.

```
struct Salad<V: Vegetable> {
    veggies: Vec<V>
}
```

Решение такое же, как и тогда: поскольку мы хотим поддерживать разные типы, нужны боксы и объекты характеристик.

```
type BoxedCallback = Box<Fn(&Request) -> Response>;

struct BasicRouter {
    routes: HashMap<String, BoxedCallback>
}
```

Боксы могут содержать замыкания разных типов, так что в одной таблице `HashMap` могут находиться различные обратные вызовы. Отметим, что параметрический тип `C` больше не нужен.

Необходимо также внести небольшие изменения в сами методы:

```
impl BasicRouter {
    // Создает пустой маршрутизатор.
    fn new() -> BasicRouter {
        BasicRouter { routes: HashMap::new() }
    }

    // Добавляет маршрут.
    fn add_route<C>(&mut self, url: &str, callback: C)
        where C: Fn(&Request) -> Response + 'static
    {
        self.routes.insert(url.to_string(), Box::new(callback));
    }
}
```

(Обратите внимание на два ограничения на `C` в сигнатуре `add_route`: конкретная характеристика `Fn` и время жизни `'static`. Rust требует, чтобы ограничение `'static` было задано. Без него вызов `Box::new(callback)` привел бы к ошибке, потому что небезопасно сохранять замыкание, если оно содержит заимствованные ссылки на переменные, которые скоро покинут область видимости.)

Наконец-то наш простой маршрутизатор готов к обработке входящих запросов:

```
impl BasicRouter {
    fn handle_request(&self, request: &Request) -> Response {
        match self.routes.get(&request.url) {
            None => not_found_response(),
            Some(callback) => callback(request)
        }
    }
}
```

Эффективное использование замыканий

Как мы уже видели, замыкания в Rust отличаются от замыканий в большинстве других языков. Главное отличие состоит в том, что в языках со сборкой мусора в замыкании можно использовать локальные переменные, не задумываясь об их времени жизни или владении. В отсутствие сборщика мусора все становится ина-

че. Некоторые паттерны проектирования, обычные в Java, C# и JavaScript, не будут работать в Rust в неизменном виде.

Возьмем, к примеру, паттерн проектирования Модель–Представление–Контроллер (Model-View-Controller, или MVC). Для каждого элемента пользовательского интерфейса каркас на основе MVC создает три объекта: *модель*, описывающую состояние элемента UI, *представление*, отвечающее за его внешний вид, и *контроллер*, обрабатывающий взаимодействия с пользователем. С годами было реализовано множество вариаций на тему MVC, но общая идея состоит в том, что эти три объекта распределяют между собой обязанности UI.

И тут возникает проблема. Обычно каждый объект хранит ссылки на один или оба остальных, напрямую или в виде обратного вызова, как показано на рисунке ниже. Когда с одним объектом что-то происходит, он уведомляет другие, так что обновление происходит быстро. Вопрос о том, какой объект «владеет» другими, даже не поднимается.

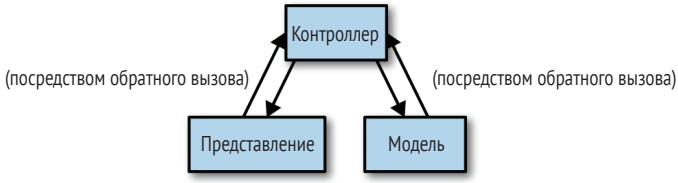


Рис. 14.3 ❖ Паттерн проектирования Модель–Представление–Контроллер

Реализовать этот паттерн в Rust без изменений не получится. Владение должно быть обозначено явно, а циклические ссылки устранены. Модель и контроллер не могут ссылаться друг на друга напрямую.

Бесшабашная ставка Rust состоит в утверждении о том, что существуют хорошие альтернативные подходы. Иногда проблему можно решить с помощью задания принадлежности и времен жизни замыканий, когда каждое замыкание принимает нужные ему ссылки в качестве аргументов. Иногда можно назначить каждому объекту в системе число и передавать эти числа вместо ссылок. Можно также реализовать один из многих вариантов MVC, в котором не все объекты хранят ссылки друг на друга. Или построить инструментарий по образцу отличной от MVC системы с односторонним потоком данных, как в архитектуре Flux, принятой в Facebook и показанной на рис. 14.4.

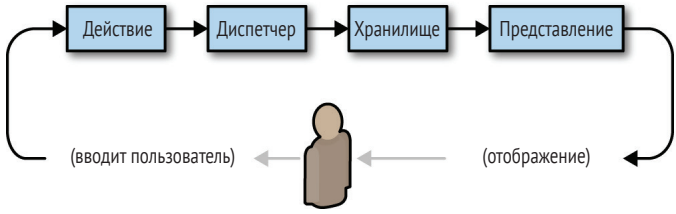


Рис. 14.4 ❖ Архитектура Flux, альтернативная MVC

Короче говоря, попытавшись использовать замыкания Rust для создания «моря объектов», вы столкнетесь с серьезными трудностями. Но альтернативы есть. В данном случае создается впечатление, что программная инженерия как дисциплина уже тяготеет к альтернативам, поскольку они проще.

В следующей главе мы займемся темой, в которой замыкания проявляют себя во всем блеске. Мы напишем код, где в полной мере задействуются лаконичность, скорость и эффективность замыканий в Rust: его приятно писать, легко читать, и он в высшей степени практичен. Встречайте – итераторы.

Глава 15

Итераторы

Наступил конец очень долгого дня.

— *Фил*

Итератором называется значение, которое порождает последовательность значений, обычно обрабатываемых в цикле. В стандартной библиотеке Rust имеются итераторы для обхода векторов, строк, хеш-таблиц и других коллекций, а также итераторы для порождения строк текста из входного потока, подключений, запрашиваемых у сетевого сервера, значений, полученных от других потоков по коммуникационным каналам, и т. д. И конечно, мы можем сами реализовать итераторы для своих целей. Цикл `for` в Rust дает естественный синтаксис для использования итераторов, но и сами итераторы предлагают богатый набор методов для отображения, фильтрации, соединения, собирания и т. д.

Итераторы в Rust гибкие, выразительные и эффективные. Рассмотрим следующую функцию, которая возвращает сумму первых `n` положительных чисел (такая сумма иногда называется «`n`-ым треугольным числом»):

```
fn triangle(n: i32) -> i32 {
    let mut sum = 0;
    for i in 1..n+1 {
        sum += i;
    }
    sum
}
```

Выражение `1..n+1` – это значение типа `Range<i32>`. А `Range<i32>` – итератор, который порождает целые числа от начального (включая) до конечного (не включая), так что его можно использовать в качестве операнда цикла `for` для суммирования значений от `1` до `n`.

Но у итераторов есть также метод `fold`, который дает тот же результат:

```
fn triangle(n: i32) -> i32 {
    (1..n+1).fold(0, |sum, item| sum + item)
}
```

Считая, что начальное значение промежуточного итога равно `0`, метод `fold` берет каждое значение, порождаемое итератором `1..n+1`, и применяет замыкание `|sum, item| sum + item` к промежуточному итогу и этому значению. Значение, возвращенное замыканием, считается новым промежуточным итогом. Последнее

возвращенное значение и есть значение самого метода `fold` – в данном случае сумма всей последовательности. Для тех, кто всю жизнь пользовался циклами `for` и `while`, это выглядит странно, но стоит привыкнуть – и `fold` покажется очень понятной и лаконичной альтернативой.

Это довольно стандартный подход в функциональных языках программирования, которые уделяют особое внимание выразительности. Но итераторы в Rust еще и тщательно спроектированы, так чтобы компилятор мог транслировать их в отличный машинный код. В выпускной сборке второго варианта функции Rust знает определение `fold` и встраивает его в `triangle`. А в результат встраивается замыкание `|sum, item| sum + item`. Наконец, Rust анализирует получившийся код и обнаруживает, что существует более простой способ просуммировать числа от единицы до `n`: сумма всегда равна `n * (n+1) / 2`. Rust транслирует все тело функции `triangle` – цикл, замыкание и все остальное – в одну команду умножения и еще несколько арифметических операций.

В этом примере все ограничивалось простой арифметикой, но итераторы отлично работают и в более сложных ситуациях. Это еще один пример того, как Rust предлагает гибкие абстракции, с которыми при типичном использовании сопряжены небольшие накладные расходы или вообще никаких.

В этой главе пять частей:

- сначала мы опишем характеристики `Iterator` и `IntoIterator`, лежащие в основе итераторов Rust;
- затем перейдем к трем стадиям типичного итераторного конвейера: создание итератора по некоторому источнику значений; адаптация одного вида итераторов к другому путем выборки или обработки поступающих значений; потреблений значений, порождаемых итератором;
- наконец, мы покажем, как реализовать итераторы для своих типов.

Методов очень много, поэтому ничего страшного, если, усвоив общую идею, вы будет просматривать последующие разделы по диагонали. Но итераторы постоянно употребляются в идиоматическом Rust-коде, так что знакомство с сопутствующими им средствами – необходимое условие овладения языком.

Характеристики Iterator и IntoIterator

Итератор – это любое значение, реализующее характеристику `std::iter::Iterator`:

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    ... // много методов по умолчанию
}
```

Здесь `Item` – тип значения, порождаемого итератором. Метод `next` возвращает либо `Some(v)`, где `v` – следующее значение итератора, либо `None`, обозначающее конец последовательности. Мы опустили многочисленные методы `Iterator` по умолчанию, но рассмотрим каждый из них далее в этой главе.

Если существует естественный способ обойти некоторый тип, то в этом типе можно реализовать характеристику `std::iter::IntoIterator`, метод `into_iter` которой принимает значение и возвращает итератор для него:

```
trait IntoIterator where Self::IntoIter::Item == Self::Item {
    type Item;
    type IntoIter: Iterator;
    fn into_iter(self) -> Self::IntoIter;
}
```

Здесь `IntoIter` – тип значения самого итератора, а `Item` – тип порождаемого им значения. Любой тип, реализующий `IntoIterator`, называется *итерируемым*, поскольку это нечто такое, что при желании можно обойти с помощью итератора.

Цикл `for` в Rust изящно сводит всё это воедино. Для обхода элементов вектора можно написать:

```
println!("Существует:");
let v = vec!["сурьма", "мышьяк", "алюминий", "селен"];

for element in &v {
    println!("{}", element);
}
```

Под капотом цикл `for` – просто сокращенная запись вызовов методов `IntoIterator` и `Iterator`:

```
let mut iterator = (&v).into_iter();
while let Some(element) = iterator.next() {
    println!("{}", element);
}
```

В цикле `for` сначала вызывается метод `IntoIterator::into_iter`, чтобы преобразовать операнд `&v` в итератор, а затем повторно вызывается метод `Iterator::next`. Всякий раз как возвращается `Some(element)`, цикл `for` выполняет свое тело, а когда возвращается `None`, цикл завершается.

Хотя цикл `for` всегда вызывает `into_iter` для своего операнда, мы можем передать циклу итератор и напрямую; так, например, происходит при обходе диапазона `Range`. Любой итератор автоматически реализует характеристику `IntoIterator` таким образом, что метод `into_iter` просто возвращает сам этот итератор.

В спецификации характеристики `Iterator` не определено, что делать, если метод итератора `next` вызван еще раз, после того как он вернул `None`. Большинство итераторов просто снова возвращает `None`, но это необязательно (если из-за этого возникают проблемы, может помочь адаптер `fuse`, рассматриваемый в разделе «Fuse» ниже в этой главе).

Подытожим терминологию, относящуюся к итераторам:

- как уже было сказано, *итератором* называется любой тип, реализующий характеристику `Iterator`;
- *итерируемым* называется любой тип, реализующий характеристику `IntoIterator`: чтобы получить итератор для его обхода, нужно вызвать метод `into_iter`. В примере выше тип ссылки на вектор `&v` является итерируемым;
- итератор *порождает* значения;
- значения, порождаемые итератором, называются *объектами* (items). В примере выше `"сурьма"`, `"мышьяк"` и т. д. – объекты;
- код, который получает объекты, порождаемые итератором, называется *потребителем*. В примере выше цикл `for` потребляет объекты итератора.

Создание итераторов

В документации по стандартной библиотеке Rust подробно описано, какие итераторы предоставляет каждый тип, но, чтобы помочь в поиске необходимой информации, библиотека следует некоторым общим соглашениям.

Методы iter и iter_mut

Большинство типов коллекций предоставляет методы `iter` и `iter_mut`, которые возвращают естественные для типа итераторы, порождающие разделяемую или изменяемую ссылку на каждый элемент коллекции. У срезок вида `&[T]` и `&str` тоже есть методы `iter` и `iter_mut`. Эти методы – самый простой способ получить итератор, если цикл `for` вам почему-то не подходит.

```
let v = vec![4, 20, 12, 8, 6];
let mut iterator = v.iter();
assert_eq!(iterator.next(), Some(&4));
assert_eq!(iterator.next(), Some(&20));
assert_eq!(iterator.next(), Some(&12));
assert_eq!(iterator.next(), Some(&8));
assert_eq!(iterator.next(), Some(&6));
assert_eq!(iterator.next(), None);
```

Тип объекта этого итератора – `&i32`: каждое обращение к методу `next` порождает ссылку на следующий элемент, пока не будет достигнут конец вектора.

Любой тип вправе реализовать методы `iter` и `iter_mut` любым способом, имеющим смысл. Метод `iter` типа `std::path::Path` возвращает итератор, который при каждом обращении порождает следующий компонент пути:

```
use std::ffi::OsStr;
use std::path::Path;

let path = Path::new("C:/Users/JimB/Downloads/Fedora.iso");
let mut iterator = path.iter();
assert_eq!(iterator.next(), Some(OsStr::new("C:")));
assert_eq!(iterator.next(), Some(OsStr::new("Users")));
assert_eq!(iterator.next(), Some(OsStr::new("JimB")));
...
```

Тип объекта этого итератора – `&std::ffi::OsStr`, заимствованная срезка строки такого вида, который принимают вызовы операционной системы.

Реализации характеристики IntoIterator

Если тип реализует характеристику `IntoIterator`, то мы можем вызвать его метод `into_iter` самостоятельно, как это делает цикл `for`:

```
// Обычно используется контейнер HashSet, но для него порядок обхода не детерминирован,
// поэтому для примеров лучше подойдет BTreeSet.
use std::collections::BTreeSet;
let mut favorites = BTreeSet::new();
favorites.insert("Lucy in the Sky With Diamonds".to_string());
favorites.insert("Liebesträume No. 3".to_string());

let mut it = favorites.into_iter();
```

```
assert_eq!(it.next(), Some("Liebesträume No. 3".to_string()));
assert_eq!(it.next(), Some("Lucy in the Sky With Diamonds".to_string()));
assert_eq!(it.next(), None);
```

На самом деле большинство коллекций предоставляет несколько реализаций `IntoIterator`: для разделяемых ссылок, для изменяемых ссылок и для передачи владения.

- Получив **разделяемую ссылку** на коллекцию, метод `into_iter` возвращает итератор, порождающий разделяемые ссылки на элементы коллекции. Так, в примере выше `(&favorites).into_iter()` вернет итератор, для которого тип `Item` совпадает с `&String`.
- Получив **изменяемую ссылку** на коллекцию, метод `into_iter` возвращает итератор, порождающий изменяемые ссылки на элементы коллекции. Например, если `vector` – значение типа `Vec<String>`, то вызов `(&mut vector).into_iter()` вернет итератор, для которого тип `Item` совпадает с `&mut String`.
- Если коллекция передана **по значению**, то метод `into_iter` возвращает итератор, который принимает владение коллекцией и возвращает ее элементы по значению; владение элементами передается от коллекции потребителю, и в процессе обхода вся коллекция потребляется. Так, в примере выше вызов `favorites.into_iter()` возвращает итератор, который порождает каждую строку по значению; потребитель становится владельцем каждой строки. После уничтожения итератора все элементы, оставшиеся в `BTreeSet`, также уничтожаются, как и оставшаяся от множества пустая скорлупка.

Поскольку цикл `for` применяет метод `IntoIterator::into_iter` к своему операнду, эти три реализации и лежат в основе следующих идиом для обхода коллекции по разделяемым или изменяемым ссылкам и потребления коллекции с передачей владения ее элементами:

```
for element in &collection { ... }
for element in &mut collection { ... }
for element in collection { ... }
```

Каждая из этих конструкций просто вызывает одну из вышеперечисленных реализаций характеристики `IntoIterator`.

Не каждый тип предоставляет все три реализации. Например, типы `HashSet`, `BTreeSet` и `BinaryHeap` не реализуют `IntoIterator` с изменяемыми ссылками, поскольку модификация элементов могла бы нарушить инварианты типа: у модифицированного значения может оказаться другой хеш-код или может измениться его порядок относительно соседей, так что после модификации оно будет расположено не там, где нужно. Другие типы поддерживают изменение, но только частично. Так, типы `HashMap` и `BTreeMap` порождают изменяемые ссылки на значения записей, но лишь разделяемые ссылки на ключи – по тем же причинам, что и выше.

Общий принцип состоит в том, что итерирование должно быть эффективным и предсказуемым, поэтому, вместо того чтобы предоставлять реализации, которые могли бы оказаться накладными или демонстрировать неожиданное поведение (например, заново хешировать модифицированные записи `HashSet`, что чревато их повторным посещением при продолжении обхода), Rust предпочитает не предоставлять их вовсе.

Срезки реализуют два из трех вариантов `IntoIterator`; поскольку они не владеют своими элементами, то случая «по значению» быть не может. Вместо этого метод `into_iter` для `&[T]` и `&mut [T]` возвращает итератор, который порождает разделяемые и изменяемые ссылки на элементы. Если считать, что базовый тип срезки `[T]` – какая-то коллекция, то такой подход хорошо укладывается в общую схему.

Вы, наверное, обратили внимание, что первые два варианта `IntoIterator` – для разделяемых и изменяемых ссылок – эквивалентны вызову методов `iter` или `iter_mut` для объекта ссылки. Зачем же Rust предоставляет то и другое?

Характеристика `IntoIterator` – это то, благодаря чему цикл `for` вообще работает, так что она, очевидно, необходима. Но если мы не используем цикла `for`, то запись `favorites.iter()`, конечно, понятнее, чем `(&favorites).into_iter()`. Итерирование по разделяемой ссылке используется часто, так что у методов `iter` и `iter_mut` имеется чисто эргономическая ценность.

Характеристика `IntoIterator` бывает полезна и в универсальном коде: мы можем использовать ограничение вида `T: IntoIterator`, чтобы разрешить лишь итерируемые параметрические типы `T`. Или написать `T: IntoIterator<Item=U>`, если хотим дополнительно потребовать, чтобы итератор порождал значения определенного типа `U`. Например, следующая функция распечатывает значения из любого итерируемого типа при условии, что объекты итератора допускают печать по формату `"{:?}"`:

```
use std::fmt::Debug;

fn dump<T, U>(t: T)
    where T: IntoIterator<Item=U>,
          U: Debug
{
    for u in t {
        println!("{:?}", u);
    }
}
```

Написать такую универсальную функцию, используя методы `iter` и `iter_mut`, невозможно, потому что это не методы характеристики; просто так сложилось, что большинство итерируемых типов предоставляет методы с такими именами.

Метод drain

Многие типы коллекций предоставляют метод `drain`, который принимает изменяемую ссылку на коллекцию и возвращает итератор, который передает владение каждым элементом потребителю. Но, в отличие от метода `into_iter()`, который принимает коллекцию по значению и потребляет ее, `drain` просто заимствует изменяемую ссылку на коллекцию и, когда итератор уничтожается, удаляет из коллекции оставшиеся в ней элементы, оставляя ее пустой.

Для типов, которые можно индексировать диапазоном, таких как `String`, `Vec` и `VecDeque`, метод `drain` принимает диапазон подлежащих удалению элементов, а не опустошает всю последовательность:

```
use std::iter::FromIterator;

let mut outer = "Earth".to_string();
```

```
let inner = String::from_iter(outer.drain(1..4));

assert_eq!(outer, "Eh");
assert_eq!(inner, "art");
```

Чтобы опустошить всю последовательность, передайте в качестве аргумента полный диапазон (`..`).

Другие источники итераторов

В предыдущих разделах нас в основном интересовали типы коллекций, например векторы и `HashMap`, но в стандартной библиотеке есть много других типов, поддерживающих итерирование. В табл. 15.1 перечислены самые интересные, но это далеко не всё. Некоторые из этих методов мы рассмотрим подробнее в главах 16, 17 и 18, посвященных конкретным типам.

Таблица 15.1. Другие итераторы в стандартной библиотеке

Тип или характеристика	Выражение	Примечания
`std::ops::Range`	`1..10`	Чтобы диапазон был итерируемым, концевые точки должны быть целыми числами. Диапазон включает начальное значение, но не включает конечное
	`(0..100).step_by(10)`	Итерирование с заданным шагом. Шаг может быть отрицательным
`std::ops::RangeFrom`	`1..`	Неограниченное итерирование. Начальная точка диапазона должна быть целым числом. Если значение становится больше максимально возможного для типа, возникает паника или переполнение
`Option<T>`	`Some(1).iter()`	Ведет себя как вектор, длина которого равна нулю (`None`) или единице (`Some(v)`)
`Result<T, E>`	`Ok("blah").iter()`	Аналогично `Option`, но порождает значения `Ok`
`Vec<T>`, `&[T]`	`v.windows(16)`	Порождает все соседние срезки заданной длины, слева направо. Окна пересекаются
	`v.chunks(16)`	Порождает непересекающиеся соседние срезки заданной длины, слева направо
	`v.chunks_mut(1024)`	Как `chunks`, но срезки изменяемые
	`v.split(|byte| byte & 1 != 0)`	Порождает срезки, разделенные элементами, удовлетворяющими заданному предикату
	`v.split_mut(...)`	То же, что и выше, но срезки изменяемые
	`v.rsplit(...)`	Как `split`, но порождает срезки справа налево
	`v.splitn(n, ...)`	Как `split`, но порождает не более `n` срезок
`String`, `&str`	`s.bytes()`	Порождает байты формы UTF-8
	`s.chars()`	Порождает символы в представлении UTF-8
	`s.split_whitespace()`	Разбивает строку по пробелам и порождает срезки, содержащие только непробельные символы
	`s.lines()`	Порождает срезки, содержащие строчки
	`s.split('/')`	Разбивает строку по заданному образцу и порождает срезки, находящиеся между участками совпадения с образцом. Образец может быть задан по-разному: символы, строки, замыкания
	`s.matches(char::is_numeric)`	Порождает срезки, соответствующие заданному образцу

Окончание табл. 15.1

Тип или характеристика	Выражение	Примечания
`std::collections::HashMap`, `std::collections::BTreeMap`	`map.keys()`, `map.values()`	Порождает разделяемые ссылки на ключи или значения отображения
	`map.values_mut()`	Порождает изменяемые ссылки на значения записей
`std::collections::HashSet`, `std::collections::BTreeSet`	`set1.union(set2)`	Порождает разделяемые ссылки на элементы объединения `set1` и `set2`
	`set1.intersection(set2)`	Порождает разделяемые ссылки на элементы пересечения `set1` и `set2`
`std::sync::mpsc::Receiver`	`recv.iter()`	Порождает значения, посылаемые из другого потока выполнения соответствующим значением `Sender`
`std::io::Read`	`stream.bytes()`	Порождает байты из потока ввода-вывода
	`stream.chars()`	Разбирает поток в кодировке UTF-8 и порождает символы `char`
`std::io::BufRead`	`bufstream.lines()`	Разбирает поток в кодировке UTF-8 и порождает строки `String`
	`bufstream.split(0)`	Разбивает поток по заданному байту, порождает межбайтовые буферы `Vec<u8>`
`std::fs::ReadDir`	`std::fs::read_dir(path)`	Порождает элементы каталога
`std::net::TcpListener`	`listener.incoming()`	Порождает входящие сетевые подключения
Свободные функции	`std::iter::empty()`	Сразу возвращает `None`
	`std::iter::once(5)`	Порождает заданное значение, а затем конец данных
	`std::iter::repeat("#9")`	Порождает заданное значение вечно

Адаптеры итераторов

Если уже имеется итератор, то можно воспользоваться многочисленными «адаптерными» методами, или просто «адаптерами», предоставляемыми характеристикой `Iterator`. Они потребляют один итератор и конструируют другой, обладающий каким-то полезным поведением. Чтобы понять, как работают адаптеры, мы продемонстрируем два наиболее популярных.

map и filter

Адаптер `map` характеристики `Iterator` позволяет преобразовать итератор, применив замыкание к его объектам. Адаптер `filter` фильтрует элементы, порождаемые итератором, применяя замыкание, чтобы решить, какие объекты оставлять, а какие отбрасывать.

Предположим, к примеру, что мы обходим строки текста и хотим убрать из каждой строки начальные и конечные пробелы. Стандартный библиотечный метод `str::trim` удаляет начальные и конечные пробелы из переданной по ссылке `&str` строки и возвращает новую ссылку `&str`, заимствованную у исходной. Адаптер `map` позволяет применить метод `str::trim` к каждой строке, полученной от итератора:

```
let text = "  ponies  \n   giraffes\niguanas  \nsquid".to_string();
let v: Vec<&str> = text.lines()
    .map(str::trim)
    .collect();
assert_eq!(v, ["ponies", "giraffes", "iguanas", "squid"]);
```

Вызов `text.lines()` возвращает итератор, порождающий строчки, составляющие строку. Вызов `map` для этого итератора возвращает другой итератор, который применяет метод `str::trim` к каждой строчке и порождает результаты в виде собственных объектов. Наконец, метод `collect` собирает из этих объектов вектор.

Разумеется, итератор `map` сам является кандидатом на дальнейшую адаптацию. Чтобы исключить из результата игуан, мы можем написать:

```
let text = "  ponies  \n   giraffes\niguanas  \nsquid".to_string();
let v: Vec<&str> = text.lines()
    .map(str::trim)
    .filter(|s| *s != "iguanas")
    .collect();
assert_eq!(v, ["ponies", "giraffes", "squid"]);
```

Здесь адаптер `filter` возвращает третий итератор, который порождает только те объекты итератора `map`, для которых замыкание `|s| s != &"iguanas"` возвращает `true`. Цепочка адаптеров итераторов похожа на конвейеры оболочки Unix: каждый адаптер решает одну задачу, а как при этом преобразуется последовательность, становится ясно, если читать слева направо.

Ниже приведены сигнатуры этих адаптеров:

```
fn map<B, F>(self, f: F) -> some Iterator<Item=B>
    where Self: Sized, F: FnMut(Self::Item) -> B;

fn filter<P>(self, predicate: P) -> some Iterator<Item=Self::Item>
    where Self: Sized, P: FnMut(&Self::Item) -> bool;
```

Нотация `some Iterator<...>`, использованная здесь для типов возвращаемого значения, не является допустимым Rust-кодом[1]. Истинные возвращаемые типы – закрытые малоинформативные структуры `struct`. На практике же важно, что эти методы возвращают итераторы с заданным типом `Item`. (В будущих версиях Rust будет предложен синтаксис, больше подходящий для этой цели.)

Поскольку большинство адаптеров принимает `self` по значению, требуется, чтобы тип `Self` был размерным (все распространенные итераторы этому условию удовлетворяют).

Итератор `map` передает каждый объект своему замыканию по значению, а владение результатом замыкания передает потребителю. Итератор `filter` передает каждый объект своему замыканию по разделяемой ссылке, сохраняя за собой владение в случае, когда объект будет отобран для передачи потребителю. Именно поэтому в нашем примере нужно сравнивать `s` с `&"iguanas"`: объект итератора `filter` имеет тип `&str`, так что тип аргумента `s` замыкания – `&&str`.

Об адаптерах итераторов нужно сделать два важных замечания.

Во-первых, сам по себе вызов адаптера для итератора не потребляет никаких объектов; он просто возвращает новый итератор, готовый порождать собственные объекты, запрашивая данные у первого итератора по мере необходимости. Чтобы добиться от цепочки адаптеров реальной работы, нужно вызвать метод `next` последнего итератора.

---

[1] В RFC 1522 для Rust описан синтаксис, который планируется добавить в язык, очень похожий на нашу нотацию `some Iterator`. Но в версии 1.17 он еще не включен в язык по умолчанию.

Так, в нашем примере вызов метода `text.lines()` не производит разбора строки и выделения из нее строчек, а просто возвращает итератор, который *смог бы* это сделать, если его попросят. Аналогично адаптеры `map` и `filter` просто возвращают новые итераторы, которые *смогли бы* произвести отображение или фильтрацию при поступлении соответствующей просьбы.

Это особенно важно, если используются адаптеры с побочными эффектами. Например, следующий код не печатает ничего:

```
["earth", "water", "air", "fire"].iter().map(|elt| println!("{}", elt));
```

Вызов `iter` возвращает итератор для обхода элементов массива, а вызов `map` – второй итератор, который применяет замыкание к каждому значению, порожденному первым. Но здесь нет ничего такого, что запрашивало бы значение у всей цепочки, так что никакой метод `next` нигде не вызывается. На самом деле Rust предупредит об этом:

```
warning: unused result which must be used:
iterator adapters are lazy and do nothing unless consumed,
#[warn(unused_must_use)] on by default
    |
399 |     ["earth", "water", "air", "fire"].iter().map(|elt| println!("{}", elt));
    |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Слово «lazy» (ленивый) в этом сообщении не несет негативного оттенка, это просто жаргонный термин, употребляемый для любого механизма, который откладывает вычисление до того момента, когда значение действительно понадобится. По соглашению, принятому в Rust, итераторы должны выполнять минимум работы, необходимой для удовлетворения каждого вызова `next`; в этом примере таких вызовов нет вовсе, поэтому и работы никакой тоже нет.

Второе важное замечание состоит в том, что адаптеры итераторов – абстракция с нулевыми накладными расходами. Поскольку `map`, `filter` и другие подобные адаптеры универсальны, применение их к итератору приводит к специализации кода в соответствии с типом конкретного итератора. Это означает, что у Rust достаточно информации, чтобы встроить метод `next` каждого итератора в его потребителя, а затем транслировать все получившееся в машинный код как единое целое. Поэтому показанная выше цепочка итераторов `lines`/`map`/`filter` по эффективности не уступает такому написанному вручную коду:

```
for line in text.lines() {
    let line = line.trim();

    if line != "iguanas" {
        v.push(line);
    }
}
```

Далее в этом разделе мы рассмотрим различные адаптеры, определенные в характеристике `Iterator`.

filter_map и flat_map

Адаптер `map` хорош в ситуациях, когда каждый входной объект порождает один выходной. Но что, если мы хотим удалить некоторые объекты, а не обрабатывать

их, или заменить один объект нулем или более других? Такую гибкость дают адаптеры `filter_map` и `flat_map`.

Адаптер `filter_map` похож на `map`, но позволяет своему замыканию либо преобразовать объект в другой (как `map`), либо отбросить его. Иначе говоря, это некая комбинация `filter` и `map`. Вот его сигнатура:

```
fn filter_map<B, F>(self, f: F) -> some Iterator<Item=B>
    where Self: Sized, F: FnMut(Self::Item) -> Option<B>;
```

Она отличается от сигнатуры `map` только тем, что замыкание возвращает `Option<B>`, а не просто `B`. Если замыкание возвращает `None`, объект отбрасывается, а если `Some(b)`, то `b` является следующим объектом, порождаемым итератором `filter_map`.

Пусть, например, мы хотим искать в строке слова, разделенные пробелами, которые можно разобрать как числа, и обработать только эти числа, отбросив все остальные слова. Тогда можно написать:

```
use std::str::FromStr;

let text = "1\nfrond .25  289\n3.1415 estuary\n";
    for number in text.split_whitespace()
                      .filter_map(|w| f64::from_str(w).ok()) {
    println!("{:4.2}", number.sqrt());
}
```

В результате будет напечатано:

```
1.00
0.50
17.00
1.77
```

Замыкание, переданное фильтру `filter_map`, пытается разобрать каждую из разделенных пробелами срезок, применяя функцию `f64::from_str`. Она возвращает значение типа `Result<f64, ParseFloatError>`, которое метод `.ok()` преобразует в `Option<f64>`: ошибка разбора превращается в `None`, а результат успешного разбора – в `Some(v)`. Итератор `filter_map` отбрасывает все значения `None`, а для каждого `Some(v)` порождает значение `v`.

Но какой смысл объединять `map` и `filter` в оду операцию, если эти адаптеры можно применить непосредственно? Ценность адаптера `filter_map` проявляется в ситуациях, когда наилучший способ решить, включить объект или отбросить, – попытаться обработать его. Именно так и устроен этот пример. Сделать то же самое с помощью `filter` и `map` можно, но получается коряво:

```
text.split_whitespace()
    .map(|w| f64::from_str(w))
    .filter(|r| r.is_ok())
    .map(|r| r.unwrap())
```

Адаптер `flat_map` можно рассматривать как развитие идеи `map` и `filter_map` с тем отличием, что замыкание может возвращать не один объект (как в случае `map`) и не нуль или один объект (как `filter_map`), а последовательность, содержащую произвольное число объектов. Итератор `flat_map` порождает конкатенацию последовательностей, возвращенных замыканием.

Сигнатура `flat_map` такова:

```
fn flat_map<U, F>(self, f: F) -> some Iterator<Item=U::Item>
    where F: FnMut(Self::Item) -> U, U: IntoIterator;
```

Замыкание, переданное `flat_map`, должно вернуть итерируемое значение, а его конкретная природа неважна[1].

Предположим, к примеру, что имеется таблица, отображающая страны на их крупные города. Имея список стран, как обойти их крупные города?

```
use std::collections::HashMap;

let mut major_cities = HashMap::new();
major_cities.insert("Japan", vec!["Tokyo", "Kyoto"]);
major_cities.insert("The United States", vec!["Portland", "Nashville"]);
major_cities.insert("Brazil", vec!["São Paulo", "Brasília"]);
major_cities.insert("Kenya", vec!["Nairobi", "Mombasa"]);
major_cities.insert("The Netherlands", vec!["Amsterdam", "Utrecht"]);

let countries = ["Japan", "Brazil", "Kenya"];

for &city in countries.iter().flat_map(|country| &major_cities[country]) {
    println!("{}", city);
}
```

В результате печатается:

```
Tokyo
Kyoto
Sao Paulo
Brasilia
Nairobi
Mombasa
```

Можно было бы интерпретировать это следующим образом: для каждой страны получаем вектор ее городов, конкатенируем все векторы в одну последовательность и печатаем ее.

Но следует помнить, что итераторы ленивые: только обращения к методу `next` итератора `flat_map` из цикла `for` заставляют его проделать хоть какую-то работу. Полная конкатенированная последовательность в памяти никогда не присутствует. Вместо этого мы здесь имеем небольшой конечный автомат, который вытягивает объекты из итератора городов по одному за раз, пока не дойдет до конца, а затем порождает новый итератор городов для следующей страны. По сути дела, получается вложенный цикл, организованный так, что его можно использовать как итератор.

scan

Адаптер `scan` напоминает `map` с тем отличием, что замыканию передается изменяемое значение, которое он может проанализировать и при необходимости за-

[1] На самом деле, поскольку `Option` – итерируемое значение, которое ведет себя как последовательность, состоящая из нуля или одного объекта, вызов `iterator.filter_map(closure)` эквивалентен `iterator.flat_map(closure)` в предположении, что `closure` возвращает `Option<T>`.

вершить итерирование досрочно. Адаптер принимает начальное состояние, а затем передает замыканию изменяемую ссылку на состояние и следующий объект от поставляющего данные итератора. Замыкание должно вернуть значение типа `Option`, которое `scan` интерпретирует как следующий объект.

Например, следующая цепочка итераторов возводит в квадрат объект предыдущего итератора и завершается, когда сумма объектов превысит 10:

```
let iter = (0..10)
    .scan(0, |sum, item| {
        *sum += item;
        if *sum > 10 {
            None
        } else {
            Some(item * item)
        }
    });

assert_eq!(iter.collect::<Vec<i32>>(), vec![0, 1, 4, 9, 16]);
```

Аргумент замыкания `sum` – изменяемая ссылка на закрытую переменную итератора, которая инициализируется первым аргументом итератора, в данном случае `0`. Замыкание обновляет `*sum`, проверяет, не превышен ли лимит, и возвращает следующий результат итератора.

take и take_while

Адаптеры `take` и `take_while` характеристики `Iterator` позволяют завершить итерирование после заданного числа объектов или когда замыкание решит, что пора. Вот их сигнатуры:

```
fn take(self, n: usize) -> some Iterator<Item=Self::Item>
    where Self: Sized;

fn take_while<P>(self, predicate: P) -> some Iterator<Item=Self::Item>
    where Self: Sized, P: FnMut(&Self::Item) -> bool;
```

Оба адаптера принимают владение итератором и возвращают новый итератор, который пропускает объекты, начиная с первого, и, возможно, прекращает обход до того, как будет просмотрена вся последовательность. Итератор `take` возвращает `None` после порождения `n` объектов. Итератор `take_while` применяет предикат `predicate` к каждому объекту и возвращает `None` вместо первого объекта, для которого предикат вернул `false`, а также при каждом последующем обращении к `next`.

Например, для сообщения электронной почты, в котором заголовки отделяются от тела пустой строкой, мы можем воспользоваться адаптером `take_while`, чтобы обойти только заголовки:

```
let message = "To: jimb\r\n\
               From: superego <editor@oreilly.com>\r\n\
               \r\n\
               Did you get any writing done today?\r\n\
               When will you stop wasting time plotting fractals?\r\n";
for header in message.lines().take_while(|l| !l.is_empty()) {
    println!("{}" , header);
}
```

Напомним (см. раздел «Строковые литералы» главы 3), что если строчка заканчивается знаком \, то Rust игнорирует красную строку в следующей строчке, так что ни одна строка в этом тексте не начинается пробелами. Следовательно, третья строка `message` пуста. Адаптер `take_while` прекращает итерирование, как только видит пустую строчку, поэтому программа напечатает всего две строчки:

```
To: jimb
From: superego <editor@oreilly.com>
```

skip и skip_while

Методы `skip` и `skip_while` характеристики `Iterator` дополняют `take` и `take_while`: они отбрасывают начальные объекты – либо заданное количество, либо до тех пор, пока замыкание не решит, что хватит, – а остальные объекты пропускают без изменения. Вот их сигнатуры:

```
fn skip(self, n: usize) -> some Iterator<Item=Self::Item>
    where Self: Sized;

fn skip_while<P>(self, predicate: P) -> some Iterator<Item=Self::Item>
    where Self: Sized, P: FnMut(&Self::Item) -> bool;
```

Одно из распространенных применений адаптера `skip` – пропуск имени команды в процессе обхода аргументов командной строки. В программе вычисления наибольшего общего знаменателя из главы 2 использовался следующий код для обхода аргументов командной строки:

```
for arg in std::env::args().skip(1) {
    ...
}
```

Функция `std::env::args` возвращает итератор, порождающий аргументы программы в виде строк, причем первым объектом является имя самой программы. Но эту строку мы не хотим обрабатывать в цикле. Вызов метода `skip(1)` этого итератора возвращает новый итератор, который отбрасывает имя программы при первом вызове и порождает все последующие аргументы.

Адаптер `skip_while` пользуется замыканием, чтобы решить, сколько начальных объектов следует отбросить. Для обхода строк тела сообщения из примера в предыдущем разделе можно написать такой код:

```
for body in message.lines()
    .skip_while(|l| !l.is_empty())
    .skip(1) { println!("{}" ,
    body);
}
```

Здесь `skip_while` пропускает непустые строки, но порождает саму пустую строку, поскольку для нее замыкание возвращает `false`. Поэтому мы вызываем метод `skip`, чтобы пропустить и эту строку тоже, и в итоге получаем итератор, для которого первым объектом будет первая строка тела сообщения. Таким образом, для приведенного выше сообщения `message` эта программа напечатает:

```
Did you get any writing done today?
When will you stop wasting time plotting fractals?
```

peekable

Итератор с возможностью предварительного просмотра позволяет подглядеть следующий порождаемый объект, не потребляя его. Почти любой итератор можно превратить в такой, вызвав метод `peekable` характеристики `Iterator`:

```
fn peekable(self) -> std::iter::Peekable<Self>
    where Self: Sized;
```

Здесь `Peekable<Self>` – структура, реализующая характеристику `Iterator<Item=Self::Item>`, а `Self` – тип исходного итератора.

У итератора `Peekable` имеется дополнительный метод `peek`, который возвращает `Option<&Item>`: `None`, если исходный итератор завершился, а в противном случае `Some(r)`, где `r` – разделяемая ссылка на следующий объект. (Отметим, что если тип объекта итератора уже является ссылкой на что-то, то в итоге получается ссылка на ссылку.)

Метод `peek` пытается получить следующий объект от исходного итератора и, если таковой существует, кэширует его в ожидании следующего обращения к `next`. Все остальные методы `Iterator`, реализованные в типе `Peekable`, знают об этом кэше. Например, метод `iter.last()` итератора `iter` с возможностью предварительного просмотра знает, что после исчерпания исходного итератора нужно проверить кэш.

Итераторы с возможностью предварительного просмотра особенно важны, когда решение о том, сколько объектов итератора необходимо потребить, можно принять лишь после того, как уже потреблен лишний объект. Например, когда мы выделяем числа из потока символов, понять, где кончается число, можно, лишь увидев первый символ, который не может быть частью числа:

```
use std::iter::Peekable;

fn parse_number<I>(tokens: &mut Peekable<I>) -> u32
    where I: Iterator<Item=char>
{
    let mut n = 0;
    loop {
        match tokens.peek() {
            Some(r) if r.is_digit(10) => {
                n = n * 10 + r.to_digit(10).unwrap();
            }
            _ => return n
        }
        tokens.next();
    }
}

let mut chars = "226153980,1766319049".chars().peekable();
assert_eq!(parse_number(&mut chars), 226153980);

// Гляди-ка, `parse_number` не потребила запятую! Тогда это сделаем мы.
assert_eq!(chars.next(), Some(','));
assert_eq!(parse_number(&mut chars), 1766319049);
assert_eq!(chars.next(), None);
```

Функция `parse_number` вызывает `peek`, чтобы проверить следующий символ, и потребляет его, только если это цифра. Если это не цифра или если итератор исчер-

пан (т. е. peek вернула None), мы возвращаем выделенное число и оставляем следующий символ в итераторе готовым к потреблению.

fuse

Спецификация характеристики Iterator ничего не говорит о том, как должен вести себя метод next, если при предыдущем обращении он уже вернул None. Большинство итераторов просто возвращает None снова, но есть и исключения. Если программа зависит от этого поведения, то вас может ждать сюрприз.

Адаптер fuse принимает произвольный итератор и преобразует его в такой, который гарантированно будет и дальше возвращать None, после того как это произошло в первый раз.

```
struct Flaky(bool);

impl Iterator for Flaky {
    type Item = &'static str;
    fn next(&mut self) -> Option<Self::Item> {
        if self.0 {
            self.0 = false;
            Some("точно последний объект")
        } else {
            self.0 = true; // D'oh!
            None
        }
    }
}

let mut flaky = Flaky(true);
assert_eq!(flaky.next(), Some("точно последний объект"));
assert_eq!(flaky.next(), None);
assert_eq!(flaky.next(), Some("точно последний объект"));

let mut not_flaky = Flaky(true).fuse();
assert_eq!(not_flaky.next(), Some("точно последний объект"));
assert_eq!(not_flaky.next(), None);
assert_eq!(not_flaky.next(), None);
```

Адаптер fuse, пожалуй, наиболее полезен в универсальном коде, который должен работать с итераторами неизвестного происхождения. Вместо того чтобы надеяться на добропорядочность подсунутого итератора, можно гарантировать ее с помощью fuse.

Обратимые итераторы и rev

Некоторые итераторы способны выбирать объекты с обеих сторон последовательности. Для обращения такого итератора служит адаптер rev. Например, итератор для обхода вектора может с одинаковым успехом выбирать элементы как с конца, так и с начала вектора. Подобные итераторы могут реализовывать характеристику std::iter::DoubleEndedIterator, расширяющую Iterator:

```
trait DoubleEndedIterator: Iterator {
    fn next_back(&mut self) -> Option<Self::Item>;
}
```

Можете считать, что двусторонний итератор имеет два пальца: один указывает на текущее начало последовательности, а другой – на текущий конец. При выборке объектов с любой стороны последовательности пальцы сдвигаются; когда они сойдутся, итерирование завершается.

```
use std::iter::DoubleEndedIterator;

let bee_parts = ["голова", "грудь", "брюшко"];

let mut iter = bee_parts.iter();
assert_eq!(iter.next(),      Some(&"голова"));
assert_eq!(iter.next_back(), Some(&"брюшко"));
assert_eq!(iter.next(),      Some(&"грудь"));

assert_eq!(iter.next_back(), None);
assert_eq!(iter.next(),      None);
```

Структура итератора по срезке упрощает реализацию такого поведения: это действительно пара указателей на начало и конец диапазона элементов, которые еще не были порождены; методы `next` и `next_back` просто выбирают элемент с одной или с другой стороны. Итераторы упорядоченных коллекций также двусторонние: метод `next_back` выбирает наибольший элемент или запись. В общем случае стандартная библиотека предоставляет двусторонние итераторы там, где это имеет практический смысл.

Но не для каждого итератора это легко сделать: итератор, порождающий значения, прибывающие из других потоков по каналу `Receiver`, не может предвидеть, каким будет последнее значение. В общем случае нужно смотреть в документации, какие итераторы реализуют характеристику `DoubleEndedIterator`, а какие – нет.

Двусторонний итератор можно обратить с помощью адаптера `rev`:

```
fn rev(self) -> some Iterator<Item=Self>
    where Self: Sized + DoubleEndedIterator;
```

Возвращенный итератор также двусторонний: его методы `next` и `next_back` просто поменялись местами:

```
let meals = ["завтрак", "обед", "ужин"];

let mut iter = meals.iter().rev();
assert_eq!(iter.next(), Some(&"ужин"));
assert_eq!(iter.next(), Some(&"обед"));
assert_eq!(iter.next(), Some(&"завтрак"));
assert_eq!(iter.next(), None);
```

Большинство адаптеров итераторов, будучи применено к обратимому итератору, возвращает другой обратимый итератор. Например, адаптеры `map` и `filter` сохраняют обратимость.

inspect

Адаптер `inspect` удобен для отладки конвейеров адаптеров, но в производственном коде используется редко. Он просто применяет замыкание к разделяемой ссылке на каждый объект, а затем передает объект дальше. Замыкание не может воздействовать на объекты, но может распечатывать их или проверять утверждения с их участием.

Ниже приведен пример, в котором преобразование строки в верхний регистр изменяет ее длину:

```
let upper_case: String = "große".chars()
    .inspect(|c| println!("   до: {:?}", c))
    .flat_map(|c| c.to_uppercase())
    .inspect(|c| println!("после:     {:?}", c))
    .collect();
assert_eq!(upper_case, "GROSSE");
```

В немецком языке строчной букве «ß» соответствуют две заглавные буквы «SS», поэтому `char::to_uppercase` возвращает итератор для обхода символов, а не единственный заменяющий символ. В этом коде `flat_map` конкатенирует все возвращенные `to_uppercase` последовательности в одну строку `String` и по ходу дела выводит следующие сообщения:

```
   до: 'g'
после:    'G'
   до: 'r'
после:    'R'
   до: 'o'
после:    'O'
   до: 'ß'
после:    'S'
после:    'S'
   до: 'e'
после:    'E'
```

chain

Адаптер `chain` помещает один итератор после другого. Точнее, `i1.chain(i2)` возвращает итератор, который выбирает объекты из `i1`, пока тот не исчерпается, а затем выбирает объекты из `i2`.

Сигнатура `chain` выглядит следующим образом:

```
fn chain<U>(self, other: U) -> some Iterator<Item=Self::Item>
    where Self: Sized, U: IntoIterator<Item=Self::Item>;
```

Иными словами, итератор можно сцеплять с любым итерируемым значением, порождающим объекты того же типа.

Например:

```
let v: Vec<i32> = (1..4).chain(vec![20, 30, 40]).collect();
assert_eq!(v, [1, 2, 3, 20, 30, 40]);
```

Итератор `chain` обратим, если оба исходных итератора обратимы:

```
let v: Vec<i32> = (1..4).chain(vec![20, 30, 40]).rev().collect();
assert_eq!(v, [40, 30, 20, 3, 2, 1]);
```

Итератор `chain` просто следит за тем, какой из двух исходных итераторов вернул `None`, и в зависимости от этого вызывает методы `next` и `next_back` того или другого.

enumerate

Адаптер `enumerate` характеристики `Iterator` присоединяет к последовательности индекс текущего элемента, т. е. если исходный итератор порождает объекты `A`, `B`,

C, ..., то новый итератор порождает пары (0, A), (1, B), (2, C), ... На первый взгляд, вещь тривиальная, но используется на удивление часто.

Потребитель может использовать индекс, чтобы отличить один объект от другого и установить контекст для обработки каждого объекта. Например, программа рисования множества Мандельброта из вводной главы разбивает изображение на восемь горизонтальных полос и назначает каждой полосе отдельный поток. В ней адаптер enumerate используется, для того чтобы сообщить потоку, какой части изображения соответствует его полоса.

Начинаем с прямоугольного буфера пикселей:

```
let mut pixels = vec![0; columns * rows];
```

Метод chunks_mut разбивает изображение на горизонтальные полосы, по одной для каждого потока:

```
let threads = 8;
let band_rows = rows / threads + 1;
...
let bands: Vec<&mut [u8]> = pixels.chunks_mut(band_rows * columns).collect();
```

Затем обходим полосы и запускаем для каждой из них поток:

```
for (i, band) in bands.into_iter().enumerate() {
    let top = band_rows * i;
    // запустить поток для отрисовки строк `top..top + band_rows`
}
```

Каждая итерация получает пару (i, band), где band – срезка &mut [u8] буфера пикселей, в которой должен рисовать поток, а i – индекс этой полосы в полном изображении, известный благодаря любезности адаптера enumerate. Вкупе с границами графика и размером полос этой информации достаточно, чтобы поток мог определить, какая часть изображения ему назначена и, следовательно, что именно рисовать в полосе band.

zip

Адаптер zip объединяет два итератора в один, порождающий пары значений по одному из каждого итератора, – как молния соединяет две стороны в один шов. «Сшитый» итератор исчерпывается, когда исчерпывается любой их двух исходных.

Например, тот же результат, что дает адаптер enumerate, можно получить, сшив полуоткрытый диапазон 0.. с другим итератором:

```
let v: Vec<_> = (0..).zip("ABCD".chars()).collect();
assert_eq!(v, vec![(0, 'A'), (1, 'B'), (2, 'C'), (3, 'D')]);
```

Поэтому zip можно рассматривать как обобщение enumerate: если enumerate присоединяет к последовательности индексы, то zip – объекты произвольного итератора. Выше мы сказали, что enumerate позволяет установить контекст для обработки объектов; zip – более гибкий способ сделать то же самое.

Аргументом zip необязательно должен быть итератор как таковой, это может быть любое итерируемое значение:

```
use std::iter::repeat;

let endings = vec!["once", "twice", "chicken soup with rice"];
```

```
let rhyme: Vec<_> = repeat("going")
    .zip(endings)
    .collect();
assert_eq!(rhyme, vec![("going", "once"),
                       ("going", "twice"),
                       ("going", "chicken soup with rice")]);
```

by_ref

В этом разделе мы присоединяли адаптеры к итераторам. А после того как это сделано, можно ли отсоединить адаптер? Обычно нет: адаптер принимает владение исходным итератором, и отдать его нет никакой возможности.

Метод итератора `by_ref` заимствует изменяемую ссылку на итератор, так чтобы можно было применять адаптеры к ссылке. Завершив потребление объектов от этих адаптеров, мы можем уничтожить их, тогда заимствование прекратится и будет восстановлен доступ к исходному итератору.

Например, выше в этой главе мы показали, как использовать адаптеры `take_while` и `skip_while` для обработки заголовков и тела почтового сообщения. Но что, если мы хотим сделать то и другое, пользуясь одним и тем же исходным итератором? Благодаря `by_ref` мы можем применить `take_while` для обработки заголовков, а по завершении этой работы вернуть себе исходный итератор, который `take_while` оставил именно в том состоянии, которое необходимо для обработки тела сообщения:

```
let message = "To: jimb\r\n\
               From: id\r\n\
               \r\n\
               Oooooh, donuts!!\r\n";

let mut lines = message.lines();

println!("Заголовки:");
for header in lines.by_ref().take_while(|l| !l.is_empty()) {
    println!("{}" , header);
}

println!("\nТело:");
for body in lines {
    println!("{}" , body);
}
```

Вызов `lines.by_ref()` заимствует изменяемую ссылку на итератор, и именно владение этой ссылкой принимает итератор `take_while`. В конце первого цикла `for` этот итератор выходит из области видимости, вследствие чего заимствованию приходит конец, так что во втором цикле `for` мы снова можем использовать `lines`. В результате печатается:

```
Заголовки:
To: jimb
From: id

Тело:
Oooooh, donuts!!
```

Определение адаптера by_ref тривиально: он просто возвращает изменяемую ссылку на итератор. А затем в стандартной библиотеке идет такая странная реализация:

```
impl<'a, I: Iterator + ?Sized> Iterator for &'a mut I {
    type Item = I::Item;
    fn next(&mut self) -> Option<I::Item> {
        (**self).next()
    }
    fn size_hint(&self) -> (usize, Option<usize>) {
        (**self).size_hint()
    }
}
```

Иными словами, если I – некоторый тип итератора, то &mut I – тоже итератор, методы next и size_hint которого просто делегируют работу объекту ссылки. Адаптер, вызываемый для изменяемой ссылки на итератор, принимает владение *ссылкой*, а не самим итератором. А это значит, что по выходе адаптера из области действия заканчивается только заимствование.

cloned

Адаптер cloned принимает итератор, порождающий ссылки, и возвращает итератор, порождающий значения, являющиеся результатом клонирования этих ссылок. Естественно, тип объекта ссылки должен реализовывать характеристику Clone.

```
let a = ['1', '2', '3', '∞'];

assert_eq!(a.iter().next(),          Some(&'1'));
assert_eq!(a.iter().cloned().next(), Some('1'));
```

cycle

Адаптер cycle возвращает итератор, который без конца повторяет последовательность, порождаемую исходным итератором. Исходный итератор должен реализовывать характеристику std::clone::Clone, чтобы cycle мог сохранить свое начальное состояние и повторно использовать его при каждом повторении цикла. Например:

```
let dirs = ["Север", "Восток", "Юг", "Запад"];
let mut spin = dirs.iter().cycle();
assert_eq!(spin.next(), Some(&"Север"));
assert_eq!(spin.next(), Some(&"Восток"));
assert_eq!(spin.next(), Some(&"Юг"));
assert_eq!(spin.next(), Some(&"Запад"));
assert_eq!(spin.next(), Some(&"Север"));
assert_eq!(spin.next(), Some(&"Восток"));
```

А вот пример совсем уж праздного использования итераторов:

```
use std::iter::{once, repeat};

let fizzes = repeat("").take(2).chain(once("fizz")).cycle();
let buzzes = repeat("").take(4).chain(once("buzz")).cycle();
```

```
let fizzes_buzzes = fizzes.zip(buzzes);

let fizz_buzz = (1..100).zip(fizzes_buzzes)
    .map(|tuple|
        match tuple {
            (i, ("", "")) => i.to_string(),
            (_, (fizz, buzz)) => format!("{}{}", fizz, buzz)
        });

for line in fizz_buzz {
    println!("{}", line);
}
```

Это детская игра в слова, которая иногда встречается в вопросах, задаваемых на интервью для программистов. Игроки по очереди считают, заменяя любое число, делящееся на три, словом «fizz», а любое число, делящееся на пять, – словом «buzz». Числа, делящиеся на 3 и на 5, превращаются в «fizzbuzz».

Потребление итераторов

До сих пор мы рассматривали создание итераторов и их адаптацию, а теперь покажем, как итераторы потребляются.

Конечно, итератор можно потребить в цикле `for` или вызвав метод `next` явно, но есть много типичных задач, которые не хотелось бы решать снова и снова. Характеристика `Iterator` предоставляет широкий спектр методов для решения многих из них.

Простое аккумулирование: count, sum, product

Метод `count` получает объекты от итератора, пока тот не вернет `None`, а затем сообщает, сколько объектов получилось. Следующая короткая программа подсчитывает количество строк в стандартном потоке ввода:

```
use std::io::prelude::*;

fn main() {
    let stdin = std::io::stdin();
    println!("{}", stdin.lock().lines().count());
}
```

Методы `sum` и `product` вычисляют соответственно сумму и произведение объектов итератора, которые должны быть числами, целыми или с плавающей точкой:

```
fn triangle(n: u64) -> u64 {
    (1..n+1).sum()
}
assert_eq!(triangle(20), 210);

fn factorial(n: u64) -> u64 {
    (1..n+1).product()
}
assert_eq!(factorial(20), 2432902008176640000);
```

(Методы `sum` и `product` можно обобщить и на другие типы, реализовав характеристики `std::iter::Sum` и `std::iter::Product`, которые в этой книге не описываются.)

max, min

Методы `min` и `max` характеристики `Iterator` возвращают соответственно наименьший и наибольший объекты, порождаемый итератором. Тип объекта итератора должен реализовывать характеристику `std::cmp::Ord`, чтобы объекты можно было сравнивать. Например:

```
assert_eq!([-2, 0, 1, 0, -2, -5].iter().max(), Some(&1));
assert_eq!([-2, 0, 1, 0, -2, -5].iter().min(), Some(&-5));
```

Эти методы на самом деле возвращают значение типа `Option<Self::Item>`, т. е. могут возвращать `None`, если итератор не порождает объектов.

В разделе «Сравнение на равенство» главы 12 отмечалось, что типы с плавающей точкой `f32` и `f64` реализуют только характеристику `std::cmp::PartialOrd`, но не `std::cmp::Ord`, так что с помощью методов `min` и `max` невозможно вычислить наименьший или наибольший члены последовательности чисел с плавающей точкой. Этот аспект Rust не вызывает особого восторга у пользователей, но так сделано сознательно: не понятно, что эти функции должны делать со значениями `NaN`. Простое игнорирование могло бы замаскировать более серьезные проблемы в коде.

Если вы знаете, как хотите обрабатывать значения `NaN`, то можете воспользоваться методами `max_by` и `min_by`, позволяющими задать собственную функцию сравнения.

max_by, min_by

Методы `max_by` и `min_by` возвращают соответственно минимальный и максимальный объекты, порожденные итератором, применяя заданную вами функцию сравнения:

```
use std::cmp::{PartialOrd, Ordering};

// Сравнить два значения типа f64. Если хотя бы одно из них NaN, паниковать.
fn cmp(lhs: &&f64, rhs: &&f64) -> Ordering {
    lhs.partial_cmp(rhs).unwrap()
}

let numbers = [1.0, 4.0, 2.0];
assert_eq!(numbers.iter().max_by(cmp), Some(&4.0));
assert_eq!(numbers.iter().min_by(cmp), Some(&1.0));

let numbers = [1.0, 4.0, std::f64::NAN, 2.0];
assert_eq!(numbers.iter().max_by(cmp), Some(&4.0)); // паника
```

(Двойные ссылки в параметрах `cmp` возникают, потому что `numbers.iter()` порождает ссылки на элементы, а затем `max_by` и `min_by` передают замыканию ссылки на объекты итератора.)

max_by_key, min_by_key

Методы `max_by_key` и `min_by_key` характеристики `Iterator` позволяют выбрать соответственно максимальный и минимальный объекты по результатам применения замыкания. Замыкание может выбрать какое-то поле объекта или выполнить вычисление с объектом. Поскольку часто нас интересуют данные, ассоциированные

с максимумом или минимумом, а не только сам экстремум, то эти функции часто оказываются полезнее `min` и `max`. Вот их сигнатуры:

```
fn min_by_key<B: Ord, F>(self, f: F) -> Option<Self::Item>
    where Self: Sized, F: FnMut(&Self::Item) -> B;

fn max_by_key<B: Ord, F>(self, f: F) -> Option<Self::Item>
    where Self: Sized, F: FnMut(&Self::Item) -> B;
```

Иначе говоря, имея замыкание, которое принимает объект и возвращает значение произвольного упорядоченного типа `B`, метод возвращает объект, для которого это замыкание вернуло максимальное или минимальное значение типа `B`, или `None`, если не было порождено ни одного объекта.

Например, если в таблице городов требуется найти города с наибольшей и наименьшей численностью населения, то можно написать такой код:

```
use std::collections::HashMap;

let mut populations = HashMap::new();
populations.insert("Portland",  583_776);
populations.insert("Fossil",        449);
populations.insert("Greenhorn",       2);
populations.insert("Boring",      7_762);
populations.insert("The Dalles", 15_340);

assert_eq!(populations.iter().max_by_key(|&(_name, pop)| pop),
    Some((&"Portland", &583_776)));
assert_eq!(populations.iter().min_by_key(|&(_name, pop)| pop),
    Some((&"Greenhorn", &2)));
```

Замыкание `|&(_name, pop)| pop` применяется к каждому объекту, порождаемому итератором, и возвращает значение, используемое для сравнения, – в данном случае численность населения. А в качестве значения возвращается объект целиком, а не только то, что вернуло замыкание. (Естественно, если такие запросы выполняются часто, то следовало бы воспользоваться более эффективным способом поиска записей, чем линейный просмотр всей таблицы.)

Сравнение последовательностей

Операторы `<` и `==` можно использовать для сравнения строк, векторов и срезок при условии, что отдельные элементы допускают сравнение. Хотя итераторы не поддерживают операторов сравнения Rust, они все же предлагают такие методы, как `eq` и `lt`, которые выполняют, по сути, то же самое: выбирают по одному элементу из каждого итератора и сравнивают их, пока не станет ясен результат. Например:

```
let packed =  "Helen of Troy";
let spaced =  "Helen   of    Troy";
let obscure = "Helen of Sandusky"; // симпатичная личность, хотя и не знаменитая

assert!(packed != spaced);
assert!(packed.split_whitespace().eq(spaced.split_whitespace()));

// true, потому что ' ' < 'o'.
assert!(spaced < obscure);

// true, потому что 'Troy' > 'Sandusky'.
assert!(spaced.split_whitespace().gt(obscure.split_whitespace()));
```

Обращения к split_whitespace возвращают итераторы для обхода разделенных пробелами слов строки. Применение методов eq и gt к этим итераторам означает пословное, а не посимвольное сравнение. Это возможно, потому что тип &str реализует характеристики PartialOrd и PartialEq.

Итераторы предоставляют методы eq и ne для сравнения на равенство и неравенство, а также методы lt, le, gt и ge для сравнения на больше-меньше. Методы cmp и partial_cmp ведут себя как соответственные методы характеристик Ord и PartialOrd.

any и all

Методы any и all применяют замыкание к каждому объекту, порождаемому итератором, и возвращают true, если замыкание возвращает true хотя бы для одного или для всех объектов соответственно:

```
let id = "Iterator";

assert!( id.chars().any(char::is_uppercase));
assert!(!id.chars().all(char::is_uppercase));
```

Эти методы потребляют ровно столько объектов, сколько необходимо для принятия решения. Например, если замыкание вернет true для какого-то объекта, то any сразу же вернет true, не запрашивая больше никаких объектов у итератора.

position, rposition и ExactSizeIterator

Метод position применяет замыкание к каждому объекту итератора и возвращает индекс первого объекта, для которого это замыкание вернет true. Точнее, возвращается значение типа Option: если замыкание не возвращает true ни для одного объекта, то position возвращает None. Выборка элементов прекращается, как только замыкание вернет true. Например:

```
let text = "Xerxes";
assert_eq!(text.chars().position(|c| c == 'e'), Some(1));
assert_eq!(text.chars().position(|c| c == 'z'), None);
```

Метод rposition ведет себя так же, только поиск начинается с правой стороны. Например:

```
let bytes = b"Xerxes";
assert_eq!(bytes.iter().rposition(|&c| c == b'e'), Some(4));
assert_eq!(bytes.iter().rposition(|&c| c == b'X'), Some(0));
```

Для метода rposition необходим обратимый итератор, чтобы можно было выбирать элементы с правой стороны последовательности. Кроме того, необходим точноразмерный итератор, чтобы индексы имели такие же значения, как если бы их присваивал метод position, начинающий индексирование с 0. Точноразмерным называется итератор, реализующий характеристику std::iter::ExactSizeIterator:

```
pub trait ExactSizeIterator: Iterator {
    fn len(&self) -> usize { ... }
    fn is_empty(&self) -> bool { ... }
}
```

Метод `len` возвращает количество оставшихся объектов, а метод `is_empty` возвращает `true`, если итерирование завершилось.

Естественно, не каждый итератор заранее знает, сколько объектов он породит; в примерах выше итератор `chars` для типа `&str` этого не знает (UTF-8 – кодировка с переменной длиной), поэтому к строкам метод `rposition` неприменим. Но итератор для обхода массива байтов точно знает длину массива, поэтому может реализовать `ExactSizeIterator`.

fold

Метод `fold` – очень общее средство аккумулирования некоторого результата по всей последовательности объектов, порождаемых итератором. Получив начальное значение, которое мы будем называть аккумулятором, и замыкание, метод `fold` повторно применяет это замыкание к текущему значению аккумулятора и следующему объекту итератора. Значение, возвращенное замыканием, считается новым аккумулятором и передается замыканию вместе со следующим объектом. Последнее значение аккумулятора и есть результат, возвращаемый методом `fold`. Если последовательность пуста, то `fold` возвращает начальный аккумулятор.

Многие другие методы потребления объектов итератора можно записать с помощью `fold`:

```
let a = [5, 6, 7, 8, 9, 10];

assert_eq!(a.iter().fold(0, |n, _| n+1), 6);       // count
assert_eq!(a.iter().fold(0, |n, i| n+i), 45);      // sum
assert_eq!(a.iter().fold(1, |n, i| n*i), 151200);  // product

// max
assert_eq!(a.iter().fold(i32::min_value(), |m, &i| std::cmp::max(m, i)),
           10);
```

Сигнатура метода `fold` такова:

```
fn fold<A, F>(self, init: A, f: F) -> A
    where Self: Sized, F: FnMut(A, Self::Item) -> A;
```

Здесь `A` – тип аккумулятора: такой тип имеет аргумент `init`, а также первый аргумент и возвращаемое значение замыкания и значение, возвращаемое самим методом `fold`.

Отметим, что владение значениями аккумулятора передается замыканию и от него, так что `fold` можно использовать, даже если тип аккумулятора некопируемый:

```
let a = ["Pack ", "my ", "box ", "with ",
         "five ", "dozen ", "liquor ", "jugs"];

let pangram = a.iter().fold(String::new(),
                            |mut s, &w| { s.push_str(w); s });
assert_eq!(pangram, "Pack my box with five dozen liquor jugs");
```

nth

Метод `nth` принимает индекс `n`, пропускает такое количество объектов итератора и возвращает следующий объект или `None`, если последовательность обрывается раньше. Вызов `.nth(0)` эквивалентен вызову `.next()`.

Метод не принимает владения итератором как адаптер, поэтому его можно вызывать многократно.

```
let mut squares = (0..10).map(|i| i*i);

assert_eq!(squares.nth(4), Some(16));
assert_eq!(squares.nth(0), Some(25));
assert_eq!(squares.nth(6), None);
```

Вот его сигнатура:

```
fn nth(&mut self, n: usize) -> Option<Self::Item>
    where Self: Sized;
```

last

Метод `last` потребляет объекты до тех пор, пока итератор не вернет `None`, а затем возвращает последний объект. Если итератор не порождает никаких объектов, то `last` возвращает `None`. Сигнатура метода такова:

```
fn last(self) -> Option<Self::Item>;
```

Например:

```
let squares = (0..10).map(|i| i*i);
assert_eq!(squares.last(), Some(81));
```

Этот код потребляет все объекты итератора, начиная с первого, даже если итератор обратим. Если имеется обратимый итератор и мы не хотим потреблять всех его объектов, то следует вместо этого написать `iter.rev().next()`.

find

Метод `find` запрашивает у итератора объекты и возвращает первый, для которого заданное замыкание возвращает `true`, или `None`, если последовательность обрывается до того, как будет найден подходящий объект. Вот его сигнатура:

```
fn find<P>(&mut self, predicate: P) -> Option<Self::Item>
    where Self: Sized,
          P: FnMut(&Self::Item) -> bool;
```

Например, в применении к таблице городов с указанием численности населения из раздела «max_by_key, min_by_key» выше можно написать такой код:

```
assert_eq!(populations.iter().find(|&(_name, &pop)| pop > 1_000_000),
           None);
assert_eq!(populations.iter().find(|&(_name, &pop)| pop > 500_000),
           Some((&"Portland", &583_776)));
```

Ни в одном городе из таблицы население не превышает миллиона человек, но есть один город, в котором проживает более полумиллиона.

Построение коллекций: collect и FromIterator

На протяжении всей книги мы использовали метод `collect` для построения векторов, содержащих объекты итератора. Например, в «Кратком обзоре Rust» мы вызывали функцию `std::env::args()`, чтобы получить итератор для обхода аргу-

ментов командной строки программы, а затем – метод `collect` этого итератора, чтобы собрать из них вектор:

```
let args: Vec<String> = std::env::args().collect();
```

Но метод `collect` не ограничен векторами, с его помощью можно построить любую стандартную библиотечную коллекцию, если только итератор порождает объекты подходящего типа:

```
use std::collections::{HashSet, BTreeSet, LinkedList, HashMap, BTreeMap};

let args: HashSet<String> = std::env::args().collect();
let args: BTreeSet<String> = std::env::args().collect();
let args: LinkedList<String> = std::env::args().collect();

// Чтобы собрать отображение, нужны пары (ключ, значение), так что в этом примере
// мы сшиваем последовательность строк с последовательностью целых чисел.
let args: HashMap<String, usize> = std::env::args().zip(0..).collect();
let args: BTreeMap<String, usize> = std::env::args().zip(0..).collect();

// и так далее
```

Естественно, сам метод `collect` ничего не знает о том, как строить значения различных типов. Но если какой-то тип коллекции, например `Vec` или `HashMap`, знает, как конструировать себя из итератора, то он реализует характеристику `std::iter::FromIterator`, удобным фасадом которой и является метод `collect`:

```
trait FromIterator<A>: Sized {
    fn from_iter<T: IntoIterator<Item=A>>(iter: T) -> Self;
}
```

Если тип коллекции реализует `FromIterator<A>`, то ее статический метод `from_iter` строит значение данного типа из итерируемого значения, порождающего объекты типа `A`.

В простейшем случае реализация могла бы построить пустую коллекцию, а затем добавлять в нее объекты итератора по одному. Например, так работает реализация `FromIterator` для типа `std::collections::LinkedList`.

Но для некоторых типов можно поступить более эффективно. Так, конструирование вектора из итератора `iter` могло бы быть таким:

```
let mut vec = Vec::new();
for item in iter {
    vec.push(item)
}
vec
```

Но и это не идеально: по мере роста вектора его буфер может увеличиваться, что требует обращения к распределителю кучи и копирования существующих элементов. Векторы предпринимают кое-какие алгоритмические меры для уменьшения этих накладных расходов, но если бы существовал способ с самого начала выделить буфер правильного размера, то изменять размер впоследствии вообще не понадобилось бы.

Тут-то и приходит на помощь метод `size_hint` характеристики `Iterator`:

```
trait Iterator {
    ...
```

```
    fn size_hint(&self) -> (usize, Option<usize>) {
        (0, None)
    }
}
```

Этот метод возвращает нижнюю границу и факультативно верхнюю границу числа порождаемых итератором объектов. В определении по умолчанию нижняя граница равна 0, а верхняя граница – `None`, что означает «Понятия не имею», но многие итераторы могут улучшить эти оценки. Так, итератор по диапазону `Range` точно знает, сколько объектов породит, и то же можно сказать об итераторах для обхода `Vec` или `HashMap`. Такие итераторы предоставляют специализированное определение `size_hint`.

Эти границы и есть та информация, которая необходима реализации `FromIterator` для типа `Vec`, чтобы с самого начала выбрать правильный размер буфера нового вектора. При вставке все равно проверяется, есть ли в буфере место, так что даже если предположение было неверным, то пострадает только производительность, но не безопасность. Другие типы могут поступить аналогично: например, `HashSet` и `HashMap` тоже переопределяют `Iterator::size_hint` с целью выбора правильного начального размера хеш-таблицы.

Одно замечание о выводе типов: в начале этого раздела был приведен странный код, в котором один и тот же вызов, `std::env::args().collect()`, порождает коллекции четырех разных типов в зависимости от контекста. Типом значения, возвращаемого методом `collect`, является его параметрический тип, поэтому первые два вызова эквивалентны таким:

```
let args = std::env::args().collect::<String>();
let args = std::env::args().collect::<HashSet<String>>();
```

Но коль скоро существует всего один тип, который мог бы иметь аргумент `collect`, то механизм вывода типов в Rust подставит его. Указывая явно аргумент `args`, мы как раз и создаем такую ситуацию.

Характеристика Extend

Если тип реализует характеристику `std::iter::Extend`, то его метод `extend` добавляет объекты итерируемого значения в коллекцию:

```
let mut v: Vec<i32> = (0..5).map(|i| 1 << i).collect();
v.extend(&[31, 57, 99, 163]);
assert_eq!(v, &[1, 2, 4, 8, 16, 31, 57, 99, 163]);
```

Поскольку все стандартные коллекции реализуют `Extend`, то у всех них есть такой метод. Есть он и в типе `String`. Но у массивов и срезок, длина которых фиксирована, метода `extend` нет.

Вот определение этой характеристики:

```
trait Extend<A> {
    fn extend<T>(&mut self, iter: T)
        where T: IntoIterator<Item=A>;
}
```

Видно, что она очень похожа на `std::iter::FromIterator`: та создает новую коллекцию, а эта расширяет существующую. На самом деле несколько реализаций

FromIterator в стандартной библиотеке просто создают новую пустую коллекцию, а затем вызывают метод extend, чтобы ее заполнить. Так работает, например, реализация FromIterator для типа std::collections::LinkedList:

```
impl<T> FromIterator<T> for LinkedList<T> {
    fn from_iter<I: IntoIterator<Item = T>>(iter: I) -> Self {
        let mut list = Self::new();
        list.extend(iter);
        list
    }
}
```

partition

Метод partition разбивает все множество объектов итератора на две коллекции, применяя замыкания для решения о том, куда поместить объект.

```
let things = ["doorknob", "mushroom", "noodle", "giraffe", "grapefruit"];

// Удивительный факт: названия объектов живой природы всегда начинаются
// буквой с нечетным номером.
let (living, nonliving): (Vec<&str>, _)
    = things.iter().partition(|name| name.as_bytes()[0] & 1 != 0);

assert_eq!(living,    vec!["mushroom", "giraffe", "grapefruit"]);
assert_eq!(nonliving, vec!["doorknob", "noodle"]);
```

Как и collect, метод partition может построить коллекцию любого типа (хотя типы обеих коллекций должны быть одинаковы). И, как и в случае collect, необходимо задать тип возвращаемого значения: в примере выше явно указан тип переменных living и nonliving, а механизм вывода типов определяет подходящие параметрические типы.

Вот сигнатура метода partition:

```
fn partition<B, F>(self, f: F) -> (B, B)
where Self: Sized,
    B: Default + Extend<Self::Item>,
    F: FnMut(&Self::Item) -> bool;
```

Если collect требует, чтобы тип возвращаемого значения реализовывал характеристику FromIterator, то partition настаивает, чтобы этот тип реализовывал std::default::Default (все коллекции Rust реализуют ее, возвращая пустую коллекцию) и std::default::Extend.

Почему partition не просто расщепляет итератор на два, а вместо этого строит две коллекции? Одна из причин – в том, что объекты, полученные от исходного итератора, но еще не запрошенные от одного из двух расщепленных итераторов, нужно где-то буферизовать, а значит, все равно придется создавать какую-то коллекцию, только на внутреннем уровне. Но есть и более глубокая причина, связанная с безопасностью. Расщепление итератора на два означало бы, что обе части разделяют один и тот же исходный итератор. Но, чтобы итератор можно было использовать, он должен быть изменяемым, поэтому исходный итератор по необходимости должен быть и разделяемым, и изменяемым, а это как раз то, что запрещают правила безопасности Rust.

Реализация собственных итераторов

Мы можем реализовать характеристики `IntoIterator` и `Iterator` для своих собственных типов, и тогда получим в свое распоряжение все адаптеры и потребители, описанные в этой главе, а также огромный объем кода в других библиотеках и крейтах, рассчитанного на работу со стандартным интерфейсом итераторов. В этом разделе мы продемонстрируем простой итератор для обхода диапазона, а также более сложный итератор для обхода двоичного дерева.

Рассмотрим следующий тип диапазона (упрощенный вариант стандартного библиотечного типа `std::ops::Range<T>`):

```
struct I32Range {
    start: i32,
    end: i32
}
```

Для обхода `I32Range` нужно хранить два элемента состояния: текущее значение и конечное значение, на котором обход должен обрываться. Для этой цели отлично подходит сам тип `I32Range`: `start` используется как следующее значение, а `end` – как конечное. Следовательно, характеристику `Iterator` можно реализовать так:

```
impl Iterator for I32Range {
    type Item = i32;
    fn next(&mut self) -> Option<i32> {
        if self.start >= self.end {
            return None;
        }
        let result = Some(self.start);
        self.start += 1;
        result
    }
}
```

Этот итератор порождает объекты типа `i32`, так что это и есть тип `Item`. Если итерирование завершилось, то `next` возвращает `None`, в противном случае порождает следующее значение и обновляет состояние, готовясь к следующему вызову.

Как обычно, цикл `for` использует метод `IntoIterator::into_iter` для преобразования своего операнда в итератор. Но стандартная библиотека предоставляет всеобъемлющую реализацию `IntoIterator` для любого типа, реализующего `Iterator`, так что `I32Range` готов к работе:

```
let mut pi = 0.0;
let mut numerator = 1.0;

for k in (I32Range { start: 0, end: 14 }) {
    pi += numerator / (2*k + 1) as f64;
    numerator /= -3.0;
}
pi *= f64::sqrt(12.0);

// IEEE 754 специфицирует этот результат точно.
assert_eq!(pi as f32, std::f32::consts::PI);
```

Но `I32Range` – это особый случай, когда итерируемое значение и итератор представлены одним и тем же типом. Многие случаи не так просты. Вот, например, тип двоичного дерева из главы 10.

```
enum BinaryTree<T> {
    Empty,
    NonEmpty(Box<TreeNode<T>>)
}

struct TreeNode<T> {
    element: T,
    left: BinaryTree<T>,
    right: BinaryTree<T>
}
```

Классический способ обхода дерева подразумевает рекурсию с использованием стека вызовов функций для отслеживания текущей позиции в дереве и еще не посещенных узлов. Но при реализации характеристики `Iterator` для `BinaryTree<T>` каждый вызов `next` должен породить ровно одно значение и вернуть управление. Для учета узлов дерева, которые еще предстоит породить, итератор должен вести собственный стек. Вот один из возможных типов итератора для `BinaryTree`:

```
use self::BinaryTree::*;

// Состояние симметричного обхода `BinaryTree`.
struct TreeIter<'a, T: 'a> {
    // Стек ссылкок на узлы дерева. Поскольку мы используем методы
    // `push` и `pop` типа Vec, вершина стека — это конец вектора.
    //
    // Узел, который итератор посетит следующим, находится на вершине
    // стека, а его еще не посещенные предки - ниже. Если стек пуст,
    // то итерирование завершено.
    unvisited: Vec<&'a TreeNode<T>>
}
```

Заталкивание в стек узлов, расположенных вдоль левого ветви поддерева, – распространенная операция, поэтому определим для нее метод в типе `TreeIter`:

```
impl<'a, T: 'a> TreeIter<'a, T> {
    fn push_left_edge(&mut self, mut tree: &'a BinaryTree<T>) {
        while let NonEmpty(ref node) = *tree {
            self.unvisited.push(node);
            tree = &node.left;
        }
    }
}
```

Располагая этим вспомогательным методом, мы можем реализовать в типе `BinaryTree` метод `iter`, возвращающий итератор для обхода дерева:

```
impl<T> BinaryTree<T> {
    fn iter(&self) -> TreeIter<T> {
        let mut iter = TreeIter { unvisited: Vec::new() };
        iter.push_left_edge(self);
        iter
    }
}
```

Метод `iter` строит пустую структуру `TreeIter`, а затем вызывает `push_left_edge`, чтобы инициализировать стек. Самый левый узел оказывается на вершине, как того и требуют правила работы со стеком `unvisited`.

Взяв за образец стандартную библиотеку, мы можем далее реализовать характеристику `IntoIterator` для разделяемой ссылки на дерево с помощью обращения к `BinaryTree::iter`:

```
impl<'a, T: 'a> IntoIterator for &'a BinaryTree<T> {
    type Item = &'a T;
    type IntoIter = TreeIter<'a, T>;
    fn into_iter(self) -> Self::IntoIter {
        self.iter()
    }
}
```

В определении `IntoIter` в качестве типа итератора для `&BinaryTree` указан `TreeIter`.

Наконец, в реализации `Iterator` мы приступаем собственно к обходу дерева. Как и в случае метода `iter` типа `BinaryTree`, поведение метода итератора `next` определяется правилами работы со стеком:

```
impl<'a, T> Iterator for TreeIter<'a, T> {
    type Item = &'a T;
    fn next(&mut self) -> Option<&'a T> {
        // Найти узел, который должна породить эта итерация,
        // или завершить итерирование.
        let node = match self.unvisited.pop() {
            None => return None,
            Some(n) => n
        };

        // Следующим узлом является самый левый потомок правого сына этого узла,
        // поэтому поместить в стек путь к нему.
        self.push_left_edge(&node.right);

        // Породить ссылку на значение этого узла.
        Some(&node.element)
    }
}
```

Если стек пуст, итерирование завершается. В противном случае `node` содержит ссылку на узел, который следует посетить сейчас; этот вызов возвращает ссылку на его поле `element`. Но сначала мы должны изменить состояние итератора, так чтобы он указывал на следующий узел. Если у этого узла есть правое поддерево, то следующим должен быть самый левый узел этого поддерева, поэтому мы вызываем `push_left_edge`, чтобы поместить его самого и всех его еще не посещенных предков в стек. Если же правого поддерева нет, то `push_left_edge` ничего не делает, а это как раз то, что нам нужно: мы можем рассчитывать на то, что новая вершина стека будет содержать первого непосещенного предка `node`, если таковой имеется.

Реализовав характеристики `IntoIterator` и `Iterator`, мы наконец можем использовать цикл `for` для обхода `BinaryTree` по ссылке:

```
fn make_node<T>(left: BinaryTree<T>, element: T, right: BinaryTree<T>)
                -> BinaryTree<T>
```

```
{
    NonEmpty(Box::new(TreeNode { left, element, right }))
}

// Строим небольшое дерево.
let subtree_l = make_node(Empty, "mecha", Empty);
let subtree_rl = make_node(Empty, "droid", Empty);
let subtree_r = make_node(subtree_rl, "robot", Empty);
let tree = make_node(subtree_l, "Jaeger", subtree_r);

// Обходим его.
let mut v = Vec::new();
for kind in &tree {
    v.push(*kind);
}
assert_eq!(v, ["mecha", "Jaeger", "droid", "robot"]);
```

На рис. 15.1 показано, как ведет себя стек в процессе обхода демонстрационного дерева. На каждом шаге следующий подлежащий посещению узел находится на вершине стека, а все его еще не посещенные предки – под ним.

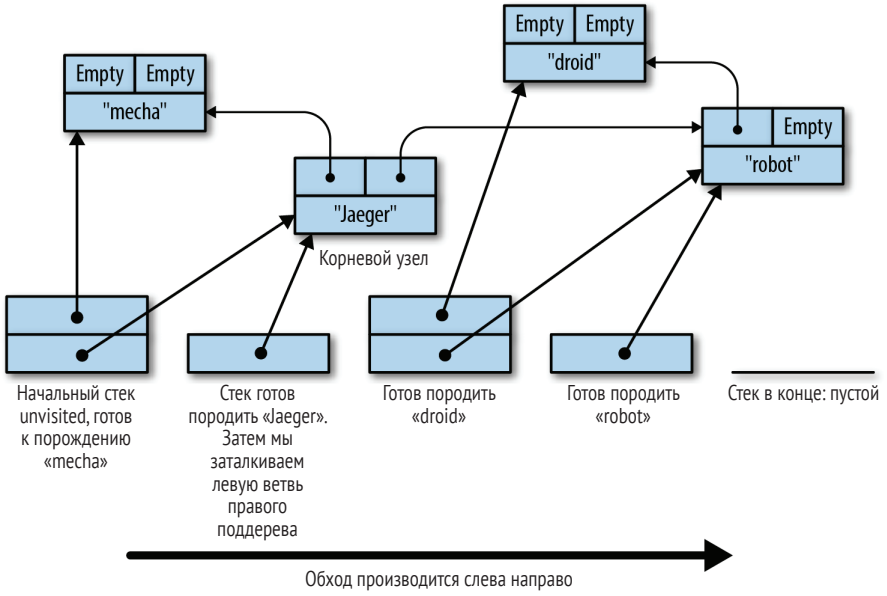


Рис. 15.1 ❖ Обход двоичного дерева

К деревьям применимы все обычные адаптеры и потребители итераторов:

```
assert_eq!(tree.iter()
           .map(|name| format!("mega-{}", name))
           .collect::<Vec<_>>(),
           vec!["mega-mecha", "mega-Jaeger",
                "mega-droid", "mega-robot"]);
```

Глава 16

Коллекции

> Все мы ведем себя как демон Максвелла. Организмы организуются. Причина, по которой рассудительные физики вот уже два столетия берегут эту мультяшную фантазию, – повседневный опыт. Мы сортируем почту, строим песочные замки, собираем пазлы, отделяем зерна от плевел, переставляем шахматные фигурки, собираем марки, расставляем книги по алфавиту, рождаем симметрию, сочиняем сонеты и сонаты, наводим порядок в комнатах – и все, что мы делаем, не требует большой энергии, если только удается применить разум.
>
> *— Джеймс Глейк*
> *«Информация: история, теория, поток»*

В стандартную библиотеку Rust входят несколько *коллекций* – универсальных типов для хранения данных в памяти. Выше мы уже пользовались коллекциями, например `Vec` и `HashMap`. В этой главе мы не только подробнее рассмотрим методы этих двух типов, но и полдюжины других стандартных коллекций. Но сначала поговорим о нескольких систематических различиях между коллекциями в Rust и в других языках.

Во-первых, это повсеместные передача владения и заимствование. Передача владения в Rust используется, чтобы избежать глубокого копирования значений. Именно поэтому метод `Vec<T>::push(item)` принимает аргумент по значению, а не по ссылке. Значение передается вектору. Рисунки в главе 4 показывают, как это работает на практике: помещение строку `String` в вектор `Vec<String>` производится быстро, потому что Rust не копирует символьных данных строк, а передает владение ими.

Во-вторых, в Rust не бывает ошибок, связанных с недействительными данными, – своего рода проблема висячего указателя, когда память коллекции перераспределяется из-за изменения размера или других модификаций, а программа хранит указатель на данные внутри нее. Такие ошибки – еще один источник неопределенного поведения в C++, они же приводят к исключению `ConcurrentModificationException` даже в языках с безопасным доступом к памяти. Благодаря контролю заимствования Rust предотвращает такие ошибки еще на этапе компиляции.

Наконец, в Rust нет значения `null`, его место занимают значения типа `Option`.

Если не считать этих отличий, коллекции в Rust оправдывают ожидания. Если вы опытный программист и куда-то торопитесь, то можете пропустить следующий материал, но прочитайте раздел «Записи».

Обзор

В табл. 16.1 перечислено восемь стандартных коллекций Rust. Всё это универсальные типы.

Таблица 16.1. Стандартные коллекции

Коллекция	Описание	Аналогичный тип коллекции в		
		C++	Java	Python
`Vec<T>`	Растущий массив	`vector`	`ArrayList`	`list`
`VecDeque<T>`	Двусторонняя очередь (растущий кольцевой буфер)	`deque`	`ArrayDeque`	`collections.deque`
`LinkedList<T>`	Двусвязный список	`list`	`LinkedList`	–
`BinaryHeap<T>` `where T: Ord`	Max-пирамида	`priority_queue`	`PriorityQueue`	`heapq`
`HashMap<K, V>` `where K: Eq + Hash`	Хеш-таблица ключей и значений	`unordered_map`	`HashMap`	`dict`
`BTreeMap<K, V>` `where K: Ord`	Отсортированная таблица ключей и значений	`map`	`TreeMap`	–
`HashSet<T>` `where T: Eq + Hash`	Хеш-таблица	`unordered_set`	`HashSet`	`set`
`BTreeSet<T>` `where T: Ord`	Отсортированное множество	`set`	`TreeSet`	–

Типы коллекций `Vec<T>`, `HashMap<K, V>` и `HashSet<T>` наиболее полезны в общем случае. У остальных ниши более узкие. В этой главе мы обсудим все типы.

- `Vec<T>` – растущий массив значений типа `T`, выделяемый в куче. Примерно половина главы посвящена типу `Vec` и его многочисленным методам.
- Тип `VecDeque<T>` похож на `Vec<T>`, но более пригоден в качестве очереди, обслуживаемой по принципу «первым пришел, первым ушел». Он поддерживает эффективное добавление и удаление значений с обеих сторон. Но ценой за это является небольшое замедление всех остальных операций.
- Тип `LinkedList<T>` поддерживает быстрый доступ к началу и концу списка, как `VecDeque<T>`, а также быструю конкатенацию списков. Но в общем случае `LinkedList<T>` медленнее, чем `Vec<T>` и `VecDeque<T>`.
- `BinaryHeap<T>` – очередь с приоритетами. Значения в `BinaryHeap` организованы так, чтобы операции поиска и удаления максимального значения производились эффективно.
- `HashMap<K, V>` – таблица пар ключей и значений. Поиск по ключу производится быстро. Порядок хранения записей не определен.
- Тип `BTreeMap<K, V>` похож на `HashMap<K, V>`, но записи отсортированы по ключу. В коллекции `BTreeMap<String, i32>` записи хранятся в порядке сравнения строк. Если порядок записей несуществен, то лучше использовать `HashMap`, поскольку она работает быстрее.
- `HashSet<T>` – множество значений типа `T`. Добавление и удаление производятся быстро, как и ответ на вопрос, принадлежит ли заданное значение множеству.
- Тип `BTreeSet<T>` похож на `HashSet<T>`, но элементы хранятся в отсортированном виде. И снова `HashSet` работает быстрее.

Тип Vec<T>

Мы предполагаем некоторое знакомство с типом `Vec`, поскольку он уже не раз использовался в книге. Базовые сведения приведены в разделе «Векторы» главы 3. А здесь мы наконец опишем его методы и их работу во всей полноте.

Проще всего создать вектор с помощью макроса `vec!`:

```
// Создать пустой вектор
let mut numbers: Vec<i32> = vec![];

// Создать вектор с заданными элементами
let words = vec!["step", "on", "no", "pets"];
let mut buffer = vec![0u8; 1024];  // вектор, состоящий из 1024 нулевых байтов
```

Как было сказано в главе 4, вектор состоит их трех полей: длина, емкость и указатель на область памяти в куче, где хранятся элементы. На рис. 16.1 показано, как размещаются в памяти созданные выше векторы. Пустой вектор `numbers` первоначально имеет емкость 0. Память в куче для него не выделяется до момента добавления первого элемента.

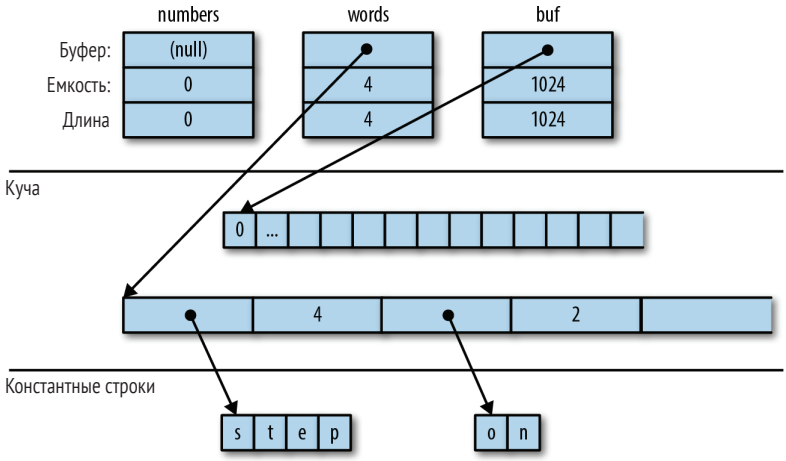


Рис. 16.1 ❖ Размещение вектора в памяти.
Каждый элемент `words` – значение типа `&str`, состоящее из указателя и длины

Как и все коллекции, тип `Vec` реализует характеристику `std::iter::FromIterator`, т. е. вектор можно создать из любого итератора методом `.collect()`:

```
// Преобразовать другую коллекцию в вектор.
let my_vec = my_set.into_iter().collect::<Vec<String>>();
```

Доступ к элементам

Получение элементов из массива, срезки или вектора по индексу производится единообразно:

```
// Получить ссылку на элемент
let first_line = &lines[0];
```

```
// Получить копию элемента
let fifth_number = numbers[4];          // требуется характеристика Copy
let second_line = lines[1].clone();     // требуется характеристика Clone

// Получить ссылку на срезку
let my_ref = &buffer[4..12];

// Получить копию срезки
let my_copy = buffer[4..12].to_vec();  // требуется характеристика Clone
```

Во всех случаях выход индекса за границы приводит к панике.

Rust строго относится к числовым типам и не делает исключения для векторов. Длина и индекс вектора должны иметь тип `usize`. Попытка назначить индексу тип `u32`, `u64` или `isize` приводит к ошибке. При необходимости можно воспользоваться приведением `n as usize`.

Несколько методов дают простой доступ к отдельным элементам вектора или срезки. Отметим, что все методы срезки применимы также к массивам и векторам.

- **`slice.first()`** возвращает ссылку на первый элемент срезки `slice`, если таковой существует.
  Возвращаемое значение имеет тип `Option<&T>`, т. е. равно `None`, если `slice` пуста, и `Some(&slice[0])`, если не пуста.

  ```
  if let Some(item) = v.first() {
      println!("We got one! {}", item);
  }
  ```

- **`slice.last()`** похож, но возвращает ссылку на последний элемент.
- **`slice.get(index)`** возвращает ссылку `Some` на элемент `slice[index]`, если таковой существует. Если `slice` содержит меньше `index+1` элементов, то возвращается `None`.

  ```
  let slice = [0, 1, 2, 3];
  assert_eq!(slice.get(2), Some(&2));
  assert_eq!(slice.get(4), None);
  ```

- **`slice.first_mut()`**, **`slice.last_mut()`** и **`slice.get_mut(index)`** – варианты описанных выше методов, заимствующие изменяемые ссылки.

  ```
  let mut slice = [0, 1, 2, 3];
  {
      let last = slice.last_mut().unwrap();   // тип last: &mut i32
      assert_eq!(*last, 3);
      *last = 100;
  }
  assert_eq!(slice, [0, 1, 2, 100]);
  ```

  Поскольку возврат `T` по значению означал бы передачу владения, методы, обращающиеся к элементам, обычно возвращают эти элементы по ссылке. Исключение составляет метод `.to_vec()`, который создает копии:

- **`slice.to_vec()`** клонирует всю срезку и возвращает новый вектор.

  ```
  let v = [1, 2, 3, 4, 5, 6, 7, 8, 9];
  assert_eq!(v.to_vec(),
             vec![1, 2, 3, 4, 5, 6, 7, 8, 9]);
  ```

```
assert_eq!(v[0..6].to_vec(),
           vec![1, 2, 3, 4, 5, 6]);
```

Этот метод доступен, только если элементы допускают клонирование, т. е. `where T: Clone`.

Итерирование

Векторы и срезки допускают итерирование по значению или по ссылке, следуя схеме, описанной в разделе «Реализации характеристики IntoIterator» главы 15.

- Итерирование `Vec<T>` порождает объекты типа `T`. Элементы передаются из вектора по одному, в результате чего вектор потребляется.
- Итерирование по значению типа `&[T; N]`, `&[T]` или `&Vec<T>` – т. е. ссылки на массив, срезку или вектор – порождает объекты типа `&T` – ссылки на отдельные элементы без передачи владения.
- Итерирование по значению типа `&mut [T; N]`, `&mut [T]` или `&mut Vec<T>` порождает объекты типа `&mut T`.

У массивов, срезок и векторов имеются также методы `.iter()`и `.iter_mut()` для создания итераторов, порождающих ссылки на элементы.

Некоторые более хитрые способы итерирования по срезке мы обсудим в разделе «Расщепление» ниже.

Увеличение и уменьшение вектора

Длиной массива, срезки или вектора называется число элементов в нем.

- **`slice.len()`** возвращает длину `slice` в виде значения типа `usize`.
- **`slice.is_empty()`** возвращает `true`, если `slice` не содержит элементов (т. е. `slice.len() == 0`).

Остальные рассматриваемые в этом разделе методы относятся к увеличению и уменьшению векторов. Их нет у массивов и срезок, размер которых фиксирован при создании.

Все элементы вектора хранятся в непрерывной области памяти, выделенной в куче. *Емкостью* вектора называется максимальное число элементов, помещающихся в этой области. При нормальных обстоятельствах тип `Vec` сам управляет емкостью – когда места не хватает, автоматически выделяется буфер большего размера, и в него перемещаются элементы. Но есть также несколько методов для явного управления емкостью:

- **`Vec::with_capacity(n)`** создает новый пустой вектор емкости `n`;
- **`vec.capacity()`** возвращает емкость `vec` в виде значения типа `usize`. Всегда соблюдается условие `vec.capacity() >= vec.len()`;
- **`vec.reserve(n)`** проверяет, что емкость вектора достаточна для размещения еще `n` элементов, т. е. что `vec.capacity()` не меньше `vec.len() + n`. Если места достаточно, то метод больше ничего не делает. В противном случае выделяется память для буфера большего размера и в него копируется содержимое вектора;
- **`vec.reserve_exact(n)`** похож на `vec.reserve(n)`, но не выделяет больше памяти, чем требуется для размещения `n` элементов, «на вырост». После выполнения метода `vec.capacity()` в точности равно `vec.len() + n`;
- **`vec.shrink_to_fit()`** пытается освободить лишнюю память, если `vec.capacity()` больше `vec.len()`.

В типе `Vec<T>` имеется много методов для добавления или удаления элементов, изменяющих длину вектора. Каждый из них принимает аргумент `self` по изменяемой ссылке.

Следующие два метода соответственно добавляют и удаляют одно значение в конце вектора:

- **`vec.push(value)`** добавляет значение `value` в конец `vec`;
- **`vec.pop()`** удаляет последний элемент и возвращает его. Возвращаемое значение имеет тип `Option<T>`. Оно равно `Some(x)`, если был удален элемент `x`, и `None`, если вектор был пуст в момент вызова.

Отметим, что `.push()` принимает аргумент по значению, а не по ссылке. И `.pop()` возвращает само удаленное значение, а не ссылку на него. Это справедливо и для большинства других методов, рассматриваемых в этом разделе. Они передают владение значениями вектору и отнимают у него владение.

Следующие два метода соответственно добавляют и удаляют значение в любом месте вектора:

- **`vec.insert(index, value)`** вставляет значение `value` в позицию `vec[index]`, сдвигая существующие в срезке `vec[index..]` значения на одну позицию вправо, чтобы освободить место. Паникует, если `index > vec.len()`;
- **`vec.remove(index)`** удаляет и возвращает элемент `vec[index]`, сдвигая существующие в срезке `vec[index+1..]` значения на одну позицию влево, чтобы закрыть лакуну. Паникует, если `index >= vec.len()`, поскольку в этом случае не существует удаляемого элемента `vec[index]`.

Чем длиннее вектор, тем дольше выполняется операция. Если метод `vec.remove(0)` приходится вызывать часто, подумайте об использовании `VecDeque` вместо `Vec`.

Методы `.insert()` и `.remove()` работают тем дольше, чем больше элементов сдвигается.

Следующие три метода изменяют длину вектора на указанное значение:

- **`vec.resize(new_len, value)`** переустанавливает длину `vec`, так что она становится равна `new_len`. Если при этом длина `vec` увеличивается, то новые позиции заполняются копиями значения `value`. Тип элемента должен реализовывать характеристику `Clone`;
- **`vec.truncate(new_len)`** уменьшает длину `vec` до `new_len`, уничтожая элементы в диапазоне `vec[new_len..]`. Если `vec.len()` уже меньше или равна `new_len`, то метод не делает ничего;
- **`vec.clear()`** удаляет все элементы `vec`. Это то же самое, что `vec.truncate(0)`.

Четыре метода добавляют или удаляют сразу несколько значений:

- **`vec.extend(iterable)`** добавляет все объекты из заданного значения `iterable` в конец вектора `vec` с сохранением порядка. Можно назвать этот метод многозначной версией `.push()`. Аргумент `iterable` может быть любым значением, реализующим характеристику `IntoIterator<Item=T>`.

  Этот метод настолько полезен, что для него существует стандартная характеристика, `Extend`, которую реализуют все стандартные коллекции. К сожалению, из-за этого `rustdoc` смешивает метод `.extend()` в одну кучу с другими методами характеристик в конце сгенерированной HTML-документации, так что отыскать его там трудно. Просто запомните, что он есть! Дополнительные сведения см. в разделе «Характеристика Extend» главы 15;

- **vec.split_off(index)** похож на vec.truncate(index), но возвращает вектор Vec<T>, содержащий значения, удаленные из vec. Можно считать, что это многозначная версия метода .pop();
- **vec.append(&mut vec2)**, где vec2 – другой вектор типа Vec<T>, перемещает все элементы из vec2 в vec. После его выполнения vec2 оказывается пустым. Это похоже на vec.extend(vec2) с тем отличием, что vec2 по завершении все-таки остается и его емкость не изменяется;
- **vec.drain(range)**, где range – диапазон, например .. или 0..4. Удаляет диапазон vec[range] из vec и возвращает итератор для обхода удаленных элементов.

Существует также несколько необычных методов для избирательного удаления элементов вектора:

- **vec.retain(test)** удаляет все элементы, не удовлетворяющие заданному условию. Аргумент test – это функция или замыкание, реализующее характеристику FnMut(&T) -> bool. Для каждого элемента vec вызывается test(&element), и если возвращенное значение равно false, то элемент удается из вектора и уничтожается.
  Если отвлечься от производительности, то это то же самое, что:

```
vec = vec.into_iter().filter(test).collect();
```

- **vec.dedup()** уничтожает дубликаты. Метод похож на утилиту uniq, имеющуюся в ОС Unix. Он ищет в vec одинаковые соседние элементы и уничтожает все, кроме одного.

```
let mut byte_vec = b"Mississsssissippi".to_vec();
byte_vec.dedup();
assert_eq!(&byte_vec, b"Misisipi");
```

  Обратите внимание, что в результирующей строке осталось две буквы 's'. Этот метод удаляет только *соседние* дубликаты. Для удаления всех дубликатов есть три пути: предварительно отсортировать вектор, переместить все данные во множество или (если нужно сохранить порядок элементов) воспользоваться приемом на основе метода .retain():

```
let mut byte_vec = b"Mississsssissippi".to_vec();
let mut seen = HashSet::new();
byte_vec.retain(|r| seen.insert(*r));
assert_eq!(&byte_vec, b"Misp");
```

  Этот прием работает, потому что .insert() возвращает false, если множество уже содержит элемент, который мы пытаемся вставить.
- **vec.dedup_by(same)** – то же, что vec.dedup(), но вместо оператора == используется функция или замыкание same(&mut elem1, &mut elem2), которая решает, следует ли считать два элемента одинаковыми.
- **vec.dedup_by_key(key)** – то же, что vec.dedup(), но два элемента считаются одинаковыми, если key(&mut elem1) == key(&mut elem2).
  Например, если errors имеет тип Vec<Box<Error>>, то можно написать:

```
// Удалить ошибки с одинаковыми сообщениями.
errors.dedup_by_key(|err| err.description().to_string());
```

Из всех методов, рассмотренных в этом разделе, только `.resize()` клонирует значения. Остальные передают значения из одного места в другое.

Соединение

Два метода работают с «массивами массивов». Под этим понимается любой массив, срезка или вектор, элементы которого сами являются массивами, срезками или векторами.

- **`slices.concat()`** возвращает вектор, образованный конкатенацией всех срезок.

```
assert_eq!([[1, 2], [3, 4], [5, 6]].concat(),
vec![1, 2, 3, 4, 5, 6]);
```

- **`slices.join(&separator)`** – то же самое, только между срезками вставляется значение `separator`.

```
assert_eq!([[1, 2], [3, 4], [5, 6]].join(&0),
vec![1, 2, 0, 3, 4, 0, 5, 6]);
```

Расщепление

Легко поместить сразу много неизменяемых ссылок в массив, срезку или вектор:

```
let v = vec![0, 1, 2, 3];
let a = &v[i];
let b = &v[j];

let mid = v.len() / 2;
let front_half = &v[..mid];
let back_half = &v[mid..];
```

Получить вектор изменяемых ссылок сложнее:

```
let mut v = vec![0, 1, 2, 3];
let a = &mut v[i];
let b = &mut v[j];  // ошибка: нельзя заимствовать `v` в качестве изменяемой
                    //         ссылки более одного раза одновременно
```

Rust запрещает это, потому что если `i == j`, то `a` и `b` будут двумя изменяемыми ссылками на одно и то же целое число, а это нарушение правил безопасности Rust.

В Rust есть несколько методов, которые позволяют заимствовать изменяемые ссылки на две или более частей массива, срезки или вектора одновременно. В отличие от приведенного выше кода, эти методы безопасны, т. к. расщепляют данные на непересекающиеся участки. Многие из этих методов также удобны для работы с неизменяемыми ссылками, так что для каждого имеются два варианта: с `mut` и без `mut`.

Эти методы иллюстрируются на рис. 16.2. Ни один из них не модифицирует массив, срезку или вектор напрямую, они просто возвращают новые ссылки на части данных.

- **`slice.iter()`** и **`slice.iter_mut()`** порождают ссылки на каждый элемент срезки `slice`. Мы рассматривали их в разделе «Итерирование» выше;
- **`slice.split_at(index)`** и **`slice.split_at_mut(index)`** разбивают срезку на две и возвращают пару. Вызов `slice.split_at(index)` эквивалентен `(&slice[..index], &slice[index..])`. Оба метода паникуют, если `index` выходит за границы;

- **`slice.split_first()`** и **`slice.split_first_mut()`** также возвращают пару: ссылку на первый элемент (`slice[0]`) и ссылку на оставшийся участок срезки (`slice[1..]`).
  Возвращаемое значение `.split_first()` имеет тип `Option<(&T, &[T])>`; результат равен `None`, если срезка `slice` пуста;
- методы **`slice.split_last()`** и **`slice.split_last_mut()`** аналогичны, но отщепляют не первый, а последний элемент.
  Возвращаемое значение `.split_last()` имеет тип `Option<(&T, &[T])>`;
- **`slice.split(is_sep)`** и **`slice.split_mut(is_sep)`** расщепляют `slice` на одну или несколько подсрезок, применяя функцию или замыкание `is_sep`, чтобы определить, в каком месте производить расщепление. Возвращается итератор для обхода подсрезок.
  В процессе потребления итератор вызывает метод `is_sep(&element)` для каждого элемента срезки. Если `is_sep(&element)` возвращает `true`, то элемент является разделителем. Разделители не включаются ни в какую выходную подсрезку.
  Результат всегда содержит по крайней мере одну подсрезку и еще столько, сколько имеется разделителей. Пустые подсрезки включаются, когда разделители соседствуют или примыкают к концам `slice`;
- **`slice.splitn(n, is_sep)`** и **`slice.splitn_mut(n, is_sep)`** делают то же самое, но порождают не более `n` подсрезок. После того как `n-1` срезок найдено, `is_sep` больше не вызывается. Последняя подсрезка содержит все оставшиеся элементы.

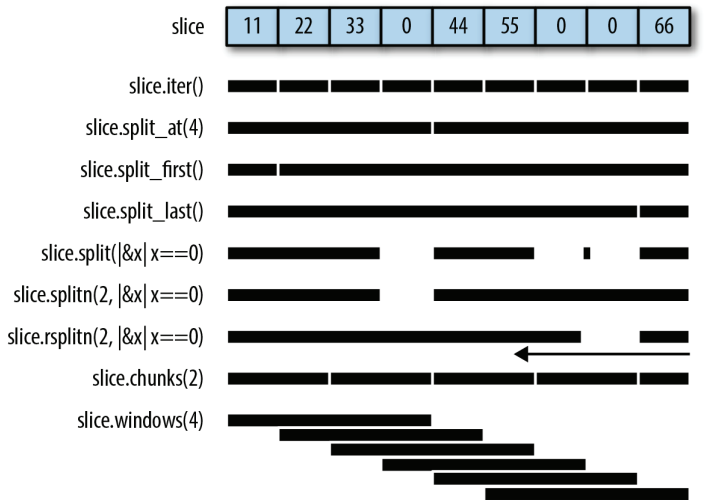


Рис. 16.2 ❖ Методы расщепления.
Небольшой прямоугольник в результате `slice.split()` – пустая срезка, появившаяся из-за двух соседних разделителей. Также обратите внимание, что, в отличие от всех остальных методов, `rsplitn` порождает результат в порядке от конца к началу

- **slice.rsplitn(n, is_sep)** и **slice.rsplitn_mut(n, is_sep)** ведут себя как .splitn() и .splitn_mut() – с тем отличием, что срезка просматривается в обратном порядке. То есть эти методы производят расщепление в местах *последних*, а не первых n-1 разделителей, и подсрезки порождаются, начиная с конца.
- **slice.chunks(n)** и **slice.chunks_mut(n)** возвращают итератор для обхода непересекающихся подсрезок длины n.
  Если slice.len() не делится нацело на n, то последняя подсрезка будет иметь длину меньше n.

Существует еще один метод для обхода подсрезок.

- **slice.windows(n)** возвращает итератор, который ведет себя как «скользящее окно» по данным в срезке slice. Он порождает подсрезки, охватывающие n последовательных элементов slice. Первым порождается значение &slice[0..n], вторым – &slice[1..n+1] и т. д.
  Если n больше, чем длина slice, то не порождается ни одной срезки. Если n равно 0, то метод паникует.
  Например, если days.len() == 31, то мы можем породить все 7-дневные промежутки в days, вызвав days.windows(7).
  Скользящее окно размера 2 полезно для изучения того, как ряд данных изменяется при переходе от одной точки к следующей:

```
let changes = daily_high_temperatures
                  .windows(2)              // получить температуры за соседние дни
                  .map(|w| w[1] - w[0])    // насколько изменилась температура?
                  .collect::<Vec<_>>();
```

  Поскольку подсрезки пересекаются, не существует варианта этого метода, который возвращал бы изменяемые ссылки.

Перестановка элементов

Существует метод для перестановки местами двух элементов:

- **slice.swap(i, j)** переставляет местами элементы slice[i] и slice[j].

У векторов имеется родственный метод для эффективного удаления любого элемента:

- **vec.swap_remove(i)** удаляет и возвращает элемент vec[i]. Метод аналогичен vec.remove(i) – с тем отличием, что вместо сдвига остальных элементов вектора для заполнения лакуны он просто перемещает в лакуну последний элемент вектора. Это полезно, если порядок хранящихся в векторе элементов не играет роли.

Сортировка и поиск

Срезки предлагают три метода, выполняющих сортировку:

- **slice.sort()** сортирует элементы в порядке возрастания. Этот метод присутствует, только если тип элементов реализует характеристику Ord.
- **slice.sort_by(cmp)** сортирует элементы slice, применяя функцию или замыкание cmp для задания порядка сортировки. Тип cmp должен реализовывать характеристику Fn(&T, &T) -> std::cmp::Ordering.
  Вручную кодировать cmp неприятно, лучше по возможности делегировать работу методу .cmp(), например:

```
students.sort_by(|a, b| a.last_name.cmp(&b.last_name));
```

Чтобы сортировать по одному полю и использовать второе для разрешения неоднозначностей, сравнивайте кортежи:

```
students.sort_by(|a, b| {
    let a_key = (&a.last_name, &a.first_name);
    let b_key = (&b.last_name, &b.first_name);
    a_key.cmp(&b_key)
});
```

- **`slice.sort_by_key(key)`** сортирует элементы `slice` в порядке возрастания ключа, задаваемого функцией или замыканием `key`. Тип `key` должен реализовывать характеристику `Fn(&T) -> K where K: Ord`.
  Это полезно, когда `T` содержит одно или несколько порядковых полей, т. е. допускает сортировку несколькими способами.

```
// Сортировать по среднему баллу в порядке возрастания.
students.sort_by_key(|s| s.grade_point_average());
```

Отметим, что значения ключа сортировки не кэшируются в процессе сортировки, так что функция `key` может вызываться более *n* раз.
По техническим причинам `key(element)` не может возвращать ссылки, заимствованные у элемента. Такой код работать не будет:

```
students.sort_by_key(|s| &s.last_name); // ошибка: невозможно вывести время жизни
```

Rust не может определить времена жизни. Но в таких случаях несложно перейти на использование `.sort_by()`.

Все три метода производят устойчивую сортировку.

Чтобы отсортировать данные в обратном порядке, следует использовать метод `sort_by`, в котором замыкание `cmp` меняет местами аргументы. Записав аргументы в виде `|b, a|`, а не `|a, b|`, мы получим сортировку в обратном порядке. Можно вместо этого применить к результату сортировки метод `.reverse()`.

- **`slice.reverse()`** обращает срезку на месте.

В отсортированной срезке можно эффективно производить поиск.

- Методы **`slice.binary_search(&value)`**, **`slice.binary_search_by(&value, cmp)`** и **`slice.binary_search_by_key(&value, key)`** ищут значение `value` в заданной отсортированной срезке `slice`. Отметим, что `value` передается по ссылке.
- Все эти методы возвращают значение типа `Result<usize, usize>`. Оно равно `Ok(index)`, если `slice[index]` равен `value` в смысле указанного критерия сравнения. Если такого индекса не существует, то возвращается значение `Err(insertion_point)` – такое, что вставка `value` в позицию `insertion_point` сохранит порядок сортировки.

Разумеется, двоичный поиск работает, только если срезка действительно отсортирована в смысле указанного критерия сравнения. В противном случае результат не определен – мусор на входе порождает мусор на выходе.

Поскольку для типов `f32` и `f64` существует значение `NaN`, они не реализуют характеристику `Ord`, и, следовательно, метод `slice.sort()` для них не работает. То же самое относится к методам `.sort_by_key()`, `.binary_search()` и `binary_search_by_key()`. Если вам нужны аналогичные методы для работы с данными с плавающей точкой, воспользуйтесь крейтом `ord_subset`.

Существует один метод для поиска в неотсортированном векторе.

- **`slice.contains(&value)`** возвращает `true`, если существует элемент `slice`, равный `value`. Он просто проверяет один элемент срезки за другим, пока не найдет совпадения. Значение `value` передается по ссылке.

Чтобы найти местоположение значения в срезке, как делает метод `array.indexOf(value)` в JavaScript, воспользуйтесь итератором:

```
slice.iter().position(|x| *x == value)
```

В результате возвращается значение типа `Option<usize>`.

Сравнение срезок

Если тип `T` поддерживает операторы `==` и `!=` (характеристику `PartialEq`), то их поддерживают также массивы `[T; N]`, срезки `[T]` и векторы `Vec<T>`. Две срезки равны, если они имеют одинаковую длину и соответственные элементы равны. То же относится к массивам и векторам.

Если `T` поддерживает операторы `<`, `<=`, `>` и `>=` (характеристику `PartialOrd`), то их поддерживают также массивы, срезки и векторы элементов типа `T`. Сравнение срезок производится в лексикографическом порядке.

Для типичных операций сравнения срезок предназначены два вспомогательных метода:

- **`slice.starts_with(other)`** возвращает `true`, если срезка `slice` начинается последовательностью значений, равных элементам срезки `other`:

```
assert_eq!([1, 2, 3, 4].starts_with(&[1, 2]), true);
assert_eq!([1, 2, 3, 4].starts_with(&[2, 3]), false);
```

- **`slice.ends_with(other)`** аналогичен, но производит проверку с конца `slice`:

```
assert_eq!([1, 2, 3, 4].ends_with(&[3, 4]), true);
```

Случайные элементы

Случайные числа не входят в стандартную библиотеку Rust. Крейт `rand`, предназначенный для этой цели, предлагает два метода для выборки случайных элементов из массива, срезки или вектора:

- **`rng.choose(slice)`** возвращает ссылку на случайный элемент срезки. Подобно `slice.first()` и `slice.last()`, он возвращает значение типа `Option<&T>`, равное `None`, только если срезка пуста;
- **`rng.shuffle(slice)`** случайным образом переставляет элементы срезки на месте. Срезка передается по изменяемой ссылке.

Это методы характеристики `rand::Rng`, поэтому для их вызова понадобится генератор случайных чисел а `Rng`. По счастью, его легко получить, вызвав функцию `rand::thread_rng()`. Чтобы перетасовать вектор `my_vec`, можно написать:

```
use rand::{Rng, thread_rng};

thread_rng().shuffle(&mut my_vec);
```

В Rust отсутствуют ошибки недействительности

В большинстве основных языков программирования имеются коллекции и итераторы, и во всех действует тот или иной вариант правила: не модифицируй коллекцию во время обхода.

Например, в Python эквивалентом вектора является список:

```
my_list = [1, 3, 5, 7, 9]
```

Попробуем удалить из списка `my_list` все значения, большие 4:

```
for index, val in enumerate(my_list):
    if val > 4:
        del my_list[index]  # ошибка: модификация списка во время обхода

print(my_list)
```

(Функция `enumerate` в Python эквивалентна методу `.enumerate()` в Rust.)

Как ни странно, эта программа печатает `[1, 3, 7]`. Но ведь семь больше четырех. Почему же это число осталось? Это ошибка недействительности: программа модифицирует данные во время их обхода, тем самым делая итератор *недействительным*. В Java это приведет к исключению, в C++ – к неопределенному поведению. В Python такое поведение определено корректно, но интуитивно неочевидно: итератор пропускает элемент. Переменная `val` никогда не принимает значение `7`.

Попытаемся воспроизвести эту ошибку в Rust:

```
fn main() {
    let mut my_vec = vec![1, 3, 5, 7, 9];
    for (index, &val) in my_vec.iter().enumerate() {
        if val > 4 {
            my_vec.remove(index); // ошибка: нельзя заимствовать `my_vec` как изменяемую
                                  // ссылку
        }
    }
    println!("{:?}", my_vec);
}
```

Естественно, Rust отклоняет эту программу на этапе компиляции. При вызове `my_vec.iter()` происходит заимствование разделяемой (неизменяемой) ссылки на вектор. Эта ссылка живет столько же, сколько итератор, – до конца цикла `for`. Мы не можем модифицировать вектор путем вызова `my_vec.remove(index)`, пока существует неизменяемая ссылка.

Хорошо, когда тебе указывают на ошибку, но все же хотелось бы найти правильный способ получить требуемое поведение! Проще всего исправить ошибку, воспользовавшись вызовом `my_vec.retain(|&val| val <= 4);`. Можно вместо этого пойти по тому же пути, что в Python или любом другом языке: создать новый вектор с помощью адаптера `filter`.

Тип VecDeque<T>

Тип `Vec` поддерживает эффективное добавление и удаление элементов только в конце вектора. Если программе нужно место для размещения значений, «стоящих в очереди», то `Vec` может оказаться слишком медленным.

Тип Rust `std::collections::VecDeque<T>` представляет собой двустороннюю очередь (иногда ее называют «дек»). Он поддерживает эффективные операции добавления и удаления в начало и в конец.

- **deque.push_front(value)** добавляет значение в начало очереди.
- **deque.push_back(value)** добавляет значение в конец очереди (этот метод используется гораздо чаще, чем .push_front(), потому что принято считать, что элементы добавляются в конец очереди, а удаляются из начала, как в очереди покупателей к кассе).
- **deque.pop_front()** удаляет и возвращает элемент в начале очереди. Возвращаемое значение имеет тип Option<T> и равно None, если очередь пуста (по аналогии с методом vec.pop()).
- **deque.pop_back()** удаляет и возвращает элемент в конце очереди. Возвращаемое значение также имеет тип Option<T>.
- **deque.front()** и **deque.back()** работают, как vec.first() и vec.last(). Возвращают ссылку соответственно на первый и последний элементы очереди. Возвращаемое значение имеет тип Option<&T> и равно None, если очередь пуста.
- **deque.front_mut()** и **deque.back_mut()** работают, как vec.first_mut() и vec.last_mut(). Возвращаемое значение имеет тип Option<&mut T>.

Тип VecDeque реализован в виде кольцевого буфера, как показано на рис. 16.3.

Как и в случае Vec, выделяется одна область в памяти для хранения элементов. Но, в отличие от Vec, данные необязательно начинаются в начале области и могут «заворачиваться» в конце. На рисунке показана двусторонняя очередь, содержащая элементы ['A', 'B', 'C', 'D', 'E'] в указанном порядке. В типе VecDeque имеются закрытые поля, обозначенные на рисунке start и stop, в них запоминается положение начала и конца данных.

Добавление значения с любой стороны очереди может привести к одному из трех действий: занятие одной из неиспользованных позиций, показанных серым цветом, заворачиванию или выделению большей области памяти.

VecDeque сам управляет заворачиванием, так что вам об этом думать не надо. На рисунке показано, как Rust обеспечивает быстрое выполнение .pop_front().

Часто при работе с очередью используются только методы .push_back() и .pop_front(). Статические методы VecDeque::new() и VecDeque::with_capacity(n) для создания очередей аналогичны таким же методам типа Vec. Многие методы Vec реализованы также для VecDeque: .len() и .is_empty(), .insert(index, value) и .remove(index), .extend(iterable) и т. д.

Двусторонние очереди, как и векторы, можно обходить по значению, по разделяемой или по изменяемой ссылке. Для них определены три итераторных метода: .into_iter(), .iter() и .iter_mut(). Их можно индексировать, как обычно: deque[index].

Но поскольку элементы двусторонней очереди не хранятся подряд, этот тип не наследует всех методов срезок. Один из способов применить к данным двусторонней очереди операции вектора и срезки состоит в том, чтобы преобразовать VecDeque в Vec, выполнить операцию, а затем произвести обратное преобразование:

- Тип Vec<T> реализует характеристику From<VecDeque<T>>, поэтому **Vec::from(deque)** преобразует двустороннюю очередь в вектор. Это занимает время порядка $O(n)$, поскольку может потребовать переупорядочения всех элементов.
- Тип VecDeque<T> реализует характеристику From<Vec<T>>, так что **VecDeque::from(vec)** преобразует вектор в двустороннюю очередь. Эта операция также занимает время порядка $O(n)$, но обычно выполняется быстрее, даже если

вектор большой, потому что область памяти, выделенную вектору, можно просто передать новой очереди.

Этот метод позволяет легко создать двустороннюю очередь с заданными элементами, несмотря на отсутствие стандартного макроса `vec_deque![]`:

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

let v = VecDeque::from(vec![1, 2, 3, 4]);
```

Тип LinkedList<T>

Связный список – еще один способ хранения последовательности значений. Для каждого значения выделяется отдельная область памяти в куче, как показано на рис. 16.4.

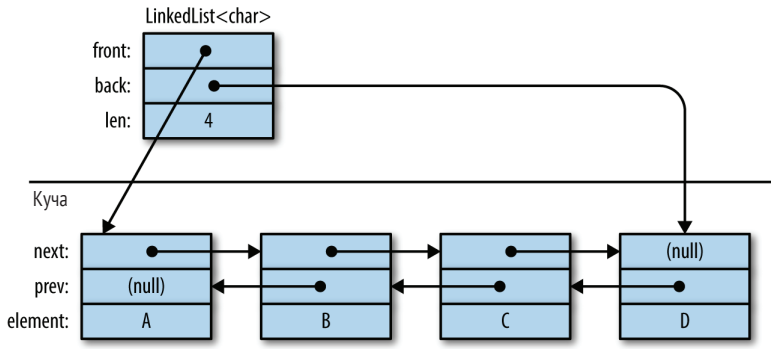


Рис. 16.4 ❖ Размещение `LinkedList<char>` в памяти

Тип `std::collections::LinkedList<T>` в Rust представляет двусвязный список. Он поддерживает подмножество методов `VecDeque`. Присутствуют все методы для работы с началом (`front`) и концом (`back`) последовательности, итераторные методы, например `LinkedList::new()`, и еще несколько. Но методы для доступа к элементам по индексу отсутствуют, т. к. в применении к связному списку они неэффективны.

В версии Rust 1.17 тип `LinkedList` не имел методов для удаления диапазонов элементов из списка или для вставки элемента в заданную позицию списка. Такой API выглядит неполным.

В настоящее время основным преимуществом `LinkedList`, по сравнению с `VecDeque`, является возможность очень быстрого объединения списков. Метод `list.append(&mut list2)`, который перемещает все элементы из одного списка в другой, сводится к изменению нескольких указателей, для чего достаточно постоянного времени. Метод `append` в типах `Vec` и `VecDeque` иногда вынужден перемещать много значений из одной области кучи в другую.

Тип BinaryHeap<T>

`BinaryHeap` – коллекция, организованная так, что наибольшее значение всегда находится в начале очереди. Ниже перечислены три наиболее употребительных метода `BinaryHeap`:

- `heap.push(value)` добавляет значение в пирамиду;
- `heap.pop()` удаляет и возвращает наибольшее значение. Возвращаемое значение имеет тип `Option<T>`, и `None` означает, что пирамида пуста;
- `heap.peek()` возвращает ссылку на наибольшее значение. Возвращаемое значение имеет тип `Option<&T>`.

`BinaryHeap` поддерживает также подмножество методов `Vec`, в том числе: `BinaryHeap::new()`, `.len()`, `.is_empty()`, `.capacity()`, `.clear()` и and `.append(&mut heap2)`.

Например, если инициализировать `BinaryHeap` последовательностью чисел:

```
use std::collections::BinaryHeap;

let mut heap = BinaryHeap::from(vec![2, 3, 8, 6, 9, 5, 4]);
```

то значение `9` окажется на вершине пирамиды:

```
assert_eq!(heap.peek(), Some(&9));
assert_eq!(heap.pop(), Some(9));
```

После удаления `9` пирамида реорганизуется так, что на вершине оказывается значение `8`, и так далее:

```
assert_eq!(heap.pop(), Some(8));
assert_eq!(heap.pop(), Some(6));
assert_eq!(heap.pop(), Some(5));
...
```

Конечно, тип `BinaryHeap` не ограничен одними числами. В пирамиде можно хранить значение любого типа, реализующего характеристику `Ord`.

Благодаря этим свойствам тип `BinaryHeap` полезен для реализации очереди работ. Мы можем определить структуру задачи, реализующую `Ord` на основе понятия приоритетности, так что более приоритетная задача оказывается больше (`Greater`) менее приоритетной. Затем создадим пирамиду `BinaryHeap` для хранения задач. Ее метод `.pop()` всегда возвращает самую важную задачу, которую программа должна выполнить следующей.

Примечание: тип `BinaryHeap` является итерируемым и располагает методом `.iter()`, но итераторы порождают элементы пирамиды в произвольном порядке, а не от наибольшего к наименьшему. Для потребления значений из `BinaryHeap` в порядке приоритета пользуйтесь циклом `while`:

```
while let Some(task) = heap.pop() {
    handle(task);
}
```

Типы HashMap<K, V> и BTreeMap<K, V>

Отображением называется коллекция пар ключ-значение (*записей*). В любых двух записях ключи различны, а все записи организованы так, что, зная ключ, можно эффективно найти соответствующее значение. Короче говоря, отображение – это таблица соответствия.

Rust предлагает два типа отображений: `HashMap<K, V>` и `BTreeMap<K, V>`. Многие их методы совпадают, а различие состоит в организации записей для быстрого поиска.

В случае `HashMap` ключи и значения хранятся в хеш-таблице, поэтому требуется, чтобы тип ключа `K` реализовывал стандартные характеристики `Hash` и `Eq`.

На рис. 16.5 показано размещение `HashMap` в памяти. Серые области не используются. Все ключи, значения и кэшированные хеш-коды хранятся в одной области памяти, выделенной в куче. По мере добавления записей рано или поздно придется выделить для таблицы область большего размера и переместить в нее все данные.

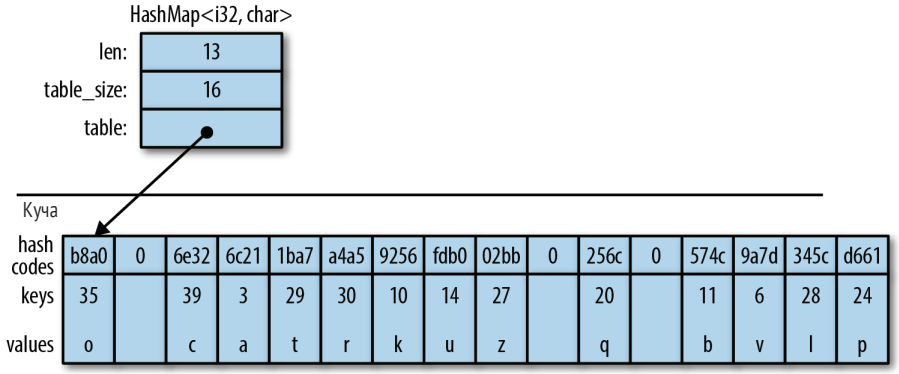


Рис. 16.5 ❖ Размещение `HashMap` в памяти

В случае `BTreeMap` записи хранятся упорядоченными по ключу в древовидной структуре, поэтому тип ключа `K` должен реализовывать характеристику `Ord`. На рис. 16.6 изображена организация `BTreeMap`. И снова серым цветом показаны неиспользуемые области, зарезервированные для будущего расширения.

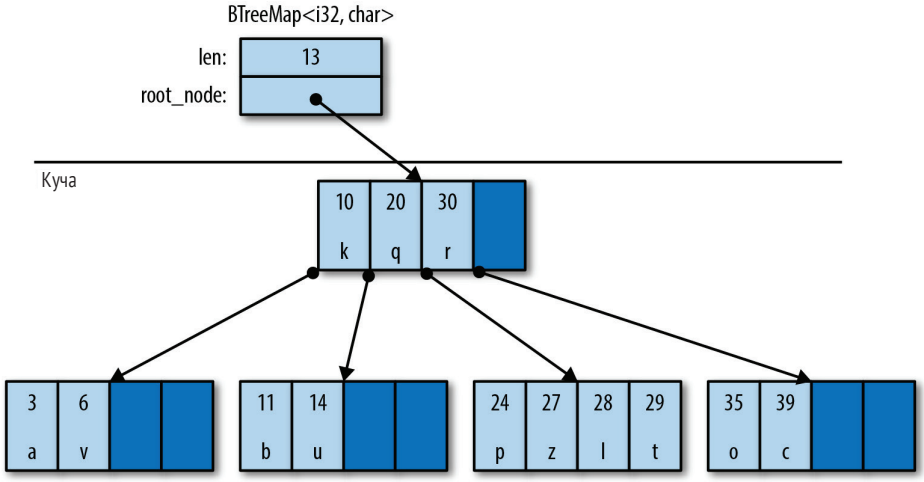


Рис. 16.6. ❖ Размещение `BTreeMap` в памяти

`BTreeMap` хранит записи в *узлах*. В большинстве узлов находятся только пары ключ-значение. В нелистовых узлах, в частности в корневом, имеется также мес-

то для указателей на дочерние узлы. Указатель, расположенный между узлами `(20, 'q')` и `(30, 'r')`, указывает на дочерний узел, содержащий ключи, большие `20` и меньшие `30`. При добавлении записи часто бывает необходимо сдвинуть некоторые хранящиеся в узле записи вправо, чтобы сохранить порядок сортировки, а иногда приходится выделять память для новых узлов.

Рисунок несколько упрощен, чтобы он мог поместиться на печатной странице. Так, в реальной структуре `BTreeMap` в узлах резервируется место для 11, а не для 4 записей.

В стандартной библиотеке Rust используются B-деревья, а не сбалансированные двоичные деревья, потому что на современном оборудовании B-деревья работают быстрее. В двоичном дереве количество операций сравнения при поиске может оказаться меньше, но поиск в B-дереве характеризуется лучшей *локальностью* – т. е. адреса памяти, к которым производятся обращения, находятся рядом, а не разбросаны по всей куче. Поэтому непопадание в кэш случается реже, а это дает значительный выигрыш в скорости.

Отображение можно создать несколькими способами:

- методы **`HashMap::new()`** и **`BTreeMap::new()`** создают новое пустое отображение;
- метод **`iter.collect()`** позволяет создать и заполнить `HashMap` или `BTreeMap` с заданными парами ключ-значение. `iter` должно быть итератором, реализующим характеристику `Iterator<Item=(K, V)>`;
- **`HashMap::with_capacity(n)`** создает пустое отображение типа `HashMap`, в котором можно разместить как минимум *n* записей. В таблице `HashMap`, как и в векторе, данные хранятся в непрерывной области памяти, поэтому для нее определено понятие емкости и соответствующие методы: `hash_map.capacity()`, `hash_map.reserve(additional)` и `hash_map.shrink_to_fit()`. В `BTreeMap` ничего этого нет.

В обоих типах `HashMap` и `BTreeMap` имеется общий набор методов для работы с ключами и значениями:

- **`map.len()`** возвращает число записей;
- **`map.is_empty()`** возвращает `true`, если в `map` нет ни одной записи;
- **`map.contains_key(&key)`** возвращает `true`, если в `map` имеется запись с данным ключом `key`;
- **`map.get(&key)`** ищет в `map` запись с данным ключом `key`. Если такая запись найдена, то возвращается `Some(r)`, где `r` – ссылка на соответствующее значение. В противном случае возвращается `None`;
- метод **`map.get_mut(&key)`** аналогичен, но возвращает изменяемую ссылку на значение.

  В общем случае отображение позволяет получать доступ для изменения хранящихся значений, но не ключей. Значения принадлежат вам, и вы можете делать с ними все что угодно. Ключи же принадлежат самому отображению; оно должно гарантировать их неизменность, поскольку от ключей зависит организация записей. Модификация ключа на месте была бы ошибкой;
- **`map.insert(key, value)`** вставляет запись `(key, value)` в отображение `map`. Если запись с ключом `key` уже существует, то старое значение перезаписывается новым значением `value`.

  Возвращается старое значение, если запись с таким ключом существовала. Тип возвращаемого значения – `Option<V>`;

- **`map.extend(iterable)`** обходит объекты `(K, V)` итерируемого значения `iterable` и вставляет в `map` каждую пару ключ-значение;
- **`map.append(&mut map2)`** перемещает все записи из `map2` в `map`. По завершении работы `map2` остается пустым;
- **`map.remove(&key)`** находит и удаляет запись с ключом `key` из `map`. Если такая запись существовала, возвращается удаленное значение. Тип возвращаемого значения – `Option<V>`;
- **`map.clear()`** удаляет все записи.

Отображение можно также опрашивать с помощью нотации с квадратными скобками: **`map[&key]`**. Иначе говоря, отображение реализует встроенную характеристику `Index`. Но если при этом оказывается, что запись с ключом `key` не существует, то возникает паника, как при доступе к несуществующему элементу массива. Поэтому такой синтаксис следует применять, только если искомая запись гарантированно существует.

Аргумент `key` методов `.contains_key()`, `.get()`, `.get_mut()` и `.remove()` не обязан иметь в точности тип `&K`. Эти методы принимают также типы, которые можно заимствовать от `K`. Разрешается вызывать `fish_map.contains_key("conger")` для типа `HashMap<String, Fish>`, несмотря на то что тип `"conger"` – не совсем `String`, поскольку `String` реализует характеристику `Borrow<&str>`. Подробнее см. раздел «Характеристики Borrow и BorrowMut» главы 13.

Поскольку в `BTreeMap<K, V>` записи хранятся отсортированными по ключу, то этот тип поддерживает дополнительную операцию:

- **`btree_map.split_at(&key)`** расщепляет дерево `btree_map` на два. Записи с ключами, меньшими `key`, остаются в `btree_map`. А этот метод возвращает новое дерево `BTreeMap<K, V>`, содержащее остальные записи.

Записи

И с `HashMap`, и с `BTreeMap` ассоциирован тип `Entry`. Смысл записей состоит в том, чтобы исключить лишние операции поиска в отображении. Например, следующий код получает запись о студенте или создает новую:

```
// Запись об этом студенте уже существует?
if !student_map.contains_key(name) {
    // Нет, создаем.
    student_map.insert(name.to_string(), Student::new());
}
// Теперь запись точно существует.
let record = student_map.get_mut(name).unwrap();
...
```

Все работает, только к отображению `student_map` производятся два или три обращения, и всякий раз выполняется одна и та же операция поиска.

Идея записи заключается в том, чтобы произвести поиск только один раз, получив при этом значение типа `Entry`, которое можно будет использовать для всех последующих операций. Следующий однострочный код эквивалентен приведенному выше, но поиск производится однократно:

```
let record = student_map.entry(name.to_string()).or_insert_with(Student::new);
```

Значение `Entry`, возвращенное при вызове `student_map.entry(name.to_string())`, играет роль изменяемой ссылки на то место внутри отображения, которое либо *занято* парой ключ-значение, либо *вакантно*, т. е. записи в нем еще нет. Если место вакантно, метод `.or_insert_with()` вставит новое значение `Student`. По большей части записи именно так и используются: кратко и выразительно.

Значения типа `Entry` всегда создаются одним и тем же методом:

- **`map.entry(key)`** возвращает запись `Entry` с заданным ключом `key`. Если такого ключа нет в отображении, то создается вакантная запись.

  Этот метод принимает аргумент `self` по изменяемой ссылке и возвращает значение `Entry` с таким же временем жизни:

  ```
  pub fn entry<'a>(&'a mut self, key: K) -> Entry<'a, K, V>
  ```

  У типа `Entry` есть параметрическое время жизни `'a`, потому что, по существу, это своеобразная заимствованная изменяемая ссылка на отображение. Пока значение `Entry` существует, оно обладает исключительным доступом к отображению.

  В разделе «Структуры, содержащие ссылки» главы 5 мы видели, как сохранить ссылку в типе и как это влияет на время жизни. Теперь мы видим, как это выглядит с точки зрения пользователя. Именно это и происходит с `Entry`.

  К сожалению, этому методу невозможно передать ссылку типа `&str`, если ключами отображения являются строки. Метод `.entry()` в этом случае требует настоящей строки `String`.

Тип `Entry` предоставляет два метода для заполнения вакантных записей:

- **`map.entry(key).or_insert(value)`** гарантирует, что `map` содержит запись с заданным ключом `key`, вставляя при необходимости новую запись со значением `value`. Он возвращает изменяемую ссылку на новое или существующее значение.

  Предположим, что требуется подсчитать голоса. Для этого можно написать:

  ```
  let mut vote_counts: HashMap<String, usize> = HashMap::new();
  for name in ballots {
      let count = vote_counts.entry(name).or_insert(0);
      *count += 1;
  }
  ```

  Метод `.or_insert()` возвращает изменяемую ссылку, так что `count` имеет тип `&mut usize`;

- метод **`map.entry(key).or_insert_with(default_fn)`** аналогичен, с тем отличием, что когда нужно создать новую запись, он вызывает функцию `default_fn()`, порождающую значение по умолчанию. Если в отображении уже существует запись с ключом `key`, то аргумент `default_fn` не используется.

  Пусть требуется узнать, какие слова встречаются в каких файлах. Мы можем написать такой код:

  ```
  // Это отображение для каждого слова содержит множество файлов, в которых оно
  // встречается.
  let mut word_occurrence: HashMap<String, HashSet<String>> = HashMap::new();
  for file in files {
      for word in read_words(file)? {
  ```

```
            let set = word_occurrence.entry(word).or_insert_with(HashSet::new);
                set.insert(file.clone());
            }
        }
```

Тип `Entry` – это перечисление, которое для `HashMap` определено следующим образом (для `BTreeMap` аналогично):

```
// (в std::collections::hash_map)
pub enum Entry<'a, K: 'a, V: 'a> {
    Occupied(OccupiedEntry<'a, K, V>),
    Vacant(VacantEntry<'a, K, V>)
}
```

Типы `OccupiedEntry` и `VacantEntry` имеют методы для вставки, удаления и получения записей без повторения начального поиска. Их можно найти в документации. Дополнительные методы иногда можно использовать для исключения одной-двух лишних операций поиска, но методы `.or_insert()` и `.or_insert_with()` покрывают типичные случаи.

Обход отображения

Отображение можно обойти несколькими способами:

- итерирование по значению (`for (k, v) in map`) порождает пары `(K, V)`. При этом отображение потребляется;
- итерирование по разделяемой ссылке (`for (k, v) in &map`) порождает пары `(&K, &V)`;
- итерирование по изменяемой ссылке (`for (k, v) in &mut map`) порождает пары `(&K, &mut V)`. (Как мы помним, не существует способа получить изменяемые ссылки на ключи отображения, потому что именно они отвечают за всю организацию структуры.)

Как и у векторов, у отображений имеются методы `.iter()` и `.iter_mut()`, которые возвращают итераторы по ссылке, делающие то же самое, что обход `&map` и `&mut map` соответственно. Кроме того:

- **`map.keys()`** возвращает итератор для обхода одних ключей по ссылке;
- **`map.values()`** возвращает итератор для обхода одних значений по ссылке;
- **`map.values_mut()`** возвращает итератор для обхода одних значений по изменяемой ссылке.

Все итераторы `HashMap` посещают записи отображения в произвольном порядке. Итераторы `BTreeMap` посещают записи в порядке ключей.

Типы HashSet<T> и BTreeSet<T>

Множества – это коллекции значений, организованные так, что проверка принадлежности производится быстро.

```
let b1 = large_vector.contains("needle");    // медленно, проверяется каждый элемент
let b2 = large_hash_set.contains("needle");  // быстро, поиск в хеше
```

Во множестве никогда не бывает дубликатов значений.

У отображений и множеств разные методы, но под капотом множество – это, по существу, отображение, содержащее только ключи, а не пары ключ-значение. На

самом деле два типа множеств, `HashSet<T>` и `BTreeSet<T>`, реализованы как тонкие обертки вокруг типов `HashMap<T, ()>` и `BTreeMap<T, ()>`:

- методы **`HashSet::new()`** и **`BTreeSet::new()`** создают новые множества;
- метод **`iter.collect()`** можно использовать для создания нового множества из итератора. Если `iter` порождает некоторые значения несколько раз, то дубликаты отбрасываются.

Основные методы у `HashSet<T>` и `BTreeSet<T>` одинаковы:

- **`set.len()`** возвращает количество значений во множестве `set`;
- **`set.is_empty()`** возвращает `true`, если множество не содержит элементов;
- **`set.contains(&value)`** возвращает `true`, если значение `value` принадлежит множеству;
- **`set.insert(value)`** добавляет `value` во множество. Возвращает `true`, если значение было добавлено, и `false`, если оно и раньше принадлежало множеству;
- **`set.remove(&value)`** удаляет `value` из множества. Возвращает `true`, если значение было удалено, и `false`, если оно не принадлежало множеству.

Как и в случае отображений, методы, которые ищут значение по ссылке, принимают также типы, которые можно заимствовать от `T`. Подробнее см. раздел «Характеристики Borrow и BorrowMut» главы 13.

Обход множества

Обойти множество можно двумя способами:

- итерирование по значению (`for v in set`) порождает элементы множества (и потребляет само множество);
- итерирование по разделяемой ссылке (`for v in &set`) порождает разделяемые ссылки на элементы множества.

Итерирование множества по изменяемой ссылке не поддерживается. Не существует никакого способа получить изменяемую ссылку на значение, хранящееся во множестве.

- **`set.iter()`** возвращает итератор для обхода множества `set` по ссылке.

Итераторы для `HashSet`, как и итераторы для `HashMap`, порождают значения в произвольном порядке. Итераторы для `BTreeSet` порождают значения по порядку, как в случае отсортированного вектора.

Когда равные значения различны

У множеств есть несколько странных методов, которые нужны только в том случае, когда важны различия между «равными» значениями.

Зачастую такие различия действительно существуют. Например, символы двух идентичных строк находятся в разных областях памяти.

```
let s1 = "hello".to_string();
let s2 = "hello".to_string();

println!("{:p}", &s1 as &str);  // 0x7f8b32060008
println!("{:p}", &s2 as &str);  // 0x7f8b32060010
```

Обычно нам это не важно.

Но если все-таки важно, то следующие методы дают возможность получить доступ к настоящим значениям, хранящимся во множестве. Каждый из них возвра-

щает значение типа `Option`, равное `None`, если множество `set` не содержит искомого значения:

- **`set.get(&value)`** возвращает разделяемую ссылку на элемент `set`, равный `value`, если таковой существует. Возвращаемое значение имеет тип `Option<&T>`;
- **`set.take(&value)`** аналогичен `set.remove(&value)`, но возвращает удаленное значение, если таковое существует. Возвращаемое значение имеет тип `Option<T>`;
- **`set.replace(value)`** аналогичен `set.insert(value)`, но если `set` уже содержит значение, равное `value`, то заменяет его и возвращает старое значение. Возвращаемое значение имеет тип `Option<T>`.

Операции над множествами как единым целым

До сих пор мы рассматривали методы, применяемые к одному значению в одном множестве. Но существуют также методы, оперирующие целыми множествами.

- **`set1.intersection(&set2)`** возвращает итератор для обхода значений, принадлежащих обоим множествам `set1` и `set2`.
  Например, чтобы напечатать имена студентов, посещающих курсы нейрохирургии и ракетостроения, можно написать такой код:

```
for student in brain_class {
    if rocket_class.contains(&student) {
        println!("{}", student);
    }
}
```

  Или более кратко:

```
for student in brain_class.intersection(&rocket_class) {
    println!("{}", student);
}
```

  Для этой цели даже есть специальный оператор.
  `&set1 & &set2` возвращает новое множество, являющееся пересечением `set1` и `set2`. Следующий код находит значения, принадлежащие `set1` *И* `set2`:

```
let overachievers = &brain_class & &rocket_class;
```

- **`set1.union(&set2)`** возвращает итератор для обхода значений, принадлежащих хотя бы одному из множеств `set1` и `set2`;
- **`&set1 | &set2`** возвращает новое множество, содержащее все такие значения;
- **`set1.difference(&set2)`** возвращает итератор для обхода значений, принадлежащих `set1`, но не принадлежащих `set2`;
- **`&set1 - &set2`** возвращает новое множество, содержащее все такие значения;
- **`set1.symmetric_difference(&set2)`** возвращает итератор для обхода значений, принадлежащих либо множеству `set1`, либо множеству `set2`, но не обоим;
- **`&set1 ^ &set2`** возвращает новое множество, содержащее все такие значения.

И еще существуют три метода для проверки соотношения между множествами.

- **`set1.is_disjoint(set2)`** возвращает `true`, если множества `set1` и `set2` не пересекаются, т. е. не имеют общих значений;
- **`set1.is_subset(set2)`** возвращает `true`, если `set1` – подмножество `set2`, т. е. любой элемент `set1` является также элементом `set2`;

- `set1.is_superset(set2)` противоположен, он возвращает `true`, если `set2` – подмножество `set1`.

Для множеств поддерживается также проверка на равенство с помощью операторов `==` и `!=`; два множества считаются равными, если они содержат в точности одни и те же значения.

Хеширование

`std::hash::Hash` – стандартная библиотечная характеристика для хешируемых типов. Ключи `HashMap` и элемент `HashSet` реализуют характеристики `Hash` и `Eq`.

Большинство встроенных типов, реализующих `Eq`, реализует также `Hash`. Все целые типы, `char` и `String` являются хешируемыми. Хешируемыми являются также кортежи, массивы, срезки и векторы – при условии, что их элементы хешируемые.

Один из принципов стандартной библиотеки состоит в том, что хеш-код значения не должен зависеть от того, где оно хранится и как на него указать. Следовательно, хеш-код ссылки такой же, как у значения, на которое оно ссылается, а у бокса `Box` – такой же, как у заключенного в нем значения. Хеш-коды вектора и срезки, содержащей все его данные (`&vec[..]`), равны. Строка `String` имеет такой же хеш-код, как ссылка `&str` с такими же символами.

Структуры и перечисления по умолчанию не реализуют характеристику `Hash`, но реализацию можно вывести:

```
/// Инвентарный номер предмета из коллекции Британского музея.
#[derive(Clone, PartialEq, Eq, Hash)]
enum MuseumNumber {
    ...
}
```

Это работает, коль скоро все поля типа хешируемые.

Если для некоторого типа характеристика `PartialEq` реализована вручную, то вручную же следует реализовать характеристику `Hash`. Рассмотрим, к примеру, тип, представляющий бесценные музейные сокровища:

```
struct Artifact {
    id: MuseumNumber,
    name: String,
    cultures: Vec<Culture>,
    date: RoughTime,
    ...
}
```

Два артефакта считаются равными, если равны их инвентарные номера:

```
impl PartialEq for Artifact {
    fn eq(&self, other: &Artifact) -> bool {
        self.id == other.id
    }
}

impl Eq for Artifact {}
```

Поскольку мы сравниваем артефакты только по инвентарным номерам, то и хешировать их должны точно так же:

```
impl Hash for Artifact {
    fn hash<H: Hasher>(&self, hasher: &mut H) {
        // Делегировать хеширование типу MuseumNumber.
        self.id.hash(hasher);
    }
}
```

(В противном случае тип `HashSet<Artefact>` работал бы неправильно; в любой хеш-таблице требуется, чтобы `hash(a) == hash(b)`, если `a == b`.)

Это позволяет создать множество `HashSet` значений типа `Artefact`:

```
let mut collection = HashSet::<Artifact>::new();
```

Как видно из приведенного выше кода, даже реализуя `Hash` вручную, мы не обязаны ничего знать об алгоритмах хеширования. Метод `.hash()` принимает ссылку на значение типа `Hasher`, представляющее алгоритм хеширования. Мы просто подаем на вход `Hasher` все данные, принимающие участие в вычислении оператора `==`. `Hasher` вычисляет хеш-код всего, что ему передается.

Применение пользовательского алгоритма хеширования

Метод `hash` универсальный, поэтому показанные выше реализации характеристики `Hash` могут подавать данные на вход любому типу, реализующему `Hasher`. Именно так Rust поддерживает сменные алгоритмы хеширования. `Hash` и `Hasher` – парные характеристики.

Третья характеристика, `std::hash::BuildHasher`, предназначена для типов, представляющих начальное состояние алгоритма хеширования. Каждое значение `Hasher` одноразовое, как и итератор: мы его один раз используем, а потом выбрасываем. Значение `BuildHasher` многоразовое.

Любое отображение типа `HashMap` содержит значение `BuildHasher`, которым пользуется всякий раз, когда нужно вычислить хеш-код. Это значение содержит ключ, начальное состояние и другие параметры, нужные алгоритму хеширования при каждом выполнении.

Полный протокол вычисления хеш-кода выглядит так:

```
use std::hash::{Hash, Hasher, BuildHasher};

fn compute_hash<B, T>(builder: &B, value: &T) -> u64
    where B: BuildHasher, T: Hash
{
    let mut hasher = builder.build_hasher();  // 1. Запустить алгоритм
    value.hash(&mut hasher);                  // 2. Подать на вход данные
    hasher.finish()                           // 3. Конец, порождается u64
}
```

`HashMap` вызывает эти три метода каждый раз, как нужно вычислить хеш-код. Все методы допускают встраивание и потому работают очень быстро.

По умолчанию в Rust используется хорошо известный алгоритм хеширования SipHash-1-3. Он быстрый и минимизирует конфликты хеширования. На самом деле это криптографический алгоритм, т. к. не известен эффективный способ сгенерировать для него конфликт. При условии что для каждой хеш-таблицы берется новый непредсказуемый ключ, Rust устойчив к DOS-атаке, известной под назва-

нием HashDoS, когда противник сознательно вызывает конфликты хеширования, чтобы спровоцировать наихудшую производительность сервера.

Но возможно, что в вашем приложении это не нужно. Если хранится много небольших ключей, например целых чисел или очень коротких строк, то можно реализовать более быструю функцию хеширования ценой уязвимости к HashDoS-атакам. В крейте `fnv` с этой целью реализован алгоритм Фаулера-Нолла-Во. Чтобы опробовать его, добавьте такую строку в свой файл `Cargo.toml`:

```
[dependencies]
fnv = "1.0"
```

Затем импортируйте типы отображения и множества из крейта `fnv`:

```
extern crate fnv;

use fnv::{FnvHashMap, FnvHashSet};
```

Эти два типа можно подставить вместо `HashMap` и `HashSet`. Заглянув в исходный код `fnv`, мы найдем их определения:

```
/// `HashMap` с использованием алгоритма хеширования FNV по умолчанию.
pub type FnvHashMap<K, V> = HashMap<K, V, FnvBuildHasher>;

/// `HashSet` с использованием алгоритма хеширования FNV по умолчанию.
pub type FnvHashSet<T> = HashSet<T, FnvBuildHasher>;
```

Стандартные коллекции `HashMap` и `HashSet` принимают необязательный параметрический тип, определяющий алгоритм хеширования; `FnvHashMap` и `FnvHashSet` – псевдонимы универсальных типов `HashMap` и `HashSet`, для которых в качестве этого типа задан алгоритм хеширования FNV.

За пределами стандартных коллекций

Пользовательский тип коллекций в Rust создается так же, как в любом другом языке. Мы организуем данные, комбинируя различные средства языка: структуры и перечисления, стандартные коллекции, типы `Option`, `Box` и т. д. В качестве примера можно взять тип `BinaryTree<T>`, определенный в разделе «Универсальные перечисления» главы 10.

Если вы поднаторели в реализации структур данных на C++ с использованием простых указателей, ручного управления памятью, оператора `new` с размещением и явных вызовов конструкторов с целью достижения оптимальной производительности, то наверняка ограничения безопасности Rust будут вас стеснять. Все вышеперечисленные механизмы принципиально небезопасны. Они есть в Rust, но только для тех, кто явно выбирает небезопасный код. В главе 21 показано, как это сделать, там приведен пример, в котором небезопасный код применяется для реализации безопасной пользовательской коллекции.

А пока погреемся в теплых лучах стандартных коллекций и их безопасного эффективного API. Как и большая часть стандартной библиотеки, они спроектированы таким образом, чтобы необходимость писать слово «`unsafe`» возникала как можно реже.

Глава 17

Строки и текст

> Строка – несгибаемая структура данных, и всюду, куда она передается, сразу возникает дублирование. Это идеальное средство для сокрытия информации.
>
> *— Алан Перлис, эпиграмма 34*

В этой книге основные текстовые типы Rust – `String`, `str` и `char` – используются повсеместно. В разделе «Строковые типы» главы 3 был описан синтаксис символьных и строковых литералов и показано, как строки представляются в памяти. В этой главе обработка текста рассматривается более подробно.

Обсуждаются следующие вопросы:

- базовые сведения о Юникоде, которые помогут понять принципы проектирования стандартной библиотеки;
- тип `char`, представляющий одну кодовую позицию Юникода;
- типы `String` и `str`, представляющие принадлежащие владельцу и заимствованные последовательности символов Юникода. Существует целый ряд методов для создания, модификации, поиска и обхода содержимого этих типов;
- средства форматирования строк, в т. ч. макросы `println!` и `format!`. Можно написать собственные макросы форматирования и обобщить их для поддержки собственных типов;
- обзор поддержки регулярных выражений в Rust;
- почему важна нормализация в Юникоде и как этот вопрос решается в Rust.

Общие сведения о Юникоде

Эта книга посвящена языку Rust, а не Юникоду – теме, на которую написаны целые тома. Но символьные и строковые типы в Rust спроектированы с учетом Юникода, поэтому мы приведем базовые сведения о Юникоде, которые позволят лучше понять Rust.

ASCII, Latin-1 и Юникод

Стандарты кодирования Юникод и ASCII совпадают на всех кодовых позициях ASCII, от `0` до `0x7f`. Например, в обоих случаях символу `'*'` соответствует кодовая

позиция 42. Аналогично в Юникоде кодовые позиции от 0 до 0xff назначены тем же символам, что в кодировке ISO/IEC 8859-1, – восьмиразрядному надмножеству ASCII, применяемому в западноевропейских языках. В Юникоде этот диапазон кодовых позиций называется «кодовым блоком Latin-1», ну а мы будем называть его просто «Latin-1».

Поскольку Юникод – надмножество Latin-1, то для преобразования Latin-1 в Юникод даже таблица соответствия не нужна:

```
fn latin1_to_char(latin1: u8) -> char {
    latin1 as char
}
```

Обратное преобразование также тривиально – при условии, что кодовая позиция принадлежит диапазону Latin-1:

```
fn char_to_latin1(c: char) -> Option<u8> {
    if c as u32 <= 0xff {
        Some(c as u8)
    } else {
        None
    }
}
```

UTF-8

Типы String и str представляют текст в кодировке UTF-8, где каждому символу соответствует от одного до четырех байтов:

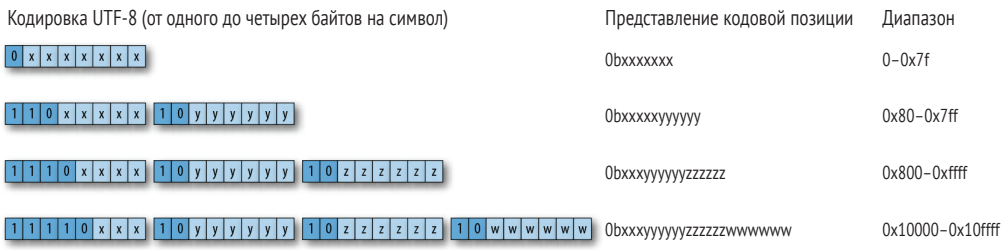


Рис. 17.1 ❖ Кодировка UTF-8

Чтобы последовательность чисел была корректной в смысле UTF-8, должно соблюдаться два ограничения. Во-первых, корректной кодировкой любой кодовой позиции считается самая короткая из возможных. Это правило гарантирует единственность кодировки UTF-8 для каждой кодовой позиции. Во-вторых, коды UTF-8 не назначаются позициям от 0xd800 до 0xdfff и большим 0x10ffff: эти позиции либо зарезервированы под цели, не относящиеся к символам, либо вообще выходят за границы диапазона Юникода.

Приведем несколько примеров:

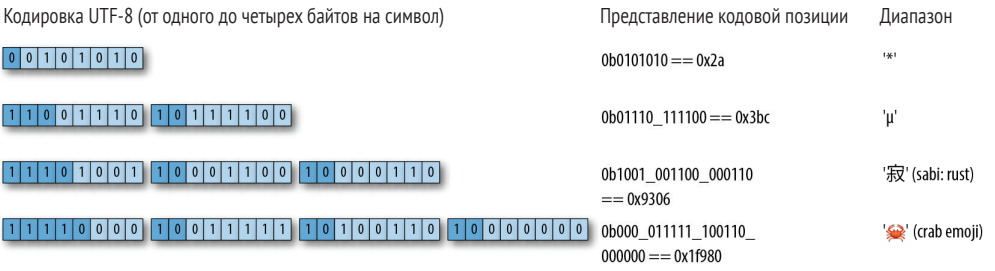


Рис. 17.2 ❖ Кодировка UTF-8

Отметим, что хотя в кодировке эмотикона «краб» начальный байт вносит в кодовую позицию только нули, для него все равно необходим четырехбайтовый код, т. к. тремя байтами в UTF-8 можно представить только 16-разрядные кодовые позиции, а `0x1f980` содержит 17 разрядов.

Вот пример строки, содержащей символы с кодами разной длины:

```
assert_eq!("うどん: udon".as_bytes(),
           &[0xe3, 0x81, 0x86, // う
             0xe3, 0x81, 0xa9, // ど
             0xe3, 0x82, 0x93, // ん
             0x3a, 0x20, 0x75, 0x64, 0x6f, 0x6e // : udon
]);
```

На рисунке показаны некоторые весьма полезные свойства UTF-8:

- поскольку в UTF-8 кодовые позиции от `0` до `0x7f` представлены байтами от `0` до `0x7f`, последовательности байтов, соответствующие ASCII-тексту, являются корректными в смысле UTF-8. А если строка в кодировке UTF-8 включает только символы из диапазона ASCII, то верно и обратное: строка в кодировке UTF-8 корректна в смысле ASCII.
  Для Latin-1 это уже неверно: например, Latin-1 кодирует букву `'é'` байтом `0xe9`, тогда как в UTF-8 он интерпретируется как первый байт трехбайтового кода;
- глядя на старшие биты байта, мы сразу можем сказать, является ли он началом кода некоторого символа в UTF-8 или внутренним байтом допустимой последовательности;
- одного лишь первого байта, точнее его начальных битов, достаточно, чтобы узнать всю длину кода;
- поскольку кодов длиннее четырех байтов не существует, то при работе с текстом в кодировке UTF-8 не может возникнуть неограниченных циклов, что очень важно, когда данные поступают из ненадежного источника;
- для корректной последовательности байтов в кодировке UTF-8 всегда можно однозначно сказать, где начинаются и заканчиваются коды символов, даже если начать с произвольно выбранной точки в середине последовательности. Первый байт кода UTF-8 всегда отличается от промежуточных, поэтому никакой код не может начинаться в середине другого. Первый байт определяет длину кода, поэтому никакой код не может быть префиксом другого. Отсюда вытекает ряд важных следствий. Например, для

поиска в строке UTF-8 ASCII-разделителя достаточно простого сравнения с байтом-разделителем. Он никогда не окажется частью многобайтовой последовательности, поэтому следить за структурой UTF-8 не нужно. Аналогично алгоритмы поиска однобайтовой строки в другой строке применимы к строкам UTF-8 без каких-либо изменений, хотя они даже не проверяют каждый байт просматриваемого текста.

Хотя кодировки переменной ширины сложнее, чем кодировки постоянной ширины, благодаря описанным выше свойствам работать с UTF-8 удобнее, чем можно было бы ожидать. Большинство проблем берет на себя стандартная библиотека.

Направление текста

В таких системах письменности, как латиница, кириллица или тайская, пишут слева направо, но в иврите и в арабском языке – справа налево. В Юникоде символы хранятся в том порядке, в котором обычно записываются или читаются, так что начальные байты строки, содержащей, к примеру, текст на иврите, кодируют символ, который должен был бы записываться справа:

```
assert_eq!("ערב טוב".chars().next(), Some('ע'));
```

В именах нескольких методов из стандартной библиотеки встречаются слова `left` и `right`, означающие начало и конец текста. При описании таких функций мы будет точно указывать, что они делают.

Символы (char)

В Rust `char` – это 32-разрядное значение, содержащее кодовую позицию Юникода. Гарантируется, что `char` находится в диапазоне от `0` до `0xd7ff` или от `0xe000` до `0x10ffff`; все методы создания и изменения значений типа `char` соблюдают это условие. Тип `char` реализует характеристики `Copy` и `Clone`, а также все обычные характеристики для сравнения, хеширования и форматирования.

В описаниях ниже переменная `ch` всегда имеет тип `char`.

Классификация символов

В типе `char` имеются методы для отнесения символов к нескольким стандартным категориям. Определения категорий взяты из стандарта Юникода.

Метод	Описание	Примеры

`ch.is_numeric()`	Числовой символ. Сюда входят такие общие категории Юникода, как «Число; цифра», «Число; буква» и «Число; прочее»	`'4'.is_numeric()`
		`'৸'.is_numeric()`
		`!'⑧'.is_numeric()`

`ch.is_alphabetic()`	Буквенный символ: производное свойство Юникода «Буквенный»	`'q'.is_alphabetic()` `'七'.is_alphabetic()`

`ch.is_alphanumeric()`	Числовой или буквенный (см. определения выше)	`'9'.is_alphanumeric()`
		`'饂'.is_alphanumeric()`
		`!'*'.is_alphanumeric()`

`ch.is_whitespace()`	Пробельный символ: свойство Юникода «`WSpace=Y`»	`' '.is_whitespace()`
		`'\n'.is_whitespace()`
		`'\u{A0}'.is_whitespace()`

`ch.is_control()`	Управляющий символ: общая категория Юникода «Прочие, управляющие»	`'\n'.is_control()` `'\u{85}'.is_control()`

Работа с цифрами

Для обработки цифр предназначены следующие методы:

- **`ch.to_digit(radix)`** определяет, является ли `ch` цифрой в системе счисления с основанием `radix`. Если да, то возвращается `Some(num)`, где `num` – число типа `u32`. В противном случае возвращается `None`. Метод распознает только цифры в кодировке ASCII, а не более широкий класс символов `char::is_numeric`. Параметр `radix` может быть числом от 2 до 36. Если основание больше 10, то цифрами считаются ASCII-буквы с кодами от 10 до 35 в любом регистре;
- свободная функция **`std::char::from_digit(num, radix)`** преобразует цифру `num` типа `u32` в символ `char`, если это возможно. Если `num` можно представить как цифру в системе с основанием `radix`, то `from_digit` возвращает `Some(ch)`, где `ch` – цифра. Если `radix` больше 10, то `ch` может быть буквой в нижнем регистре. В противном случае возвращается `None`.
  Эта функция обратна методу `to_digit`. Если `std::char::from_digit(num), radix)` равно `Some(ch)`, то `ch.to_digit(radix)` равно `Some(num)`. Если `ch` – ASCII-цифра или буква в нижнем регистре, то обратимость также имеет место;
- **`ch.is_digit(radix)`** возвращает `true`, если `ch` – ASCII-цифра в системе с основанием `radix`. Это эквивалентно условию `ch.to_digit(radix) != None`.

Например:

```
assert_eq!('F'.to_digit(16), Some(15));
assert_eq!(std::char::from_digit(15, 16), Some('f'));
assert!(char::is_digit('f', 16));
```

Преобразование регистра символов

Для преобразования регистра символов используются:

- методы **`ch.is_lowercase()`** и **`ch.is_uppercase()`** определяют, является ли символ `ch` строчной или заглавной буквой. Они согласованы с производными свойствами Юникода «Нижний регистр» и «Верхний регистр», поэтому работают не только для латиницы, но и, например, для кириллицы и греческого алфавита. Для ASCII также получаются ожидаемые результаты;
- методы **`ch.to_lowercase()`** и **`ch.to_uppercase()`** возвращают итераторы, порождающие строчные и заглавные эквиваленты `ch` согласно алгоритмам преобразования регистра по умолчанию, принятым в Юникоде:

```
let mut upper = 's'.to_uppercase();
assert_eq!(upper.next(), Some('S'));
assert_eq!(upper.next(), None);
```

Возвращается итератор, потому что преобразование регистра в Юникоде не всегда дает один символ:

```
// Заглавной формой немецкой буквы "эс цет" является "SS":
let mut upper = 'ß'.to_uppercase();
assert_eq!(upper.next(), Some('S'));
assert_eq!(upper.next(), Some('S'));
assert_eq!(upper.next(), None);

// Согласно Юникоду, строчным вариантом турецкой заглавной буквы 'İ'
// является буква 'i', за которой следует символ `'\u{307}'`,
```

```
// МОДИФИЦИРУЮЩАЯ ВЕРХНЯЯ ТОЧКА, так что последующее обратное
// преобразование в верхний регистр сохраняет точку.
let ch = 'İ'; // `'\u{130}'`
let mut lower = ch.to_lowercase();
assert_eq!(lower.next(), Some('i'));
assert_eq!(lower.next(), Some('\u{307}'));
assert_eq!(lower.next(), None);
```

Для удобства эти итераторы реализуют характеристику `std::fmt::Display`, чтобы их можно было непосредственно передавать макросу `println!` или `write!`.

Преобразование в целое число и обратно

Оператор Rust `as` преобразует `char` в любой целый тип, молчаливо отбрасывая старшие биты:

```
assert_eq!('B' as u32, 66);
assert_eq!('鱦' as u8, 66);   // старшие биты отброшены
assert_eq!('二' as i8, -116); // то же самое
```

Оператор `as` преобразует любое число типа `u8` в `char`, а тип `char` реализует характеристику `From<u8>`. Но более широкие типы могут представлять недопустимые кодовые позиции, поэтому для них нужно использовать метод `std::char::from_u32`, возвращающий `Option<char>`:

```
assert_eq!(char::from(66), 'B');
assert_eq!(std::char::from_u32(0x9942), Some('鱦'));
assert_eq!(std::char::from_u32(0xd800), None); // зарезервировано для UTF-16
```

Типы String и str

Гарантируется, что типы `String` и `str` содержат корректные в смысле UTF-8 последовательности байтов. Для этого библиотека ограничивает способы создания значений типа `String` и `str` и операции над ними, так чтобы значение было корректным при появлении на свет и оставалось таковым в процессе различных манипуляций. Все методы этих типов дают такую гарантию, никакая безопасная операция над ними не может привести к некорректной последовательности байтов. Это упрощает код программ для работы с текстом.

Методы обработки текстов помещены в тип `str` или `String` в зависимости от того, нужен ли методу буфер переменного размера или ему достаточно операций на месте. Поскольку результатом разыменования `String` является `&str`, любой метод, определенный в типе `str`, применим также и к `String`. В этом разделе описываются методы обоих типов, приблизительно сгруппированные по функциональности.

Эти методы индексируют текст по смещению байта и измеряют длину текста в байтах, а не в символах. Принимая во внимание природу Юникода, индексирование по символам на практике не так полезно, как могло бы показаться, а с байтовыми смещениями работать проще и быстрее. При попытке задать смещение байта, так что он окажется в середине какого-то кода UTF-8, метод паникует, так что получить некорректную последовательность байтов UTF-8 таким образом не получится.

Тип `String` реализован как обертка вокруг `Vec<u8>`, гарантирующая корректность последовательности байтов в смысле UTF-8. В Rust никогда не будет использоваться более сложное представление строк, поэтому можно смело предполагать, что с точки зрения производительности `String` и `Vec` одинаковы.

Далее предполагаются следующие типы переменных.

Переменная	Предполагаемый тип
`string`	`String`
`slice`	`&str` или нечто, что разыменовывается в `&str`, например `String` или `Rc<String>`
`ch`	`char`
`n`	`usize`, длина
`i`, `j`	`usize`, смещение байта
`range`	диапазон байтовых смещений типа `usize`, ограниченный с обеих сторон (`i..j`) или частично – `i..`, `..j` или `..`
`pattern`	образец любого типа: `char`, `String`, `&str`, `&[char]` или `FnMut(char) -> bool`

Типы образцов описываются в разделе «Образцы для поиска текста» ниже в этой главе.

Создание значений типа String

Есть несколько стандартных способов создать строку:

- **`String::new()`** возвращает новую пустую строку. У нее еще нет выделенного в куче буфера, но он будет выделен, когда потребуется;
- **`String::with_capacity(n)`** возвращает новую пустую строку с заранее выделенным буфером, достаточным для размещения как минимум `n` байтов. Если длина создаваемой строки известна заранее, то этот конструктор позволяет задать размер буфера с самого начала, вместо того чтобы перестраивать его по мере построения строки. В случае превышения этого размера буфер будет расти, как обычно. Как и у вектора, у строки есть методы `capacity`, `reserve` и `shrink_to_fit`, но обычно подразумеваемой по умолчанию логики выделения памяти достаточно;
- **`slice.to_string()`** выделяет памяти для нового значения `String`, в которое копируется содержимое срезки `slice`. В этой книге нам неоднократно встречались выражения вида `"literal text".to_string()` для создания строк из строковых литералов;
- **`iter.collect()`** конструирует строку посредством конкатенации объектов итератора, которые могут быть значениями типа `char`, `&str` или `String`. Например, чтобы удалить все пробелы из строки, можно написать:

```
let spacey = "man hat tan";
let spaceless: String =
    spacey.chars().filter(|c| !c.is_whitespace()).collect();
assert_eq!(spaceless, "manhattan");
```

  Такое использование метода `collect` позволяет извлечь выгоду из того, что тип `String` реализует характеристику `std::iter::FromIterator`;
- тип `&str` не может реализовать характеристику `Clone`: для этого необходимо, чтобы метод `clone` в применении к `&T` возвращал значение типа `T`, но тип

str безразмерный. Однако &str реализует характеристику ToOwned, что позволяет автору реализации задать собственный принадлежащий владельцу эквивалент, поэтому **slice.to_owned()** возвращает копию slice в виде новой строки String с выделенным буфером.

Простая инспекция

Следующие методы возвращают основную информацию о срезке строки:

- **slice.len()** возвращает длину slice в байтах;
- **slice.is_empty()** возвращает true, если slice.len() == 0;
- **slice[range]** возвращает срезку, заимствующую заданную часть slice. Поддерживаются диапазоны, ограниченные с одной стороны и не ограниченные вовсе. Например:

```
let full = "bookkeeping";
assert_eq!(&full[..4], "book");
assert_eq!(&full[5..], "eeping");
assert_eq!(&full[2..4], "ok");
assert_eq!(full[..].len(), 11);
assert_eq!(full[5..].contains("boo"), false);
```

- запрещается индексировать срезку строки из одного символа, например slice[i]. Извлечь один символ с заданным байтовым смещением можно, но неудобно: придется создать итератор по символам срезки и запросить у него один символ UTF-8:

```
let parenthesized = "Rust (鱈)";
assert_eq!(parenthesized[6..].chars().next(), Some('鱈'));
```

  Однако необходимость в этом возникает довольно редко. В Rust есть куда более элегантные способы обхода срезок, которые мы опишем в разделе «Обход текста» ниже;
- **slice.split_at(i)** возвращает кортеж из двух разделяемых срезок, заимствованных у slice: участок до байта со смещением i включительно и участок после него. Иными словами, возвращается (slice[..i], slice[i..]);
- **slice.is_char_boundary(i)** возвращает true, если байт со смещением i оказывается на границе между символами и потому может служить смещением от начала срезки slice.

Естественно, срезки можно сравнивать на равенство и на больше-меньше, а также хешировать. При сравнении на больше-меньше строка рассматривается как последовательность кодовых позиций Юникода, которые сравниваются лексикографически.

Дописывание и вставка текста

Следующие методы добавляют текст в строку String:

- **string.push(ch)** дописывает символ ch в конец string;
- **string.push_str(slice)** дописывают в конец все содержимое срезки slice;
- **string.extend(iter)** дописывает в конец объекты, порождаемые итератором iter. Итератор может порождать значения типа char, str или String. Все это реализации характеристики std::iter::Extend для типа String.

```
let mut also_spaceless = "con".to_string();
also_spaceless.extend("tri but ion".split_whitespace());
assert_eq!(also_spaceless, "contribution");
```

- **`string.insert(i, ch)`** вставляет один символ `ch` в позицию `string` с байтовым смещением `i`. Все символы, начиная со `i`-го, сдвигаются, чтобы освободить место, поэтому построение строки таким образом может требовать времени, квадратично зависящего от длины строки;
- **`string.insert_str(i, slice)`** делает то же самое для срезки `slice` с аналогичными последствиями для производительности.

Тип `String` реализует характеристику `std::fmt::Write`, т. е. для добавления отформатированного текста в конец строки можно использовать макросы `write!` и `writeln!`:

```
use std::fmt::Write;

let mut letter = String::new();
writeln!(letter, "Whose {} these are I think I know", "rutabagas")?;
writeln!(letter, "His house is in the village though;")?;
assert_eq!(letter, "Whose rutabagas these are I think I know\n\
                    His house is in the village though;\n");
```

Поскольку макросы `write!` и `writeln!` предназначены для записи в поток вывода, они возвращают значение типа `Result`, при игнорировании которого компилятор ругается. В этом коде мы используем оператор `?`, чтобы избежать ошибок компиляции, но поскольку запись в строку не может завершиться ошибкой, то подошел бы и вызов `.unwrap()`.

Поскольку тип `String` реализует характеристики `Add<&str>` и `AddAssign<&str>`, можно писать код вида:

```
let left = "partners".to_string();
let mut right = "crime".to_string();
assert_eq!(left + " in " + &right, "partners in crime");

right += " doesn't pay";
assert_eq!(right, "crime doesn't pay");
```

В применении к строкам оператор `+` принимает левый операнд по значению, поэтому может повторно использовать ту же строку как результат сложения. Следовательно, если буфер левого операнда достаточно велик, то никакого дополнительного выделения памяти не производится.

Из-за печального отсутствия симметрии левый операнд `+` не может иметь тип `&str`, так что следующий код недопустим:

```
let parenthetical = "(" + string + ")";
```

Вместо этого нужно написать так:

```
let parenthetical = "(".to_string() + string + ")";
```

Однако это ограничение мешает строить строки с конца. Такой способ работал бы медленно, потому что текст пришлось бы все время сдвигать в конец буфера.

Впрочем, построение строк от начала к концу путем добавления небольших фрагментов работает эффективно. Строка ведет себя как вектор, т. е. в случае от-

сутствия места в буфере выделяется новый удвоенного размера. В разделе «Поэлементное построение векторов» главы 3 отмечалось, что в результате накладные расходы на копирование оказываются пропорциональны окончательному размеру. Но применение метода `String::with_capacity` для создания строки с буфером изначально правильного размера позволяет вовсе избежать изменения размера, а значит, и обращений к распределителю памяти.

Удаление текста

В типе `String` есть несколько методов для удаления текста (они не влияют на емкость строки; если хотите освободить память, пользуйтесь методом `shrink_to_fit`):

- **`string.clear()`** опустошает строку `string`;
- **`string.truncate(n)`** отбрасывает все символы, начиная с байтового смещения `n`, так что длина результирующей строки оказывается не больше `n`. Если строка `string` короче `n` байтов, метод ничего не делает;
- **`string.pop()`** удаляет последний символ строки, если она не пуста, и возвращает его в виде значения типа `Option<char>`;
- **`string.remove(i)`** удаляет из строки символ в позиции с байтовым смещением `i` и возвращает его. Все последующие символы сдвигаются к началу. Время работы линейно зависит от числа последующих символов;
- **`string.drain(range)`** возвращает итератор по заданному диапазону индексов байтов и удаляет этим символы после уничтожения итератора. Символы, следующие за диапазоном, сдвигаются к началу строки.

```
let mut choco = "chocolate".to_string();
assert_eq!(choco.drain(3..6).collect::<String>(), "col");
assert_eq!(choco, "choate");
```

Если требуется просто удалить диапазон символов, то можно сразу уничтожить итератор, ничего не читая из него:

```
let mut winston = "Churchill".to_string();
winston.drain(2..6);
assert_eq!(winston, "Chill");
```

Соглашения о поиске и итерировании

Стандартные библиотечные функции для поиска текста и его обхода следуют нескольким соглашениям об именовании, чтобы их было легче запомнить:

- большинство операций обрабатывает текст от начала к концу, но операции с именами, начинающимися буквой **`r`**, выполняются от конца к началу. Например, `rsplit` – вариант `split`, работающий от конца к началу. В некоторых случаях изменение направления может повлиять не только на порядок порождения значений, но и на сами значения. Пример показан на рис. 17.3;
- итераторы с именами, оканчивающимися буквой **`n`**, порождают не более заданного числа совпадений;
- итераторы с именами, оканчивающимися суффиксом **`_indices`**, порождают не только обычные объекты, но и их байтовые смещения от начала срезки.

Стандартная библиотека не предоставляет всех комбинаций для каждой операции. Например, для многих операций не нужен вариант **`n`**, поскольку можно просто завершить обход досрочно.

Образцы для поиска текста

Когда стандартной библиотечной функции необходимо произвести поиск в тексте, разбить или усечь его, она позволяет задать искомое в виде значений различных типов:

```
let haystack = "One fine day, in the middle of the night";

assert_eq!(haystack.find(','), Some(12));
assert_eq!(haystack.find("night"), Some(35));
assert_eq!(haystack.find(char::is_whitespace), Some(3));
```

Эти типы называются *образцами* и поддерживаются большинством операций:

```
assert_eq!("## Elephants"
           .trim_left_matches(|ch: char| ch == '#' || ch.is_whitespace()),
           "Elephants");
```

Стандартная библиотека поддерживает образцы четырех основных видов:

- `char` в качестве образца сопоставляется с одним символом;
- `String`, `&str` или `&&str` в качестве образца сопоставляется с подстрокой, равной образцу;
- замыкание `FnMut(char) -> bool` в качестве образца сопоставляется с одним символом, для которого замыкание возвращает `true`;
- `&[char]` в качестве образца (не `&str`, а срезка значений типа `char`) сопоставляется с одним любым символом из этого списка. Отметим, что если список записан в виде литерального массива, то для получения нужного типа может потребоваться выражение `as`:

  ```
  let code = "\t    function noodle() { ";
  assert_eq!(code.trim_left_matches(&[' ', '\t'] as &[char]),
             "function noodle() { ");
  // Более короткий эквивалент: &[' ', '\t'][..]
  ```

  В противном случае Rust примет это за массив фиксированного размера `&[char; 2]`, который, к сожалению, не может быть типом образца.

В коде самой библиотеки образцом может быть любой тип, реализующий характеристику `std::str::Pattern`. Детали `Pattern` еще не устаканились, поэтому ее нельзя использовать в собственных типах, но открыта дверь для использования регулярных выражений и других нетривиальных образцов в будущем. Rust не гарантирует, что типы образцов, поддерживаемые в настоящее время, будут работать и в будущем.

Поиск и замена

В Rust есть несколько методов для поиска образцов в срезках и, возможно, замены их другим текстом:

- **`slice.contains(pattern)`** возвращает `true`, если срезка `slice` содержит совпадение с образцом `pattern`;
- **`slice.starts_with(pattern)`** и **`slice.ends_with(pattern)`** возвращает `true`, если соответственно начало или конец `slice` совпадает с `pattern`:

  ```
  assert!("2017".starts_with(char::is_numeric));
  ```

- **slice.find(pattern)** и **slice.rfind(pattern)** возвращает `Some(i)`, если `slice` содержит совпадение с `pattern`, при этом `i` – смещение байта, в котором начинается это совпадение. Метод `find` возвращает первое совпадение, метод `rfind` – последнее:

```
let quip = "We also know there are known unknowns";
assert_eq!(quip.find("know"), Some(8));
assert_eq!(quip.rfind("know"), Some(31));
assert_eq!(quip.find("ya know"), None);
assert_eq!(quip.rfind(char::is_uppercase), Some(0));
```

- **slice.replace(pattern, replacement)** возвращает новую строку, получающуюся в результате замены всех вхождений `pattern` строкой `replacement`:

```
assert_eq!("The only thing we have to fear is fear itself"
           .replace("fear", "spin"),
           "The only thing we have to spin is spin itself");

assert_eq!("`Borrow` and `BorrowMut`"
           .replace(|ch:char| !ch.is_alphanumeric(), ""),
           "BorrowandBorrowMut");
```

- **slice.replacen(pattern, replacement, n)** делает то же самое, но производит не более `n` замен.

Обход текста

Стандартная библиотека предлагает несколько способов обхода текста срезки. На рис. 17.3 приведено несколько примеров.

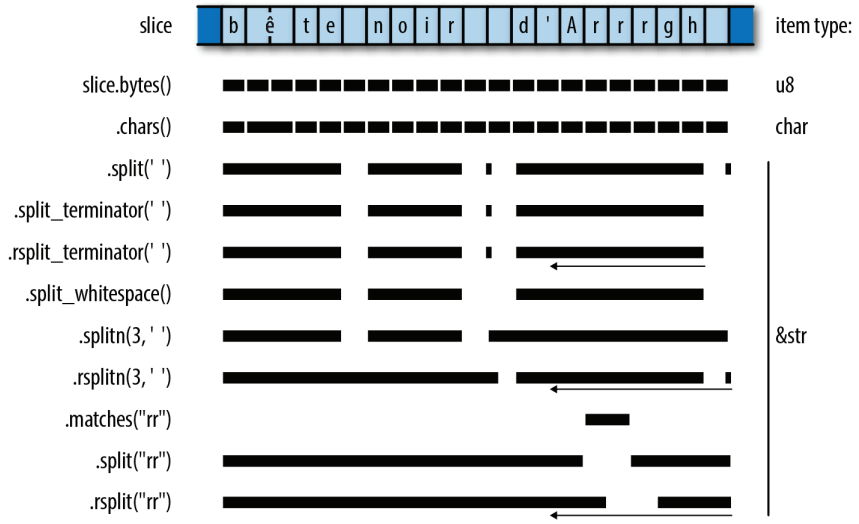


Рис. 17.3 ❖ Несколько способов обхода срезки

Семейства методов `split` и `match` можно считать взаимодополняющими: результатом разбиения являются диапазоны между совпадениями с образцом.

Для некоторых видов образцов поиск от конца к началу может давать другой набор значений, как, например, в случае разбиения по образцу `"rr"` на рисунке. Образцы, которые всегда сопоставляются с одним символом, не могут вести себя подобным образом. Если итератор порождает один и тот же набор объектов в любом направлении (т. е. изменяется только их порядок), то он является двусторонним – `DoubleEndedIterator`, т. е. мы можем применить его метод `rev` для обхода в противоположном порядке и получать объекты с любой стороны.

- **`slice.chars()`** возвращает итератор для обхода символов срезки `slice`.
- **`slice.char_indices()`** возвращает итератор для обхода символов срезки `slice` и их байтовых смещений:

```
assert_eq!("élan".char_indices().collect::<Vec<_>>(),
           vec![(0, 'é'), // has a two-byte UTF-8 encoding
                (2, 'l'),
                (3, 'a'),
                (4, 'n')]);
```

  Отметим, что это не эквивалентно методу `.chars().enumerate()`, поскольку мы получаем не просто порядковые номера символов, а смещения их первых байтов от начала срезки.
- **`slice.bytes()`** возвращает итератор для обхода байтов `slice`:

```
assert_eq!("élan".bytes().collect::<Vec<_>>(),
           vec![195, 169, b'l', b'a', b'n']);
```

- **`slice.lines()`** возвращает итератор для обхода строчек `slice`. Каждая строчка завершается строкой `"\n"` или `"\r\n"`. Каждый порождаемый объект является ссылкой `&str`, заимствованной у `slice`. Объекты не включают символов, завершающих строчки.
- **`slice.split(pattern)`** возвращает итератор для обхода участков `slice`, разделенных вхождениями образца `pattern`. Между соседними вхождениями, а также перед вхождением в начале `slice` и после вхождения в конце `slice` образуются пустые строки.
- Метод **`slice.rsplit(pattern)`** делает то же самое, но срезка `slice` просматривается от конца к началу, так что вхождения порождаются именно в таком порядке.
- Методы **`slice.split_terminator(pattern)`** и **`slice.rsplit_terminator(pattern)`** похожи с тем отличием, что образец трактуется как завершитель, а не как разделитель: если совпадение с `pattern` встречается в правом конце срезки `slice`, то итераторы не порождают пустую срезку, представляющую пустую строку между этим совпадением и концом срезки, как в случае `split` и `rsplit`. Например:

```
// Здесь символы ':' — разделители. Обратите внимание на последнюю "".
assert_eq!("jimb:1000:Jim Blandy:".split(':').collect::<Vec<_>>(),
           vec!["jimb", "1000", "Jim Blandy", ""]);

// Здесь символы '\n' — завершители.
assert_eq!("127.0.0.1 localhost\n\
            127.0.0.1 www.reddit.com\n"
           .split_terminator('\n').collect::<Vec<_>>(),
```

```
            vec!["127.0.0.1 localhost",
                 "127.0.0.1 www.reddit.com"]);
                 // Обратите внимание на отсутствие последней ""!
```

- Методы **slice.splitn(n, pattern)** и **slice.rsplitn(n, pattern)** похожи на **split** и **rsplit** с тем отличием, что строка разбивается не более чем на `n` срезок – соответственно по первым или последним `n-1` совпадениям с `pattern`.
- **slice.split_whitespace()** возвращает итератор для обхода разделенных пробелами участков `slice`. Серия из нескольких пробелов считается одним разделителем. Конечные пробелы игнорируются. Под «пробелом» понимается любой символ, для которого метод `char::is_whitespace` возвращает `true`.

```
let poem = "This is just to say\n\
            I have eaten\n\
            the plums\n\
            again\n";

assert_eq!(poem.split_whitespace().collect::<Vec<_>>(),
           vec!["This", "is", "just", "to", "say",
                "I", "have", "eaten", "the", "plums",
                "again"]);
```

- **slice.matches(pattern)** возвращает итератор для обхода совпадений с образцом `pattern` в срезке. Метод **slice.rmatches(pattern)** делает то же самое, но обход производится от конца к началу.
- **slice.match_indices(pattern)** и **slice.rmatch_indices(pattern)** похожи, но порождаемые объекты представляют собой пары `(offset, match)`, где `offset` – смещение первого байта совпадения от начала срезки, а `match` – само совпадение.

Усечение

Под *усечением* понимается удаление текста, обычно пробелов, в начале или в конце строки. Это часто бывает полезно для очистки входных данных, прочитанных из файла, когда в тексте присутствуют специально оставленные начальные отступы или случайно оставшиеся пробелы в конце строки.

- **slice.trim()** возвращает подсрезку `slice`, в которой нет начальных и конечных пробелов. **slice.trim_left()** исключает только начальные пробелы, а **slice.trim_right()** – только конечные.

```
assert_eq!("\t*.rs ".trim(), "*.rs");
assert_eq!("\t*.rs ".trim_left(), "*.rs ");
assert_eq!("\t*.rs ".trim_right(), "\t*.rs");
```

- **slice.trim_matches(pattern)** возвращает подсрезку `slice`, в которой исключены все вхождения образца `pattern` в начале и в конце. Методы **trim_left_matches** и **trim_right_matches** делают то же самое только для начальных или конечных вхождений.

```
assert_eq!("001990".trim_left_matches('0'), "1990");
```

Отметим, что слова `left` и `right` в именах этих методов всегда относятся к началу и концу срезки соответственно вне зависимости от направления текста.

Преобразование регистра

Методы `slice.to_uppercase()` и `slice.to_lowercase()` возвращают новую строку, содержащую текст срезки, преобразованный соответственно в верхний или нижний регистр. Длина результата может отличаться от длины исходной срезки (детали см. в разделе «Преобразование регистра символов» выше).

Создание значений других типов из строк

Rust предлагает стандартные характеристики для разбора строковых значений и порождения текстовых представлений значений других типов.

Если тип реализует характеристику `std::str::FromStr`, то он предоставляет стандартный способ разбора значения, представленного срезкой строки:

```
pub trait FromStr: Sized {
    type Err;
    fn from_str(s: &str) -> Result<Self, Self::Err>;
}
```

Все стандартные машинные типы реализуют `FromStr`:

```
use std::str::FromStr;

assert_eq!(usize::from_str("3628800"), Ok(3628800));
assert_eq!(f64::from_str("128.5625"), Ok(128.5625));
assert_eq!(bool::from_str("true"), Ok(true));

assert!(f64::from_str("not a float at all").is_err());
assert!(bool::from_str("TRUE").is_err());
```

Также эту характеристику реализует тип `std::net::IpAddr` – перечисление, в котором хранится адрес Интернета в формате IPv4 или IPv6:

```
use std::net::IpAddr;

let address = IpAddr::from_str("fe80::0000:3ea9:f4ff:fe34:7a50")?;
assert_eq!(address,
           IpAddr::from([0xfe80, 0, 0, 0, 0x3ea9, 0xf4ff, 0xfe34, 0x7a50]));
```

У срезок строки имеется метод `parse`, который преобразует строку в любой тип, реализующий `FromStr`. Как и в случае метода `Iterator::collect`, иногда необходимо явно указать желаемый тип, поэтому `parse` не всегда оказывается более понятным, чем вызов метода `from_str` напрямую:

```
let address = "fe80::0000:3ea9:f4ff:fe34:7a50".parse::<IpAddr>()?;
```

Преобразование других типов в строки

Существуют три основных способа преобразовать нетекстовое значение в строку:

- типы, у которых имеется естественное понятное человеку представление, могут реализовывать характеристику `std::fmt::Display`, которая позволяет использовать спецификатор формата `{}` в макросе `format!`:

  ```
  assert_eq!(format!("{}, wow", "doge"), "doge, wow");
  assert_eq!(format!("{}", true), "true");
  assert_eq!(format!("({:.3}, {:.3})", 0.5, f64::sqrt(3.0)/2.0),
             "(0.500, 0.866)");
  ```

```
// Используется тот же адрес `address`, что и выше.
let formatted_addr: String = format!("{}", address);
assert_eq!(formatted_addr, "fe80::3ea9:f4ff:fe34:7a50");
```

Все машинные числовые типы в Rust, а также символы строки и срезки реализуют `Display`. Типы интеллектуальных указателей `Box<T>`, `Rc<T>` и `Arc<T>` реализуют `Display`, если ее реализует сам тип `T`: отображаются они точно так же, как значение, на которое ведет указатель. Контейнеры, в частности `Vec` и `HashMap`, не реализуют `Display`, поскольку для них не существует естественного понятного человеку представления в виде единой сущности;

- если некий тип реализует характеристику `Display`, то стандартная библиотека автоматически реализует для него характеристику `std::str::ToString` с единственным методом `to_string`, который может оказаться удобнее, если гибкость `format!` не нужна:

```
// Продолжение предыдущего примера.
assert_eq!(address.to_string(), "fe80::3ea9:f4ff:fe34:7a50");
```

Характеристика `ToString` появилась раньше `Display` и обладает меньшей гибкостью. Для своих собственных типов обычно имеет смысл реализовывать `Display`, а не `ToString`;

- все открытые типы в стандартной библиотеке реализуют характеристику `std::fmt::Debug`, которая принимает значение и представляет его в виде, удобном для программистов. Самый простой способ использовать `Debug` для порождения строки дает спецификатор формата `{:?}` в макросе `format!`:

```
// Продолжение предыдущего примера.
let addresses = vec![address,
                     IpAddr::from_str("192.168.0.1")?];
assert_eq!(format!("{:?}", addresses),
           "[V6(fe80::3ea9:f4ff:fe34:7a50), V4(192.168.0.1)]");
```

Здесь используется общая реализация `Debug` для `Vec<T>`, где тип `T` сам реализует `Debug`. Для всех типов коллекций в Rust существуют такие реализации. Характеристику `Debug` следует реализовывать и для своих собственных типов. Обычно лучше всего поручить Rust вывести реализацию автоматически, как мы поступали для типа `Complex` выше:

```
#[derive(Copy, Clone, Debug)]
struct Complex { r: f64, i: f64 }
```

Помимо `Display` и `Debug`, в макросе `format!` и родственных ему используются и другие характеристики для форматирования значений. Мы рассмотрим их в разделе «Форматирование значений» ниже в этой главе и там же объясним, как они реализуются.

Заимствование в виде других текстообразных типов

Есть несколько способов заимствования содержимого срезки:

- срезки и строки реализуют характеристики `AsRef<str>`, `AsRef<[u8]>`, `AsRef<Path>` и `AsRef<OsStr>`. Многие стандартные библиотечные функции указывают эти характеристики в ограничениях на свои параметрические типы, так что строки и срезки им можно передавать непосредственно, даже если в сигна-

туре указан какой-то другой тип. Подробности см. в разделе «Характеристики AsRef и AsMut» главы 13;

- строки и срезки реализуют также характеристику `std::borrow::Borrow<str>`. В типах `HashMap` и `BTreeMap` эта характеристика используется, для того чтобы строки можно было использовать в качестве ключей таблицы, как в функции `[T]::binary_search`. Подробности см. в разделе «Характеристики Borrow и BorrowMut» главы 13.

Доступ к байтам текста в кодировке UTF-8

Есть два основных способа получить байты, представляющие текст. Выбор зависит от того, хотите ли вы получить байты во владение или только заимствовать их:

- **`slice.as_bytes()`** заимствует байты срезки `slice` в виде ссылки `&[u8]`. Поскольку ссылка неизменяемая, срезка может предполагать, что последовательность байтов остается корректной в смысле UTF-8;
- **`string.into_bytes()`** принимает владение строкой `string` и возвращает вектор `Vec<u8>` байтов строки по значению. Это преобразование обходится дешево, потому что просто передается вектор, который строка до этого использовала как буфер. Поскольку строки `string` больше не существует, следить за тем, чтобы байты и дальше образовывали корректную последовательность UTF-8, нет необходимости, и вызывающая сторона может модифицировать `Vec<u8>`, как сочтет нужным.

Порождение текста из данных в кодировке UTF-8

Если имеется блок байтов, предположительно содержащий данные в кодировке UTF-8, то преобразовать его в строки или срезки можно несколькими способами, а выбор того или другого зависит от того, как вы хотите обрабатывать ошибки:

- **`str::from_utf8(byte_slice)`** принимает байтовую срезку `&[u8]` и возвращает `Result`: либо `Ok(&str)`, если `byte_slice` содержит корректную последовательность байтов в смысле UTF-8, либо ошибку;
- **`String::from_utf8(vec)`** пытается сконструировать строку из вектора `Vec<u8>`, переданного по значению. Если `vec` содержит корректную последовательность UTF-8, то `from_utf8` возвращает `Ok(string)`, где `string` принимает владение `vec` и использует его в качестве своего буфера. Не производится ни выделения памяти, ни копирования.
  Если последовательность байтов не является корректным текстом в кодировке UTF-8, то метод возвращает `Err(e)`, где `e` – значение типа `FromUtf8Error`. Вызов `e.into_bytes()` дает исходный вектор `vec`, так что он не теряется в случае ошибки преобразования.

```
let good_utf8: Vec<u8> = vec![0xe9, 0x8c, 0x86];
assert_eq!(String::from_utf8(good_utf8).ok(),   Some("錆".to_string()));

let bad_utf8: Vec<u8> = vec![0x9f, 0xf0, 0xa6, 0x80];
let result = String::from_utf8(bad_utf8);
assert!(result.is_err());
// Поскольку String::from_utf8 завершился ошибкой, исходный вектор
// не потреблен, и мы можем получить его из значения ошибки.
assert_eq!(result.unwrap_err().into_bytes(),
           vec![0x9f, 0xf0, 0xa6, 0x80]);
```

- **`String::from_utf8_lossy(byte_slice)`** пытается сконструировать `String` или `&str` из разделяемой байтовой срезки `&[u8]`. Такое преобразование всегда завершается успешно, поскольку вместо неправильных в смысле UTF-8 байтов подставляется символ замены Юникода. Возвращаемое значение имеет тип `Cow<str>`: оно либо заимствует `&str` непосредственно у `byte_slice`, если содержит корректную последовательность байтов UTF-8, либо владеет новой строкой `String`, в которой вместо некорректных байтов подставлены символы замены. Таким образом, если срезка `byte_slice` корректна, то не производится ни выделения памяти, ни копирования. Подробнее мы обсудим тип `Cow<str>` в разделе «Откладывание выделения памяти» ниже;
- если точно известно, что вектор `Vec<u8>` содержит корректную последовательность UTF-8, то можно вызвать небезопасную функцию **`String::from_utf8_unchecked`**. Она просто обертывает `Vec<u8>` строкой `String` и возвращает ее безо всякой проверки байтов. Ответственность за недопущение некорректных строк UTF-8 в систему теперь несете вы, потому функция и помечена ключевым словом `unsafe`;
- аналогично метод **`str::from_utf8_unchecked`** принимает ссылку `&[u8]` и возвращает ее в виде `&str`, не проверяя корректности UTF-8. Как и в случае `String::from_utf8_unchecked`, за безопасность отвечаете вы сами.

Откладывание выделения памяти

Пусть мы хотим, чтобы программа приветствовала пользователя. В Unix можно было бы написать:

```
fn get_name() -> String {
    std::env::var("USER") // в Windows используется "USERNAME"
        .unwrap_or("кем бы ты ни был".to_string())
}

println!("Приветствую, {}!", get_name());
```

Пользователей Unix эта программа приветствует по имени пользователя. Для пользователей Windows и для тех, кто по прихоти судьбы остался без имени, выводится альтернативный готовый текст.

Функция `std::env::var` возвращает `String` – и тому есть веские причины, в которые мы здесь не будем вдаваться. Но тогда и альтернативный текст следует возвращать в виде `String`. А это досадно: ведь когда `get_name` возвращает статическую строку, никакое выделение памяти не нужно. Суть проблемы в том, что иногда возвращенное имя `name` должно быть принадлежащей какому-то владельцу строкой `String`, а иногда иметь тип `&'static str`, и какой случай имеет место, мы не знаем до момента выполнения программы. Эта динамичность и побуждает нас обратиться к типу `std::borrow::Cow`, реализующему «клонирование при записи», в котором могут храниться либо принадлежащие, либо заимствованные данные.

Как было объяснено в разделе «Borrow и ToOwned за работой: скромное копирование при записи» главы 13, `Cow<'a, T>` – перечисление с двумя вариантами: `Owned` и `Borrowed`. В `Borrowed` хранится ссылка `&'a T`, а в `Owned` – принадлежащий владельцу вариант `&T`: `String` для типа `&str`, `Vec<i32>` для `&[i32]` и т. д. В любом случае `Cow<'a, T>`

может породить значение типа `&T`, пригодное к использованию. На самом деле `Cow<'a, T>` разыменовывается в `&T`, т. е. ведет себя как своего рода интеллектуальный указатель.

Изменив функцию `get_name`, так чтобы она возвращала `Cow`, получим:

```
use std::borrow::Cow;

fn get_name() -> Cow<'static, str> {
    std::env::var("USER")
        .map(|v| Cow::Owned(v))
        .unwrap_or(Cow::Borrowed("кем бы ты ни был"))
}
```

Если переменная среды `"USER"` была прочитана успешно, то `map` возвращает результирующую строку `String` в виде `Cow::Owned`. В противном случае `unwrap_or` возвращает ссылку на статическую строку `&str` в виде `Cow::Borrowed`. Вызывающая сторона при этом не изменяется:

```
println!("Приветствую, {}!", get_name());
```

Коль скоро `T` реализует характеристику `std::fmt::Display`, значение `Cow<'a, T>` выводится на экран так же, как `T`.

Тип `Cow` полезен и в тех случаях, когда заимствованный текст иногда модифицируется, а иногда нет. Если изменять его не нужно, то можно продолжать заимствовать. Но благодаря свойству копирования при записи `Cow` может предоставить нам во владение изменяемую копию значения по запросу. Метод `to_mut` типа `Cow` сначала убеждается, что в перечислении хранится вариант `Cow::Owned`, применяя при необходимости реализацию характеристики `ToOwned`, а затем возвращает изменяемую ссылку на значение.

Таким образом, если для некоторых, но не для всех пользователей определен титул, то можно написать такой код:

```
fn get_title() -> Option<&'static str> { ... }

let mut name = get_name();
if let Some(title) = get_title() {
    name.to_mut().push_str(", ");
    name.to_mut().push_str(title);
}

println!("Приветствую, {}!", name);
```

В результате может быть напечатано:

```
$ cargo run
Приветствую, jimb, Esq.!

$
```

Обратим внимание на приятную особенность: если `get_name()` возвращает статическую строку, а `get_title` – `None`, то `Cow` просто протолкнет статическую строку прямо в `println!`. Нам удалось отложить выделение памяти до того момента, когда (и если) оно действительно необходимо, и при этом сохранить простоту и линейность кода.

Поскольку тип `Cow` часто используется для строк, в стандартной библиотеке имеется специальная поддержка для `Cow<'a, str>`. Этот тип позволяет выполнять

преобразования в типы `String` и `&str` и обратно, так что функцию можно записать более лаконично:

```
fn get_name() -> Cow<'static, str> {
    std::env::var("USER")
        .map(|v| v.into())
        .unwrap_or("кем бы ты ни был".into())
}
```

Тип `Cow<'a, str>` реализует также характеристики `std::ops::Add` и `std::ops::AddAssign`, поэтому для добавления титула к имени можно написать:

```
if let Some(title) = get_title() {
    name += ", ";
    name += title;
}
```

Или, поскольку строка может быть местом назначения для макроса `write!`:

```
use std::fmt::Write;

if let Some(title) = get_title() {
    write!(name.to_mut(), ", {}", title).unwrap();
}
```

Как и раньше, никакое выделение памяти не производится, пока не делается попытка модифицировать `Cow`.

Следует помнить, что не всякое значение типа `Cow<..., str>` обязано иметь время жизни `'static`: тип `Cow` может заимствовать ранее вычисленный текст до того момента, как понадобится настоящая копия.

Строки как универсальные коллекции

Тип `String` реализует характеристики `std::default::Default` и `std::iter::Extend`: метод `default` возвращает пустую строку, а метод `extend` может дописывать символы, срезки строки и строки в конец строки. Та же самая комбинация характеристик реализуется и другими типами коллекций в Rust, например `Vec` и `HashMap`, в интересах таких механизмов универсального конструирования, как методы `collect` и `partition`.

Тип `&str` также реализует характеристику `Default` и возвращает пустую срезку. Это удобно в некоторых случаях; например, чтобы автоматически вывести `Default` для структур, содержащих срезки.

Форматирование значений

В этой книге мы неоднократно использовали макросы форматирования текста, например `println!`:

```
println!("{:.3}µs: relocated {} at {:#x} to {:#x}, {} bytes",
         0.84391, "object",
         140737488346304_usize, 6299664_usize, 64);
```

При этом печатается:

```
0.844µs: relocated object at 0x7fffffffdcc0 to 0x602010, 64 bytes
```

Строковый литерал играет роль шаблона вывода: каждое вхождение `{...}` заменяется отформатированным представлением одного из последующих аргументов. Шаблонная строка должна быть константой, чтобы Rust мог проверить типы аргументов на этапе компиляции. Все без исключения аргументы должны использоваться, иначе компилятор выдаст ошибку.

Мини-язык форматных строк находит применение в нескольких библиотечных средствах:

- макрос `format!` служит для построения строк;
- макросы `println!` и `print!` записывают отформатированный текст в стандартный поток вывода;
- макросы `writeln!` и `write!` записывают его в указанный поток вывода;
- макрос `panic!` служит для построения (желательно информативного) сообщения, выражающего полное смятение.

Средства форматирования в Rust открыты для расширения. Эти макросы можно распространить на пользовательские типы, реализовав характеристики форматирования из модуля `std::fmt`. А макрос `format_args!` и тип `std::fmt::Arguments` можно использовать для поддержки языка форматирования в своих функциях и макросах.

Макросы форматирования всегда заимствуют разделяемые ссылки на свои аргументы, они никогда не принимают владения ими и не изменяют их.

Конструкции `{...}` в шаблонной строке называются *параметрами формата* и имеют вид `{which:how}`. Обе части факультативны, нередко можно встретить просто `{}`.

Значение *which* указывает, какой аргумент из числа следующих за шаблоном нужно подставить вместо этого параметра. Аргументы можно задавать по индексу или по имени. Параметры, в которых *which* отсутствует, просто сопоставляются с аргументами по порядку слева направо.

Значение *how* говорит, как форматировать аргумент: заполнение, точность, основание системы счисления и т. д. Если часть *how* присутствует, то двоеточие перед ней обязательно.

Ниже приведено несколько примеров:

Шаблонная строка	Список аргументов	Результат
`"number of {}: {}"`	`"elephants", 19`	`"number of elephants: 19"`
`"from {1} to {0}"`	`"the grave", "the cradle"`	`"from the cradle to the grave"`
`"v = {:?}"`	`vec![0,1,2,5,12,29]`	`"v = [0, 1, 2, 5, 12, 29]"`
`"name = {:?}"`	`"Nemo"`	`"name = \"Nemo\""`
`"{:8.2} km/s"`	`11.186`	`"  11.19 km/s"`
`"{:20} {:02x} {:02x}"`	`"adc #42", 105, 42`	`"adc #42          69 2a"`
`"{1:02x} {2:02x} {0}"`	`"adc #42", 105, 42`	`"69 2a adc #42"`
`"{lsb:02x} {msb:02x} {insn}"`	`insn="adc #42", lsb=105, msb=42`	`"69 2a adc #42"`

Чтобы включить в выходную строку символ `'{'` или `'}'`, его следует повторить в шаблоне два раза:

```
assert_eq!(format!("{{a, c}} ⊂ {{a, b, c}}"),
           "{a, c} ⊂ {a, b, c}");
```

Форматирование текстовых значений

При форматировании значений текстовых типов, например `&str` или `String` (`char` рассматривается как односимвольная строка), часть *how* параметра форматирования может состоять из следующих элементов, причем все они необязательные:

- **предельная длина текста** – Rust усекает аргумент, если он оказывается длиннее. Если предел не задан, текст используется целиком;
- **минимальная ширина поля** – если после усечения аргумент оказывается короче, то Rust дополняет его справа (по умолчанию) пробелами (по умолчанию), чтобы довести поле до такой ширины. Если ширина не задана, аргумент не дополняется;
- **выравнивание** – если аргумент необходимо дополнить до минимальной ширины, то этот элемент говорит, где в поле должен находиться текст: `<` – в начале, `^` – в центре, `>` – в конце;
- символ **дополнения** – если он не задан, то аргумент дополняется пробелами. Если символ дополнения задан, то должно быть задано и выравнивание.

Ниже приведено несколько примеров с результатами. Во всех случаях предполагается один и тот же восьмибуквенный аргумент `"bookends"`.

Используемые элементы	Шаблонная строка	Результат
Все по умолчанию	`"{}"`	`"bookends"`
Минимальная ширина поля	`"{:4}"` `"{:12}"`	`"bookends"` `"bookends  "`
Предельная длина текста	`"{:.4}"` `"{:.12}"`	`"book"` `"bookends"`

Ширина поля, предельная длина	`"{:12.20}"`
	`"{:4.20}"`
	`"{:4.6}"`
	`"{:6.4}"`
		`"bookends  "`
		`"bookends"`
		`"booken"`
		`"book  "`

Выравнивание влево, ширина	`"{:<12}"`	`"bookends  "`
Выравнивание по центру, ширина	`"{:^12}"`	`" bookends "`
Выравнивание вправо, ширина	`"{:>12}"`	`"  bookends"`
Дополнение знаком '=', выравнивание по центру, ширина	`"{:=^12}"`	`"==bookends==="`
Дополнение знаком '*', выравнивание вправо, ширина, предельная длина	`"{:*>12.4}"`	`"********book"`

У форматера Rust наивное представление о «ширине»: он считает, что каждый символ занимает один столбец, и не обращает внимания на комбинируемые символы, символы азбуки катакана половинной ширины, пробелы нулевой ширины и многие другие неприглядные стороны Юникода. Например:

```
assert_eq!(format!("{:4}", "th\u{e9}"),   "th\u{e9} ");
assert_eq!(format!("{:4}", "the\u{301}"), "the\u{301}");
```

Юникод говорит, что обе эти строки эквивалентны `"thé"`, но форматер Rust не знает, что для символа `'\u{301}'`, COMBINING ACUTE ACCENT, необходима специальная обработка. Он правильно дополняет первую строку, но считает, что ширина второй равна 4 столбцам, и не дополняет ее. Легко понять, как можно было бы улучшить Rust в этом конкретном случае, но настоящее многоязыковое форматирование текста для всех представленных в Юникоде систем письменности –

это монументальная задача, которую лучше оставить платформенным библиотекам построения пользовательского интерфейса или, быть может, сгенерировать HTML и CSS и поручить браузеру разобраться.

Помимо `&str` и `String`, макросам форматирования можно передать также интеллектуальный указатель на текстовое значение, например `Rc<String>` или `Cow<'a, str>`.

Поскольку имена файлов могут и не быть корректными строками в кодировке UTF-8, тип `std::path::Path` не является текстовым; непосредственно передать значение типа `std::path::Path` макросу форматирования нельзя. Однако метод `display` этого типа возвращает значение, допускающее форматирование; он разрешает все проблемы платформенно-зависимым способом.

```
println!("обрабатывается файл: {}", path.display());
```

Форматирование чисел

Если аргумент макроса форматирования имеет числовой тип, например `usize` или `f64`, то часть *how* соответствующего параметра может состоять из следующих элементов, причем все они необязательные:

- **дополнение** и **выравнивание** работают так же, как для текстовых типов;
- знак `+`, означающий, что знак числа всегда следует показывать, даже если число положительное;
- знак `#`, позволяющий задать основание системы счисления, например `0x` или `0b`. См. пункт «нотация» ниже;
- знак `0`, означающий, что для удовлетворения требования на минимальную ширину поля нужно дополнить число начальными нулями вместо обычного подхода к дополнению;
- **минимальная ширина поля** – если отформатированное число уже, то Rust дополняет его слева (по умолчанию) пробелами (по умолчанию) до указанной ширины;
- **точность** для аргументов с плавающей точкой – показывает, сколько цифр оставлять после десятичной точки. Rust округляет число или добавляет необходимое число нулей, чтобы получить ровно столько цифр в дробной части. Если этот элемент не задан, то Rust пытается точно представить значение с помощью минимального количества цифр. Для аргументов целого типа точность игнорируется;
- **нотация** – для целых типов допустимы значения `b` (двоичное), `o` (восьмеричное), `x` или `X` (шестнадцатеричное соответственно со строчными или заглавными буквами). Если включен также знак `#`, то добавляются еще явные префиксы основания в стиле Rust: `0b`, `0o`, `0x`, `0X`. Для типов с плавающей точкой основание `e` или `E` означает научную нотацию с нормировочным коэффициентом и использованием «e» или «E» для обозначения показателя степени. Если нотация не задана, то числа представляются в десятичной системе.

Ниже приведено несколько примеров форматирования числа `1234` типа `i32`:

Используемые элементы	Шаблонная строка	Результат
Все по умолчанию	`"{}"`	`"1234"`
Обязательный знак	`"{:+}"`	`"+1234"`

Используемые элементы	Шаблонная строка	Результат
Минимальная ширина поля	`"{:12}"` `"{:2}"`	`"  1234"` `"1234"`
Знак, ширина	`"{:+12}"`	`"  +1234"`
Начальные нули, ширина	`"{:012}"`	`"000000001234"`
Знак, нули, ширина	`"{:+012}"`	`"+00000001234"`
Выравнивание влево, ширина	`"{:<12}"`	`"1234  "`
Выравнивание по центру, ширина	`"{:^12}"`	`"  1234  "`
Выравнивание вправо, ширина	`"{:>12}"`	`"  1234"`
Выравнивание влево, знак, ширина	`"{:<+12}"`	`"+1234  "`
Выравнивание по центру, знак, ширина	`"{:^+12}"`	`"  +1234  "`
Выравнивание вправо, знак, ширина	`"{:>+12}"`	`"  +1234"`
Дополнение знаками '=', по центру, ширина	`"{:=^12}"`	`"====1234===="`
Двоичная нотация	`"{:b}"`	`"10011010010"`
Ширина, восьмеричная нотация	`"{:12o}"`	`"  2322"`
Знак, ширина, шестнадцатеричная нотация	`"{:+12x}"`	`"  +4d2"`
Знак, ширина, шестнадцатеричная с заглавными буквами	`"{:+12X}"`	`"  +4D2"`
Знак, явный префикс основания, ширина, шестнадцатеричная	`"{:+#12x}"`	`"  +0x4d2"`
Знак, префикс, нули, ширина, шестнадцатеричная	`"{:+#012x}"` `"{:+#06x}"`	`"+0x0000004d2"` `"+0x4d2"`

Как показывают последние два примера, минимальная ширина поля относится ко всему числу, включая знак, префикс основания и все остальное

Отрицательные числа всегда отображаются со знаком. Результат будет выглядеть, как в примерах с «обязательным знаком».

Если запрошены начальные нули, то режим выравнивания и символы дополнения просто игнорируются, поскольку нулями заполняются все свободные позиции в поле.

Форматирование чисел с плавающей точкой показано на примере аргумента `1234.5678`.

Используемые элементы	Шаблонная строка	Результат
Все по умолчанию	`"{}"`	`"1234.5678"`
Точность	`"{:.2}"` `"{:.6}"`	`"1234.57"` `"1234.567800"`
Минимальная ширина поля	`"{:12}"`	`"  1234.5678"`
Ширина, точность	`"{:12.2}"` `"{:12.6}"`	`"  1234.57"` `" 1234.567800"`
Начальные нули, ширина, точность	`"{:012.6}"`	`"01234.567800"`
Научная нотация	`"{:e}"`	`"1.2345678e3"`
Научная, точность	`"{:.3e}"`	`"1.235e3"`
Научная, ширина, точность	`"{:12.3e}"` `"{:12.3E}"`	`"  1.235e3"` `"  1.235E3"`

Форматирование прочих типов

Помимо строк и чисел, можно форматировать еще несколько стандартных библиотечных типов:

- все типы ошибок можно форматировать непосредственно, что упрощает включение их в сообщения об ошибках. Любой тип ошибки должен реали-

зовывать характеристику `std::error::Error`, расширяющую обычную форматную характеристику `std::fmt::Display`. Отсюда следует, что любой тип, реализующий `Error`, готов к форматированию;

- можно форматировать типы адресов протоколов Интернета, например `std::net::IpAddr` и `std::net::SocketAddr`;
- булевы значения `true` и `false` также допускают форматирование, хотя обычно показывать их конечному пользователю – не лучшая идея.

Применимы те же параметры формата, что для строк. Предельная длина, ширина поля и режимы выравнивания работают, как и ожидается.

Форматирование значений для отладки

Для отладки и протоколирования имеется конструкция `{:?}`, позволяющая отформатировать любой открытый тип из стандартной библиотеки способом, полезным программистам. Она пригодна для инспектирования векторов, срезок, кортежей, хеш-таблиц, потоков и сотен других типов.

Например, можно написать:

```
use std::collections::HashMap;
let mut map = HashMap::new();
map.insert("Portland", (45.5237606,-122.6819273));
map.insert("Taipei",   (25.0375167, 121.5637));
println!("{:?}", map);
```

В результате будет напечатано:

```
{"Taipei": (25.0375167, 121.5637), "Portland": (45.5237606, -122.6819273)}
```

Типы `HashMap` и `(f64, f64)` уже знают, как себя форматировать, так что никаких усилий с нашей стороны не нужно.

Если включить в параметр формата знак `#`, то Rust постарается красиво отформатировать значение. Так, макрос `println!("{:#?}", map)` выведет те же данные следующим образом:

```
{
    "Taipei": (
        25.0375167,
        121.5637
    ),
    "Portland": (
        45.5237606,
        -122.6819273
    )
}
```

Точно такая форма не гарантируется, и иногда при переходе к новой версии Rust формат изменяется.

Как уже отмечалось, чтобы заставить собственные типы правильно обрабатывать `{:?}`, можно воспользоваться синтаксисом `#[derive(Debug)]`:

```
#[derive(Copy, Clone, Debug)]
struct Complex { r: f64, i: f64 }
```

При таком определении мы можем использовать формат {:?} для печати значений типа Complex:

```
let third = Complex { r: -0.5, i: f64::sqrt(0.75) };
println!("{:?}", third);
```

В результате печатается:

```
Complex { r: -0.5, i: 0.8660254037844386 }
```

Для отладки это годится, но было бы лучше, если бы по формату {} можно было напечатать комплексное число в более привычной форме: -0.5+0.8660254037844386i. В разделе «Форматирование пользовательских типов» ниже мы покажем, как это сделать.

Форматирование указателей для отладки

Обычно если макросу форматирования передается какой-то указатель – ссылка, Box, Rc – то макрос просто форматирует значение, на которое этот указатель ведет, а указатель как таковой его не интересует. Но во время отладки видеть указатель иногда полезно – он может служить чем-то вроде «имени» отдельного значения, что важно при исследовании структур с циклами или разделяемыми ссылками.

Конструкция {:p} служит для форматирования ссылок, боксов и других указательных типов и адресов:

```
use std::rc::Rc;
let original = Rc::new("mazurka".to_string());
let cloned = original.clone();
let impostor = Rc::new("mazurka".to_string());
println!("текст:     {}, {}, {}",       original, cloned, impostor);
println!("указатели: {:p}, {:p}, {:p}", original, cloned, impostor);
```

Этот код печатает:

```
текст: mazurka, mazurka, mazurka
указатели: 0x7f99af80e000, 0x7f99af80e000, 0x7f99af80e030
```

Разумеется, значения указателей изменяются при каждом прогоне, но все равно из сравнения адресов видно, что первые два указателя ссылаются на одну строку String, а третий – на другую.

Адреса выглядят как шестнадцатеричная мешанина, поэтому улучшенная визуализация желательна, но для быстрого получения результата формат {:p} вполне эффективен.

Ссылка на аргументы по индексу или по имени

В параметре формата можно явно указать, какой аргумент использовать, например:

```
assert_eq!(format!("{1},{0},{2}", "zeroth", "first", "second"),
           "first,zeroth,second");
```

Можно также включить свойства форматирования после двоеточия:

```
assert_eq!(format!("{2:#06x},{1:b},{0:=>10}", "first", 10, 100),
           "0x0064,1010,=====first");
```

Есть также возможность выбирать аргументы по имени. При этом сложные шаблоны с большим числом параметров становится проще читать. Например:

```
assert_eq!(format!("{description:.<25}{quantity:2} @ {price:5.2}",
                   price=3.25,
                   quantity=3,
                   description="Maple Turmeric Latte"),
           "Maple Turmeric Latte..... 3 @  3.25");
```

(Здесь именованные аргументы напоминают ключевые аргументы в Python, но это всего лишь особенность макросов форматирования, а не часть синтаксиса вызова функций в Rust.)

В одном макросе разрешается смешивать обращения к параметрам по индексу и по имени, а также позиционные параметры (без указания индекса или имени). Позиционные параметры сопоставляются с аргументами слева направо, как если бы индексированных и именованных параметров не существовало:

```
assert_eq!(format!("{mode} {2} {} {}",
                   "people", "eater", "purple", mode="flying"),
           "flying purple people eater");
```

Именованные аргументы должны находиться в конце списка.

Динамическая ширина и точность

Минимальная ширина поля, предельная длина текста и числовая точность необязательно должны быть фиксированными, их значения можно задавать во время выполнения.

В следующем выражении строка `content` выравнивается на правую границу поля шириной 20 символов:

```
format!("{:>20}", content)
```

А если бы мы хотели задать ширину поля во время выполнения, то написали бы:

```
format!("{:>1$}", content, get_width())
```

`1$` вместо ширины поля говорит макросу `format!`, что значение второго аргумента следует интерпретировать как ширину. Этот аргумент должен иметь тип `usize`. На него можно также сослаться по имени:

```
format!("{:>width$}", content, width=get_width())
```

Та же идея работает для предельной длины текста:

```
format!("{:>width$.limit$}", content,
        width=get_width(), limit=get_limit())
```

Вместо предельной длины текста или точности числа с плавающей точкой можно указать также `*`, это означает, что в качестве значения нужно брать следующий позиционный аргумент. В следующем предложении из строки `content` печатается не более `get_limit()` символов:

```
format!("{:.*}", get_limit(), content)
```

Аргумент, интерпретируемый как точность, должен иметь тип `usize`. Аналогичный синтаксис для ширины поля не предусмотрен.

Форматирование пользовательских типов

В макросах форматирования для преобразования значений в текстовую форму используется ряд характеристик, определенных в модуле `std::fmt`. Реализовав одну или несколько из них в своем типе, вы заставите эти макросы форматировать значения этого типа.

Элемент «нотация» в параметре формата показывает, какую характеристику должен реализовывать соответствующий ему аргумент.

Нотация	Пример	Характеристика	Назначение
Отсутствует	`{}`	`std::fmt::Display`	Текст, числа, ошибки: всеобъемлющая характеристика
b	`{bits:#b}`	`std::fmt::Binary`	Числа в двоичном формате
o	`{#5o}`	`std::fmt::Octal`	Числа в восьмеричном формате
x	`{:4x}`	`std::fmt::LowerHex`	Числа в шестнадцатеричном формате со строчными буквами
X	`{:016x}`	`std::fmt::UpperHex`	Числа в шестнадцатеричном формате с заглавными буквами
e	`{:.3e}`	`std::fmt::LowerExp`	Числа с плавающей точкой в научной нотации
E	`{:.3E}`	`std::fmt::UpperExp`	То же, с буквой E
?	`{:#?}`	`std::fmt::Debug`	Отладочный формат, для разработчиков
p	`{:p}`	`std::fmt::Pointer`	Указатель в виде адреса, для разработчиков

Снабжая определение типа атрибутом `#[derive(Debug)]`, чтобы можно было использовать формат `{:?}`, мы просто просим Rust реализовать характеристику `std::fmt::Debug`.

Все форматные характеристики имеют общую структуру и отличаются только именами. Мы расскажем о них на примере характеристики `std::fmt::Display`:

```
trait Display {
    fn fmt(&self, dest: &mut std::fmt::Formatter)
        -> std::fmt::Result;
}
```

Задача метода `fmt` – построить надлежащим образом отформатированное представление аргумента `self` и записать его в `dest`. Аргумент `dest` не только играет роль потока вывода, но и содержит детали, полученные в процессе разбора параметра формата, например режим выравнивания и минимальную ширину поля.

Например, выше в этой главе мы сказали, что было бы хорошо представлять значения типа `Complex` в привычной форме `a + bi`. Ниже приведена реализация характеристики `Display`, делающая именно это:

```
use std::fmt;

impl fmt::Display for Complex {
    fn fmt(&self, dest: &mut fmt::Formatter) -> fmt::Result {
        let i_sign = if self.i < 0.0 { '-' } else { '+' };
        write!(dest, "{} {} {}i", self.r, i_sign, f64::abs(self.i))
    }
}
```

Здесь мы воспользовались тем фактом, что `Formatter` – это, ко всему прочему, еще и поток вывода, поэтому большую часть работы за нас может проделать макрос `write!`. При такой реализации справедливы следующие утверждения:

```
let one_twenty = Complex { r: -0.5, i: 0.866 };
assert_eq!(format!("{}", one_twenty),
           "-0.5 + 0.866i");

let two_forty = Complex { r: -0.5, i: -0.866 };
assert_eq!(format!("{}", two_forty),
           "-0.5 - 0.866i");
```

Иногда бывает полезно представить комплексные числа в полярных координатах: если провести на комплексной плоскости отрезок, соединяющий число с началом координат, то полярными координатами будут длина этого отрезка и угол между ним и положительным направлением оси x, отсчитываемый по часовой стрелке. Знак `#` в параметре формата обычно выбирает ту или иную форму представления, в реализации `Display` его можно было бы рассматривать как запрос на выбор полярной формы:

```
impl fmt::Display for Complex {
    fn fmt(&self, dest: &mut fmt::Formatter) -> fmt::Result {
        let (r, i) = (self.r, self.i);
        if dest.alternate() {
            let abs = f64::sqrt(r * r + i * i);
            let angle = f64::atan2(i, r) / std::f64::consts::PI * 180.0;
            write!(dest, "{} ∠{}°", abs, angle)
        } else {
            let i_sign = if i < 0.0 { '-' } else { '+' };
            write!(dest, "{} {} {}i", r, i_sign, f64::abs(i))
        }
    }
}
```

А вот пример использования этой реализации:

```
let ninety = Complex { r: 0.0, i: 2.0 };
assert_eq!(format!("{}", ninety),
           "0 + 2i");
assert_eq!(format!("{:#}", ninety),
           "2 ∠90°");
```

Хотя методы `fmt` форматных характеристик возвращают значение типа `fmt::Result` (тип `Result`, специфичный для модуля), нужно только распространять ошибки операций с форматером `Formatter`, как это сделано в реализации `fmt::Display` при обращениях к `write!`; сами функции форматирования не должны порождать никаких ошибок. Благодаря этому такие макросы, как `format!`, могут возвращать просто `String`, а не `Result<String, ...>`, т. к. дописывание форматированного текста в конец `String` всегда завершается успешно. Это также гарантирует, что любые ошибки, полученные от `write!` или `writeln!`, отражают реальные проблемы базового потока ввода-вывода, а не проблемы форматирования.

В типе `Formatter` много других полезных методов, в т. ч. для работы со структурированными данными: отображениями, списками и т. д. Здесь мы не будем их рассматривать, подробности см. в онлайновой документации.

Применение языка форматирования в своем коде

Мы можем написать собственные функции и макросы, принимающие шаблоны и аргументы форматирования. Для этого нам понадобится макрос `format_args!` и тип `std::fmt::.Arguments`

Пусть, например, программа должна записывать в журнал сообщения о состоянии, и мы хотим использовать для этого язык форматирования текста, встроенный в Rust. Начать можно было бы так:

```
fn logging_enabled() -> bool {
    ...
}

use std::fs::OpenOptions;
use std::io::Write;

fn write_log_entry(entry: std::fmt::Arguments) {
    if logging_enabled() {
        // Пока не будем усложнять - каждый раз открываем файл заново.
        let mut log_file = OpenOptions::new()
            .append(true)
            .create(true)
            .open("log-file-name")
            .expect("ошибка при открытии файла журнала");

        log_file.write_fmt(entry)
            .expect("ошибка при записи в файл журнала");
    }
}
```

Функция `write_log_entry` вызывается так:

```
write_log_entry(format_args!("Hark! {:?}\n", mysterious_value));
```

На этапе компиляции макрос `format_args!` разбирает шаблонную строку, проверяет типы аргументов и сообщает о найденных ошибках. На этапе выполнения он вычисляет аргументы и строит значение типа `Arguments`, в котором собрана вся информация, необходимая для форматирования текста: предварительно разобранный шаблон и разделяемые ссылки на значения аргументов.

Конструирование значения `Arguments` обходится дешево: нужно только собрать вместе несколько указателей. Пока никакое форматирование еще не производится, мы только готовим необходимую информацию. Это важный момент: если протоколирование выключено, то время, потраченное на преобразование чисел в десятичную запись, дополнение значений и т. д., было бы потрачено впустую.

Тип `File` реализует характеристику `std::io::Write`, метод `write_fmt` которой принимает `Argument` и выполняет форматирование. Результат выводится в базовый поток.

Это обращение к функции `write_log_entry` выглядит некрасиво. И тут на помощь приходит макрос:

```
macro_rules! log { // знак ! после имени в определения макросов не нужен
    ($format:tt, $($arg:expr),*) => (
        write_log_entry(format_args!($format, $($arg),*))
    )
}
```

Мы будем подробно рассматривать макросы в главе 20. А пока примите на веру, что здесь действительно определен макрос `log!`, который передает свои аргументы макросу `format_args!`, а затем вызывает нашу функцию `write_log_entry` с получившимся в результате значением `Arguments` в качестве аргумента. Все форматные макросы, включая `println!`, `writeln!` и `format!`, устроены примерно так же.

Макрос `log!` используется следующим образом:

```
log!("О день и ночь! Все это крайне странно! {:?}\n",
     mysterious_value);
```

Так-то лучше.

Регулярные выражения

Официальной библиотекой регулярных выражений для Rust является внешний крейт `regex`. Он предоставляет обычные функции поиска и сравнения с образцом. Он может похвастаться хорошей поддержкой Юникода, но вместе с тем содержит средства для поиска отдельных байтов. Правда, он не поддерживает некоторых возможностей, присутствующих в других пакетах, в т. ч. обратные ссылки и «оглядывание», но благодаря этим упрощениям гарантируется, что время поиска линейно зависит от длины выражения и длины просматриваемого текста. А это делает `regex` безопасным даже при использовании регулярных выражений и текстов, полученных из ненадежных источников.

В этой книге приводится лишь краткий обзор крейта `regex`, подробности смотрите в онлайновой документации.

Хотя крейт `regex` не принадлежит `std`, его сопровождает та же команда разработчиков библиотеки Rust, что отвечает за `std`. Для использования `regex` включите в секцию `[dependencies]` своего файла `Cargo.toml` такую строчку:

```
regex = "0.2.2"
```

Затем поместите артикул `extern crate` в корень своего крейта:

```
extern crate regex;
```

Далее предполагается, что эти изменения внесены.

Основы работы с Regex

Значение типа `Regex` представляет разобранное регулярное выражение, готовое к использованию. Конструктор `Regex::new` пытается разобрать строку `&str` как регулярное выражение и возвращает значение типа `Result`:

```
use regex::Regex;

// Семантический номер версии, например 0.2.1.
// Может содержать суффикс предвыпускной версии, например 0.2.1-alpha.
// (Для краткости суффикс метаданных сборки опущен.)
//
// Обратите внимание на синтаксис простой строки r"...", позволяющий избежать
// круговерти обратных слэшей.
let semver = Regex::new(r"(\d+)\.(\d+)\.(\d+)(-[-.[:alnum:]]*)?")?;

// Простой поиск, возвращающий булево значение.
```

```
let haystack = r#"regex = "0.2.5""#;
assert!(semver.is_match(haystack));
```

Метод `Regex::captures` ищет в строке первое совпадение и возвращает значение типа `regex::Captures`, содержащее информацию о каждой группе:

```
// Мы можем получить запомненные группы:
let captures = semver.captures(haystack)
    .ok_or("должно соответствовать регулярному выражению semver")?;
assert_eq!(&captures[0], "0.2.5");
assert_eq!(&captures[1], "0");
assert_eq!(&captures[2], "2");
assert_eq!(&captures[3], "5");
```

Индексирование значения типа `Captures` паникует, если не было найдено соответствие запрошенной группе. Чтобы проверить, было ли найдено такое соответствие, вызовите метод `Captures::get`, возвращающий `Option<regex::Match>`. Значение типа `Match` представляет соответствие одной группе.

```
assert_eq!(captures.get(4), None);
assert_eq!(captures.get(3).unwrap().start(), 13);
assert_eq!(captures.get(3).unwrap().end(), 14);
assert_eq!(captures.get(3).unwrap().as_str(), "5");
```

Мы можем обойти все найденные в строке соответствия:

```
let haystack = "In the beginning, there was 1.0.0. \
                For a while, we used 1.0.1-beta, \
                but in the end, we settled on 1.2.4.";

let matches: Vec<&str> = semver.find_iter(haystack)
    .map(|match_| match_.as_str())
    .collect();
assert_eq!(matches, vec!["1.0.0", "1.0.1-beta", "1.2.4"]);
```

Итератор `find_iter` порождает значение типа `Match` для каждого соответствия выражению (без перекрытий), от начала до конца строки. Итератор `captures_iter` аналогичен, но порождает значения типа `Captures` для каждой запомненной группы. Поиск замедляется, если необходимо сообщать информацию о группах, поэтому если она вам не нужна, то лучше использовать методы, не возвращающие группы.

Ленивое построение значений типа Regex

Конструктор `Regex::new` может оказаться дорогим удовольствием: создание `Regex` для регулярного выражения длиной 1200 символов может занять половину миллисекунды на быстрой машине для разработки, и даже для тривиальных выражений требуется несколько микросекунд. Поэтому лучше вынести конструирование `Regex` из циклов и делать это только один раз, а затем использовать одно и то же значение многократно.

Крейт `lazy_static` предлагает изящный способ отложить конструирование статических значений до момента, когда они впервые оказываются нужны. Для начала добавьте зависимость в свой файл `Cargo.toml`:

```
[dependencies]
lazy_static = "0.2.8"
```

Этот крейт предоставляет макрос для объявления таких переменных:

```
#[macro_use]
extern crate lazy_static;

lazy_static! {
    static ref SEMVER: Regex
        = Regex::new(r"(\d+)\.(\d+)\.(\d+)(-[-.[:alnum:]]*)?")
            .expect("error parsing regex");
}
```

Макрос расширяется в объявление статической переменной `SEMVER`, но ее тип – не совсем `Regex`. Сгенерированный макросом тип реализует характеристику `Deref<Target=Regex>`, а значит, обладает теми же методами, что `Regex`. При первом разыменовании `SEMVER` вычисляется инициализатор и полученное значение сохраняется на будущее. Поскольку `SEMVER` – статическая, а не локальная переменная, инициализатор работает не более одного раза при каждом запуске программы.

При таком объявлении использовать `SEMVER` просто:

```
use std::io::BufRead;

let stdin = std::io::stdin();
for line in stdin.lock().lines() {
    let line = line?;
    if let Some(match_) = SEMVER.find(&line) {
        println!("{}", match_.as_str());
    }
}
```

Вы можете поместить объявление `lazy_static!` в модуль или даже внутрь функции, в которой используется `Regex`, если это самая подходящая область видимости. Регулярное выражение все равно будет компилироваться только один раз за все время выполнения программы.

Нормализация

Большинство людей считает, что французское слово «thé» (чай) состоит из трех букв. Однако в Юникоде его можно представить двумя способами:

- в *составной* (composed) форме слово «thé» состоит из трех символов: `'t'`, `'h'`, и `'é'`, где `'é'` – один символ Юникода в кодовой позиции `0xe9`;
- в *разложенной* (decomposed) форме слово «thé» состоит из четырех символов: `'t'`, `'h'`, `'e'` и `'\u{301}'`, где `'e'` – обычный символ ASCII без диакритического знака, а кодовая позиция `0x301` – символ «COMBINING ACUTE ACCENT», который добавляет диакритический знак акут к любому предшествующему ему символу.

Ни составная, ни разложенная формы буквы «é» не считаются в Юникоде единственно «правильными», обе они – эквивалентные представления одного и того же абстрактного символа. Юникод специфицирует, что обе формы должны отображаться одинаково, а методам ввода текста разрешено порождать любую форму, поэтому в общем случае пользователи не знают, какую форму видят или вводят. (Rust разрешает употреблять символы Юникода в строковых литералах,

поэтому можно писать просто "thé", если кодировка вас не волнует. Мы будем использовать форму с \u для ясности.)

Однако если рассматривать "th\u{e9}" и "the\u{301}" как значения типа &str или String в Rust, то они совершенно различны. У них разная длина, они не считаются равными при сравнении, у них разные хеш-коды, они по-разному располагаются относительно других строк.

```
assert!("th\u{e9}" != "the\u{301}");
assert!("th\u{e9}" >  "the\u{301}");

// Тип Hasher проектировался для аккумулирования хэшей последовательности
// значений, поэтому хэширование всего одного значения выглядит несколько неуклюже.
use std::hash::{Hash, Hasher};
use std::collections::hash_map::DefaultHasher;
fn hash<T: ?Sized + Hash>(t: &T) -> u64 {
    let mut s = DefaultHasher::new();
    t.hash(&mut s);
    s.finish()
}

// В будущих версиях Rust эти значения могут измениться.
assert_eq!(hash("th\u{e9}"), 0x53e2d0734eb1dff3);
assert_eq!(hash("the\u{301}"), 0x90d837f0a0928144);
```

Понятно, что если мы намереваемся с чем-то сравнивать текст, полученный от пользователя, или использовать его в качестве ключа хеш-таблицы или B-дерева, то нужно сначала привести строку к какой-то канонической форме.

По счастью, в Юникоде специфицировано, что такое «нормализованная» форма строки. Если две строки по правилам Юникода считаются эквивалентными, то их нормализованные формы совпадают с точностью до символа. В кодировке UTF-8 они идентичны с точностью до байта. Это означает, что нормализованные строки можно сравнить с помощью оператора ==, использовать их как ключи HashMap или HashSet и т. д. Теперь вы понимаете, что такое равенство в Юникоде.

Пренебрежение нормализацией может повлечь за собой и нарушение безопасности. Например, если веб-сайт в одних случаях нормализует имена пользователей, а в других – нет, то может оказаться два разных пользователя с именем bananasflambé, которые в одних частях программы считаются одинаковыми, а других различаются, и в результате один пользователь будет незаконно работать с правами другого. Конечно, избежать такой ошибки можно многими способами, но история учит, что есть не меньше способов ее не заметить.

Формы нормализации

В Юникоде определены четыре формы нормализации, каждая из которых рассчитана на определенное применение. Следует ответить на два вопроса:

- во-первых, мы предпочитаем, чтобы символы были настолько *составными*, насколько возможно, или настолько разложенными, насколько возможно? Например, максимально составное представление вьетнамского слова «Phở» – строка из трех символов "Ph\u{1edf}", в которой знак тональности ̉ и знак огласовки ́, примененные к базовому символу «о», составляют один символ Юникода '\u{1edf}', имеющий официальное название LATIN SMALL LETTER O WITH HORN AND HOOK ABOVE.

В максимально разложенном представлении базовая буква и два диакритических знака остаются тремя отдельными символами: `'o'`, `'\u{31b}'` (COMBINING HORN) и `'\u{309}'` (COMBINING HOOK ABOVE), так что получается строка `"Pho\u{31b}\u{309}"`. (Если модифицирующие знаки являются отдельными символами, а не частями составного символа, то для каждой формы нормализации четко специфицирован порядок их следования, так что форма остается однозначной, даже когда с символом связано несколько диакритических знаков.)

Обычно у составной формы меньше проблем с совместимостью, поскольку она больше напоминает представления, использовавшиеся в большинстве языков до появления Юникода. Кроме того, эта форма лучше работает с наивными средствами форматирования типа макроса `format!` в Rust. С другой стороны, разложенная форма лучше подходит для отображения текста и поиска в нем, потому что проясняет детальную структуру текста;

- и второй вопрос: если две последовательности символов содержательно представляют один и тот же текст, но различаются с точки зрения его форматирования, то хотим ли мы рассматривать их как эквивалентные или различные?

  В Юникоде есть разные символы для обычной цифры `'5'`, надстрочного индекса `'⁵'` (или `'\u{2075}'`) и цифры 5 в кружочке `'⑤'` (или `'\u{2464}'`), но все они объявлены «совместимо эквивалентными». Аналогично имеется один символ для лигатуры «ffi» (`'\u{fb03}'`), но он объявлен совместимо эквивалентным последовательности из трех символов `"ffi"`.

  Совместимая эквивалентность имеет смысл при поиске: поиск строки `"difficult"`, содержащей только символы ASCII, должен находить строку `"di\u{fb03}cult"`, в которой встречается лигатура «ffi». Применение совместимой декомпозиции к последней строке должно заменить лигатуру тремя буквами `"ffi"`, что упростит поиск. Но преобразование текста в совместимо эквивалентную форму может привести к потере существенной информации, поэтому к ней не следует относиться беспечно. Например, в большинстве контекстов было бы неправильно хранить `"2⁵"` как `"25"`.

Форма нормализации C (NFC) соответствует максимально составной форме каждого символа, а форма нормализации D (NFD) – максимально разложенной, но ни та, ни другая не пытаются унифицировать совместимо эквивалентные последовательности. Формы нормализации NFKC и NFKD аналогичны NFC и NFD, но нормализуют все совместимо эквивалентные последовательности в некоторый представитель своего класса.

В документе «Модель символов для веба» (Character Model For the World Wide Web) консорциум World Wide Web рекомендует использовать форму NFC для всего контента. В приложении «Синтаксис идентификаторов и образцов в Юникоде» рекомендуется использовать форму NFKC для идентификаторов в языках программирования и предлагаются принципы адаптации этой формы при необходимости.

Крейт unicode-normalization

Крейт `unicode-normalization` предоставляет характеристику, которая добавляет в тип `&str` ряд методов для преобразования текста в любую форму нормализации. Чтобы воспользоваться ими, добавьте следующую строку в секцию `[dependencies]` своего файла `Cargo.toml`:

```
unicode-normalization = "0.1.5"
```

В головном файле вашего крейта должно быть объявление `extern crate`:

```
extern crate unicode_normalization;
```

Если все это сделано, то тип `&str` получает четыре новых метода, которые возвращают итераторы для обхода конкретной формы нормализации строки:

```
use unicode_normalization::UnicodeNormalization;

// Вне зависимости от представления левой строки (одного взгляда недостаточно,
// чтобы определить, какое это представление), справедливы следующие
// утверждения.
assert_eq!("Phở".nfd().collect::<String>(), "Pho\u{31b}\u{309}");
assert_eq!("Phở".nfc().collect::<String>(), "Ph\u{1edf}");

// В левой строке употребляется лигатура "ffi".
assert_eq!("ⓘ Di\u{fb03}culty".nfkc().collect::<String>(), "1 Difficulty");
```

Если взять нормализованную строку и еще раз преобразовать ее в ту же форму нормализации, то гарантированно будет возвращен точно такой же текст.

Любая подстрока нормализованной строки сама нормализована, но конкатенация двух нормализованных строк необязательно будет нормализованной. Например, вторая строка может начинаться модифицирующими символами, которые должны быть помещены перед модифицирующими символами в конце первой строки.

При условии, что в нормализованной форме текста нет неназначенных кодовых позиций, Юникод гарантирует, что его нормализованная форма не изменится в будущих версиях стандарта. Это означает, что нормализованные формы можно безопасно хранить в постоянной памяти, даже если стандарт Юникод будет развиваться.

Глава 18

Ввод и вывод

Дулитл: Какие у тебя есть конкретные доказательства собственного существования?
Бомба 20: Гм… ну… я мыслю, следовательно, я существую.
Дулитл: Это хорошо. Очень хорошо. Но откуда ты знаешь, что существует что-то еще?
Бомба 20: Так говорят мои органы чувств.

— *Темная звезда*

Средства ввода-вывода в стандартной библиотеке Rust организованы вокруг трех характеристик – `Read`, `BufRead` и `Write` – и различных реализующих их типов:

- типы, реализующие `Read`, обладают методами для байтового ввода-вывода. Они называются *читателями*;
- типы, реализующие `BufRead`, называются *буферизованными* читателями. Они поддерживают все методы `Read`, а также методы для чтения строк текста и т. д.;
- типы, реализующие `Write`, поддерживают как байтовый вывод, так и вывод текста в кодировке UTF-8. Они называются *писателями*.

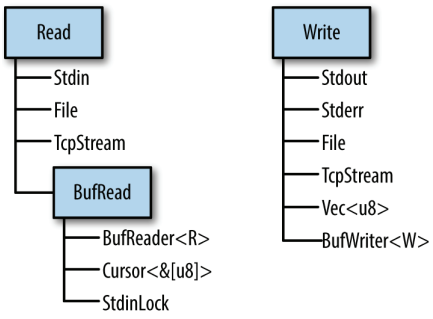


Рис. 18.1 ❖ Некоторые типы читателей и писателей в стандартной библиотеке Rust

В этой главе мы покажем, как использовать эти характеристики и их методы, опишем различные реализующие их типы и расскажем о других способах взаимодействия с файлами, терминалом и сетью.

Читатели и писатели

Читатели – это значения, из которых программа может читать байты, например:

- файлы, открытые методом `std::fs::File::open(filename)`;
- потоки `std::net::TcpStream` для получения данных из сети;
- метод `std::io::stdin()` для чтения из потока стандартного ввода процесса;
- значения `std::io::Cursor<&[u8]>` для «чтения» из байтового массива, который уже находится в памяти.

Писатели – это значения, в которые программа может записывать байты, например:

- файлы, открытые методом `std::fs::File::create(filename)`;
- потоки `std::net::TcpStream` для отправки данных в сеть;
- методы `std::io::stdout()` и `std::io:stderr()` для вывода на терминал;
- значения `std::io::Cursor<&mut [u8]>`, которые позволяют рассматривать любую изменяемую срезку байтов как файл для записи;
- вектор `Vec<u8>` является писателем, методы `write` которого добавляют элементы в конец вектора.

Поскольку для читателей и писателей существуют стандартные характеристики (`std::io::Read` и `std::io::Write`), очень часто пишут универсальный код, работающий с разными каналами ввода-вывода. Например, следующая функция копирует все байты из произвольного читателя в произвольный писатель:

```
use std::io::{self, Read, Write, ErrorKind};

const DEFAULT_BUF_SIZE: usize = 8 * 1024;

pub fn copy<R: ?Sized, W: ?Sized>(reader: &mut R, writer: &mut W)
    -> io::Result<u64>
    where R: Read, W: Write
{
    let mut buf = [0; DEFAULT_BUF_SIZE];
    let mut written = 0;
    loop {
        let len = match reader.read(&mut buf) {
            Ok(0) => return Ok(written),
            Ok(len) => len,
            Err(ref e) if e.kind() == ErrorKind::Interrupted => continue,
            Err(e) => return Err(e),
        };
        writer.write_all(&buf[..len])?;
        written += len as u64;
    }
}
```

Это реализация функции `std::io::copy()` из стандартной библиотеки Rust. В силу универсальности она пригодна для копирования данных из `File` в `TcpStream`, из `Stdin` в вектор `Vec<u8>` в памяти и т. д.

Если код обработки ошибок непонятен, обратитесь к главе 7. Далее мы будем постоянно использовать тип `Result`, поэтому важно хорошо понимать, как он работает.

Все четыре характеристики из модуля `std::io` – `Read`, `BufRead`, `Write` и `Seek` – используются настолько часто, что специально для них заведен отдельный модуль `prelude`:

```
use std::io::prelude::*;
```

Мы пару раз столкнемся с ним в этой главе. Кроме того, возьмите в привычку импортировать сам модуль `std::io`:

```
use std::io::{self, Read, Write, ErrorKind};
```

Здесь ключевое слово `self` объявляет `io` как псевдоним модуля `std::io`. Таким образом, типы `std::io::Result` и `std::io::Error` можно записать короче: `io::Result` и `io::Error`. То же относится и к другим подобным типам.

Читатели

Характеристика `std::io::Read` определяет несколько методов чтения данных. Все они принимают самого читателя по изменяемой ссылке.

- **`reader.read(&mut buffer)`** читает несколько байтов из источника данных и сохраняет их в заданном буфере `buffer`, который имеет тип `&mut [u8]`. Метод читает не более `buffer.len()` байтов.
  Возвращаемое значение имеет тип `io::Result<u64>` – псевдоним типа `Result<u64, io::Error>`. В случае успеха значение типа `u64` содержит количество прочитанных байтов, которое может быть меньше или равно `buffer.len()`, *даже если имеются еще данные* (это зависит от особенностей источника данных). Результат `Ok(0)` означает, что больше читать нечего.
- В случае ошибки метод `.read()` возвращает `Err(err)`, где `err` – значение типа `io::Error`. Значение `io::Error` допускает вывод на печать – для удобства человека, а для программ у него есть метод `.kind()`, который возвращает код ошибки типа `io::ErrorKind`. Элементы этого перечисления имеют имена: `PermissionDenied`, `ConnectionReset` и т. д. По большей части они означают серьезную ошибку, которую нельзя игнорировать, но один вид ошибок нуждается в специальной обработке. Элемент `io::ErrorKind::Interrupted` соответствует коду ошибки `EINTR` в Unix, означающей, что операция чтения была прервана сигналом. Если программа не обрабатывает сигналов каким-то особым образом, то она должна просто повторить чтение. В коде функции `copy()` выше приведен соответствующий пример.
  Как видим, метод `.read()` очень низкоуровневый, он даже отражает особенности базовой операционной системы. Если вы реализуете характеристику `Read` для нового типа источника данных, то это дает большую свободу действий. Но если вы попытаетесь с его помощью читать данные, то придется бороться с трудностями. Поэтому Rust предоставляет несколько методов более высокого уровня. У всех них имеется реализация по умолчанию, написанная в терминах `.read()`. Все они обрабатывают ошибку `ErrorKind::Interrupted`, так что вам этого делать не придется.
- **`reader.read_to_end(&mut byte_vec)`** получает все оставшиеся у читателя данные и добавляет их в конец вектора `byte_vec` типа `Vec<u8>`. Возвращает `io::Result<()>`.

На объем данных, помещаемых этим методом в вектор, нет никаких ограничений, поэтому не пользуйтесь им, если источник данных ненадежен (ниже показано, как наложить ограничение с помощью метода `.take()`).

- **`reader.read_to_string(&mut string)`** – то же самое, но данные дописываются в заданную строку типа `String`. Если поток не содержит корректных данных в кодировке UTF-8, то метод возвращает ошибку `ErrorKind::InvalidData`.
  В некоторых языках байтовый и символьный ввод реализованы разными методами. Но в наши дни кодировка UTF-8 распространена настолько широко, что Rust считает ее стандартом де-факто и поддерживает всюду. Другие кодировки поддерживаются крейтом `encoding` с открытым исходным кодом.
- **`reader.read_exact(&mut buf)`** читает ровно столько данных, чтобы заполнить переданный буфер. Аргумент имеет тип `&[u8]`. Если у читателя заканчиваются данные, прежде чем будет прочитано `buf.len()` байтов, то метод возвращает ошибку `ErrorKind::UnexpectedEof`.

Это основные методы характеристики `Read`. Кроме них, есть еще четыре адаптерных метода, которые принимают `reader` по значению и преобразуют его в итератор или в другой читатель.

- **`reader.bytes()`** возвращает итератор для обхода байтов потока ввода. Тип объекта – `io::Result<u8>`, поэтому для каждого байта следует проверять, не было ли ошибки. К тому же метод вызывает `reader.read()` для каждого байта, поэтому для небуферизованных читателей он крайне неэффективен.
- **`reader.chars()`** делает то же самое, но возвращает символы, т. е. рассматривает ввод как поток в кодировке UTF-8. Если встретится недопустимый символ, то возвращается ошибка `InvalidData`.
- **`reader.chain(reader2)`** возвращает новый читатель, который порождает сначала все данные от `reader`, а потом все данные от `reader2`.
- **`reader.take(n)`** возвращает новый читатель, читающий из того же источника, что `reader`, но не более `n` байтов.

Для закрытия читателя нет никакого метода. Обычно читатели и писатели реализуют характеристику `Drop`, поэтому закрываются автоматически.

Буферизованные читатели

Ради эффективности читатели и писатели могут быть буферизованными. Это означает, что с ними ассоциирована область памяти (буфер), в которой хранится часть входных или выходных данных. Это позволяет уменьшить число системных вызовов, как показано на рис. 18.2. В данном случае приложение читает данные из `BufReader` построчно, вызывая метод `.read_line()`. `BufReader`, в свою очередь, получает от операционной системы блоки большего размера.

Реальный размер буфера `BufReader` по умолчанию составляет несколько килобайтов, поэтому один системный вызов `read` может обслужить сотни вызовов `.read_line()`. Это существенно, потому что системные вызовы работают медленно. (Как видно из рисунка, у операционной системы тоже есть буфер – и по той же причине: системные вызовы медленные, но чтение с диска еще медленнее.)

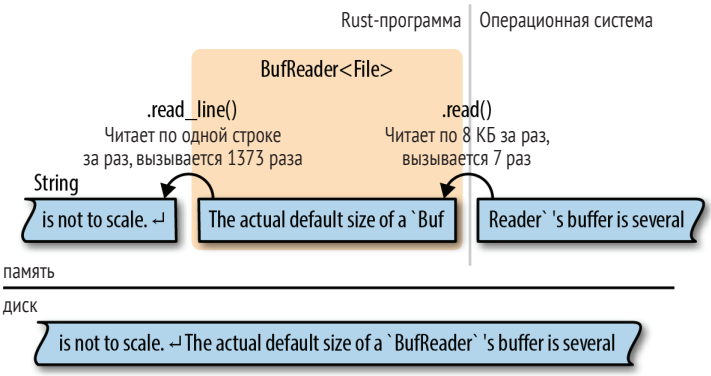


Рис. 18.2 ❖ Буферизованный читатель файла

Буферизованные читатели реализуют характеристику `Read`, а также характеристику `BufRead`, добавляющую следующие методы:

- **`reader.read_line(&mut line)`** читает строку текста и добавляет ее к аргументу `line` типа `String`. Символ новой строки `'\n'`, завершающий строку, включается в `line`. Если входные строки разделяются парой символов `"\r\n"`, как в Windows, то в `line` включаются оба символа.
  Возвращаемое значение имеет тип `io::Result<usize>` и равно количеству прочитанных байтов, включая концы строк, если таковые были.
  Если читатель уже дошел до конца данных, то `line` не изменяется, а метод возвращает `Ok(0)`.
- **`reader.lines()`** возвращает итератор для обхода строк входных данных. Тип объекта – `io::Result<String>`. Символы новой строки *не* включаются. Если строки завершаются парой символов `"\r\n"`, как в Windows, то игнорируются оба символа.
  Этот метод – почти всегда то, что требуется для ввода текста. В следующих двух разделах приведены примеры его использования.
  Методы **`reader.read_until(stop_byte, &mut byte_vec)`** и **`reader.split(stop_byte)`** похожи на `.read_line()` и `.lines()`, но читают байты, а не строки, т. е. порождают вектор `Vec<u8>`. Ограничитель `stop_byte` задаете вы сами.

Характеристика `BufRead` предоставляет также два низкоуровневых метода, `.fill_buf()` и `.consume(n)`, для прямого доступа к внутреннему буферу читателя. Информацию о них см. в документации.

В следующих двух разделах буферизованные читатели рассматриваются подробнее.

Чтение строк

Следующая функция реализует утилиту Unix `grep`. Они ищет заданную строку в строчках текста, которые обычно поступают от другой команды по конвейеру.

```
use std::io;
use std::io::prelude::*;

fn grep(target: &str) -> io::Result<()> {
    let stdin = io::stdin();
```

```
    for line_result in stdin.lock().lines() {
        let line = line_result?;
        if line.contains(target) {
            println!("{}", line);
        }
    }
    Ok(())
}
```

Поскольку мы собираемся вызывать метод `.lines()`, то нужен источник ввода, реализующий `BufRead`. В данном случае мы вызываем `io::stdin()`, чтобы получить данные, поставляемые нам по конвейеру. Однако стандартная библиотека Rust защищает `stdin` мьютексом. Мы вызываем метод `.lock()`, чтобы захватить `stdin` для монопольного использования текущим потоком выполнения; этот метод возвращает значение типа `StdinLock`, реализующее `BufRead`. В конце цикла значение `StdinLock` уничтожается, освобождая мьютекс. (Если бы мьютекса не было, то попытка одновременного чтения `stdin` из двух потоков привела бы к неопределенному поведению. В языке C имеется та же проблема, и решается она точно так же: все стандартные функции ввода-вывода захватывают некую блокировку. Единственное отличие состоит в том, что в Rust эта блокировка является частью API.)

Все остальное тривиально: функция вызывает метод `.lines()` и в цикле обходит объекты, порождаемые итератором. Поскольку итератор порождает значения типа `Result`, мы пользуемся оператором `?` для проверки ошибок.

А теперь попробуем сделать следующий шаг и добавить в программу `grep` поддержку поиска в файлах на диске. Мы можем сделать эту функцию универсальной:

```
fn grep<R>(target: &str, reader: R) -> io::Result<()>
    where R: BufRead
{
    for line_result in reader.lines() {
        let line = line_result?;
        if line.contains(target) {
            println!("{}", line);
        }
    }
    Ok(())
}
```

Теперь ей можно передавать как `StdinLock`, так и буферизованный файл `File`:

```
let stdin = io::stdin();
grep(&target, stdin.lock())?;  // правильно

let f = File::open(file)?;
grep(&target, BufReader::new(f))?;  // тоже правильно
```

Заметим, что `File` не буферизуется автоматически. Тип `File` реализует характеристику `Read`, но не `BufRead`. Однако для файла легко создать буферизованный читатель или любой другой небуферизованный читатель. Это делает метод `BufReader::new(reader)` (если хотите задать размер буфера, пользуйтесь методом `BufReader::with_capacity(size, reader)`).

В большинстве языков файлы буферизуются по умолчанию, а если нужен небуферизованный ввод или вывод, то приходится искать, как отключить буфери-

зацию. В Rust `File` и `BufReader` – два разных библиотечных механизма, поскольку иногда требуются файлы без буферизации, а иногда – буферизация без файлов (например, бывает, что нужно буферизовать ввод из сети).

Ниже приведен полный текст программы, включающий обработку ошибок и разбор аргументов командной строки.

```
// grep - поиск строк, содержащих заданную строку, в stdin или в файле.

use std::error::Error;
use std::io::{self, BufReader};
use std::io::prelude::*;
use std::fs::File;
use std::path::PathBuf;

fn grep<R>(target: &str, reader: R) -> io::Result<()>
    where R: BufRead
{
    for line_result in reader.lines() {
        let line = line_result?;
        if line.contains(target) {
            println!("{}", line);
        }
    }
    Ok(())
}

fn grep_main() -> Result<(), Box<Error>> {
    // Получить аргументы командной строки. Первый аргумент — строка, в которой
    // производится поиск, остальные — имена файлов.
    let mut args = std::env::args().skip(1);
    let target = match args.next() {
        Some(s) => s,
        None => Err("usage: grep PATTERN FILE...")?
    };
    let files: Vec<PathBuf> = args.map(PathBuf::from).collect();

    if files.is_empty() {
        let stdin = io::stdin();
        grep(&target, stdin.lock())?;
    } else {
        for file in files {
            let f = File::open(file)?;
            grep(&target, BufReader::new(f))?;
        }
    }
    Ok(())
}

fn main() {
    let result = grep_main();
    if let Err(err) = result {
        let _ = writeln!(io::stderr(), "{}", err);
    }
}
```

Собирание строк

Некоторые методы читателя, в т. ч. `.lines()`, возвращают итераторы, порождающие значения типа `Result`. Как только вы захотите собрать все прочитанные из файла строки в один большой вектор, возникнет вопрос, как избавиться от `Result`.

```
// правильно, но не то, что мы хотим
let results: Vec<io::Result<String>> = reader.lines().collect();

// ошибка: невозможно преобразовать коллекцию Result в Vec<String>
let lines: Vec<String> = reader.lines().collect();
```

Второй вариант не компилируется, ведь мы проигнорировали потенциальные ошибки. Прямолинейное решение – написать цикл `for`, в котором проверять каждый объект на отсутствие ошибок:

```
let mut lines = vec![];
for line_result in reader.lines() {
    lines.push(line_result?);
}
```

Неплохо, но хорошо было бы как-то использовать здесь метод `.collect()` – и такой способ есть. Нужно только знать, какой тип запрашивать:

```
let lines = reader.lines().collect::<io::Result<Vec<String>>>()?;
```

Как это работает? Стандартная библиотека содержит реализацию характеристики `FromIterator` для типа `Result` (читая документацию, на это легко не обратить внимания), благодаря которой это и возможно:

```
impl<T, E, C> FromIterator<Result<T, E>> for Result<C, E>
    where C: FromIterator<T>
{
    ...
}
```

Здесь говорится: если можно собрать объекты типа `T` в коллекцию типа `C` (`where C: FromIterator<T>`), то можно собрать объекты типа `Result<T, E>` в результат типа `Result<C, E>` (`FromIterator<Result<T, E>> for Result<C, E>`).

Иными словами, `io::Result<Vec<String>>` – тип коллекции, так что метод `.collect()` можно использовать для создания и заполнения значений этого типа.

Писатели

Как мы видели, ввод производится в основном с помощью методов. С выводом всё обстоит несколько иначе. В этой книге мы не раз использовали макрос `println!()` для вывода простого текста.

```
println!("Здравствуй, мир!");

println!("Наибольший общий делитель {:?} равен {}", numbers, d);
```

Существует также макрос `print!()`, который не добавляет символ новой строки в конце. Коды форматирования для `print!()` и `println!()` те же, что для макроса `format!`, они были описаны в разделе «Форматирование значений» главы 17.

Для отправки данных писателю служат макросы `write!()` и `writeln!()`. От `print!()` и `println!()` они отличаются в двух отношениях:

```
writeln!(io::stderr(), "ошибка: миру нельзя сказать здравствуй")?;

writeln!(&mut byte_vec, "Наибольший общий делитель {:?} равен {}", numbers, d)?;
```

Во-первых, оба макроса `write` принимают дополнительный первый аргумент – писатель. Во-вторых, они возвращают `Result`, так что ошибки надлежит обрабатывать. Именно поэтому в конце каждой строки мы добавили оператор `?`.

Макросы `print` не возвращают `Result`, они просто паникуют в случае ошибки записи. Но поскольку они пишут на терминал, такое случается редко.

Характеристика `Write` определяет следующие методы:

- метод **`writer.write(&buf)`** записывает байты из срезки `buf` в поток. Он возвращает значение типа `io::Result<usize>`. В случае успеха оно содержит количество записанных байтов, которое может оказаться меньше `buf.len()`, если так устроен поток.
  Как и `Reader::read()`, это низкоуровневый метод, которым лучше не пользоваться напрямую;
- метод **`writer.write_all(&buf)`** записывает все байты из срезки `buf` и возвращает `Result<()>`;
- **`writer.flush()`** сбрасывает оставшиеся в буфере данные в поток и возвращает `Result<()>`.

Как и читатели, писатели автоматически закрываются в момент уничтожения.

Если `BufReader::new(reader)` наделяет буфером любого читателя, то `BufWriter::new(writer)` наделяет буфером писателя.

```
let file = File::create("tmp.txt")?;
let writer = BufWriter::new(file);
```

Если хотите задать размер буфера, пользуйтесь методом `BufWriter::with_capacity(size, writer)`.

Когда значение типа `BufWriter` уничтожается, все оставшиеся в буфере данные передаются базовому писателю. Но любые возникающие в этот момент ошибки *игнорируются* (поскольку это происходит внутри метода `.drop()` значения `BufWriter`, об ошибке некому сообщить). Чтобы приложение могло заметить все ошибки вывода, явно вызывайте метод `.flush()` буферизованного писателя перед его уничтожением.

Файлы

Мы уже знакомы с двумя способами открытия файлов:

- метод **`File::open(filename)`** открывает существующий файл для чтения. Он возвращает значение типа `io::Result<File>`, содержащее ошибку, если файл не существует;
- метод **`File::create(filename)`** создает новый файл для записи. Если файл с таким именем существует, он усекается до нулевой длины.

Отметим, что тип `File` находится в модуле файловой системы `std::fs`, а не в `std::io`.

Если ни один из этих двух методов не подходит, то можно воспользоваться типом `OpenOptions` и точно задать желаемое поведение:

```
use std::fs::OpenOptions;

let log = OpenOptions::new()
```

```
    .append(true)  // если файл существует, дописывать в конец
    .open("server.log")?;

let file = OpenOptions::new()
    .write(true)
    .create_new(true)  // если файл существует, вернуть ошибку
    .open("new_file.txt")?;
```

Методы `.append()`, `.write()`, `.create_new()` и прочие можно сцеплять, как показано выше, поскольку все они возвращают `self`. Этот паттерн проектирования распространен настолько широко, что в Rust ему присвоено специальное название: «построитель». Другой пример дает тип `std::process::Command`. Дополнительные сведения о типе `OpenOptions` смотрите в документации.

После того как `File` открыт, он ведет себя, как любой другой читатель или писатель. При желании его можно буферизовать. В момент уничтожения значения файл автоматически закрывается.

Поиск

Тип `File` реализует также характеристику `Seek`, т. е. мы можем сразу перейти в произвольное место файла, а не «дочитывать» до него. Эта характеристика определена следующим образом:

```
pub trait Seek {
    fn seek(&mut self, pos: SeekFrom) -> io::Result<u64>;
}

pub enum SeekFrom {
    Start(u64),
    End(i64),
    Current(i64)
}
```

Благодаря перечислению метод `seek` очень выразителен: `file.seek(SeekFrom::Start(0))` означает поиск от начала файла, `file.seek(SeekFrom::Current(-8))` – возврат на несколько байтов и т. д.

Поиск внутри файла работает медленно. При работе с жестким или SSD-диском одна операция поиска занимает столько же времени, сколько чтение нескольких мегабайтов данных.

Другие типы читателей и писателей

Выше в этой главе мы привели несколько примеров типов, отличных от `File`, которые реализуют характеристики `Read` и `Write`. Сейчас мы расскажем о них подробнее:

- **`io::stdin()`** возвращает читатель из стандартного потока ввода типа `io::Stdin`. Поскольку этот поток ввода разделяется всеми потоками выполнения, то при каждой операции чтения захватывается и освобождается мьютекс.
  В типе `Stdin` имеется метод `.lock()`, который захватывает мьютекс и возвращает значение типа `io::StdinLock`, буферизованный читатель, который удерживает этот мьютекс до момента своего уничтожения. Поэтому операции с `StdinLock` уже могут не думать о мьютексе. Пример использования этого метода был приведен в разделе «Чтение строк» выше.

По техническим причинам вызов io::stdin().lock() не работает. Блокировка удерживает ссылку на значение Stdin, а это означает, что значение Stdin должно где-то храниться, чтобы прожить достаточно долго:

```
let stdin = io::stdin();
let lines = stdin.lock().lines(); // ok
```

- методы **io::stdout()** и **io::stderr()** возвращают писателей для стандартных потоков вывода и ошибок. У них тоже есть мьютексы и метод .lock();
- тип **Vec<u8>** реализует характеристику Write. Запись в Vec<u8> приводит к дописыванию новых данных в конец вектора.
  (Однако тип String *не* реализует Write. Чтобы построить строку с помощью Write, сначала запишите данные в Vec<u8>, а затем преобразуйте вектор в строку методом String::from_utf8(vec).);
- метод **Cursor::new(buf)** создает курсор Cursor – буферизованный читатель из buf. Именно так создается читатель для строки String. Аргумент buf может иметь любой тип, реализующий характеристику AsRef<[u8]>, т. е. можно передавать &[u8], &str или Vec<u8>.
  На внутреннем уровне курсор устроен тривиально. В нем есть два поля: сам буфер buf и целое число, равное смещению от начала buf, с которого начинается следующая операция чтения. Первоначально смещение равно 0.
  Курсоры реализуют характеристики Read, BufRead и Seek. Если buf имеет тип &mut [u8] или Vec<u8>, то Cursor реализует также характеристику Write. При записи в курсор перезаписываются байты buf, начиная с текущей позиции. При попытке записи за концом &mut [u8] запись будет осуществлена частично или возникнет ошибка io::Error. Но использование курсора для записи за концом вектора Vec<u8> допускается: при этом увеличивается размер вектора. Таким образом, типы Cursor<&mut [u8]> и Cursor<Vec<u8>> реализуют все четыре характеристики из прелюдии std::io::prelude;
- тип std::net::TcpStream представляет сетевое TCP-соединение. Поскольку протокол TCP двунаправленный, этот тип является одновременно читателем и писателем.
  Статический метод TcpStream::connect(("hostname", PORT)) пытается подключиться к серверу и возвращает значение типа io::Result<TcpStream>;
- тип std::process::Command поддерживает запуск дочернего процесса и передачу ему данных через стандартный ввод:

```
use std::process::{Command, Stdio};

let mut child = Command::new("grep")
    .arg("-e")
    .arg("a.*e.*i.*o.*u")
    .stdin(Stdio::piped())
    .spawn()?;

let mut to_child = child.stdin.take().unwrap();
for word in my_words {
   writeln!(to_child, "{}", word)?;
}
drop(to_child); // закрыть стандартный ввод grep, чтобы она могла завершиться
child.wait()?;
```

Значение `child.stdin` имеет тип `Option<std::process::ChildStdin>`. Выше мы воспользовались методом `.stdin(Stdio::piped())` при настройке дочернего процесса, так что `child.stdin` гарантированно будет инициализирован, когда метод `.spawn()` успешно завершится. Если бы мы этого не сделали, то `child.stdin` было бы равно `None`.

В типе `Command` определены также методы `.stdout()` и `.stderr()`, которые можно использовать для запроса заголовков в `child.stdout` и `child.stderr`.

Модуль `std::io` предлагает также ряд функций, возвращающих тривиальных читателей и писателей:

- **`io::sink()`** – пустой писатель. Все его методы записи возвращают `Ok`, но данные просто отбрасываются;
- **`io::empty()`** – пустой читатель. Чтение из него всегда завершается успешно, но возвращает конец данных;
- **`io::repeat(byte)`** возвращает читателя, который бесконечно возвращает один и тот же байт.

Двоичные данные, сжатие и сериализация

На базе модуля `std::io` построено много крейтов с открытым исходным кодом, предоставляющих дополнительные возможности.

Крейт `byteorder` предоставляет характеристики `ReadBytesExt` и `WriteBytesExt`, добавляющие читателям и писателям методы ввода и вывода двоичных данных.

```
use byteorder::{ReadBytesExt, WriteBytesExt, LittleEndian};

let n = reader.read_u32::<LittleEndian>()?;
writer.write_i64::<LittleEndian>(n as i64)?;
```

Крейт `flate2` предоставляет адаптерные методы для чтения и записи сжатых данных в формате `gzip`:

```
use flate2::FlateReadExt;

let file = File::open("access.log.gz")?;
let mut gzip_reader = file.gz_decode()?;
```

Крейт `serde` содержит средства сериализации и десериализации: он преобразует структуры Rust в массивы байтов и наоборот. Мы уже упоминали его в разделе «Характеристики и сторонние типы» главы 11, а теперь рассмотрим подробнее.

Пусть имеются некоторые данные – карта для текстовой приключенческой игры, – хранящиеся в отображении `HashMap`:

```
type RoomId = String;                       // у каждой комнаты уникальное название
type RoomExits = Vec<(char, RoomId)>;       // ...и список выходов
type RoomMap = HashMap<RoomId, RoomExits>;  // названия комнат и выходы, просто

// Создать простую карту.
let mut map = RoomMap::new();
map.insert("Cobble Crawl".to_string(),
           vec![('W', "Debris Room".to_string())]);
map.insert("Debris Room".to_string(),
           vec![('E', "Cobble Crawl".to_string()),
                ('W', "Sloping Canyon".to_string())]);
...
```

Для преобразования этих данных в формат JSON для вывода достаточно нескольких строчек кода:

```
use std::io;
use serde::Serialize;
use serde_json::Serializer;

let mut serializer = Serializer::new(io::stdout());
map.serialize(&mut serializer)?;
```

Здесь используется метод `serialize` характеристики `serde::Serialize`. Библиотека присоединяет эту характеристику ко всем типам, которые она умеет сериализовывать, а к ним относятся все типы, встречающиеся в наших данных: строки, символы, кортежи, векторы и `HashMap`.

Библиотека `serde` гибкая. В этой программе данные выводятся в формате JSON, поскольку мы выбрали сериализатор `serde_json`. Но есть и другие форматы, например MessagePack. Да и вывод можно было бы направить в файл, в вектор `Vec<u8>` или любому другому писателю. Приведенный выше код печатает данные на `stdout`:

```
{"Debris Room":[["E","Cobble Crawl"],["W","Sloping Canyon"]],"Cobble Crawl":
[["W","Debris Room"]]}
```

`serde` поддерживает также автоматический вывод двух основных характеристик:

```
#[derive(Serialize, Deserialize)]
struct Player {
    location: String,
    items: Vec<String>,
    health: u32
}
```

В версии Rust 1.17 для использования этого атрибута `#[derive]` необходимо несколько дополнительных шагов на этапе настройки проекта. Мы не будем их здесь рассматривать, обратитесь к документации по крейту `serde`. Короче говоря, система сборки автоматически генерирует реализации `serde::Serialize` и `serde::Deserialize` для типа `Player`, так что сериализовать значение `Player` просто:

```
player.serialize(&mut serializer)?;
```

Результат выглядит так:

```
{"location":"Cobble Crawl","items":["a wand"],"health":3}
```

Файлы и каталоги

Далее мы рассмотрим имеющиеся в Rust средства для работы с файлами и каталогами. Они находятся в модулях `std::path` и `std::fs`. Все они подразумевают работу с именами файлов, так что с них и начнем.

Типы OsStr и Path

Как это ни печально, операционная система не требует, чтобы имена файлов были допустимыми с точки зрения Юникода. Следующие две команды оболочки Linux создают текстовые файлы. Но только первое имя – корректная строка UTF-8.

```
$ echo "hello world" > ô.txt
$ echo "O brave new world, that has such filenames in't" > $'\xf4'.txt
```

Обе команды выполняются безо всяких помех, потому что ядро Linux ничего не знает о UTF-8. С точки зрения ядра, любая строка байтов (кроме нуля и символов косой черты) может быть именем файла. И в Windows такая же история: почти любая строка 6-разрядных «широких» символов – допустимое имя файла, даже строки, не являющиеся корректными в кодировке UTF-16. Это относится и к другим строкам, обрабатываемым операционной системой, включая аргументы командной строки и переменные окружения.

В Rust строки – всегда корректные последовательности байтов в Юникоде. Имена файлов на практике *почти* всегда удовлетворяют правилам Юникода, но Rust как-то должен справляться с теми редкими случаями, когда это не так. Потому-то в Rust и имеются типы `std::ffi::OsStr` и `OsString`.

`OsStr` – строковый тип, представляющий надмножество UTF-8. Его цель – служить для представления всех имен файлов, аргументов командной строки и переменных окружения в текущей системе, даже если они недопустимы с точки зрения Юникода. В Unix значение типа `OsStr` может содержать любую последовательность байтов. В Windows такие значения хранятся в кодировке, являющейся расширением UTF-8, которая позволяет закодировать любую последовательность 16-разрядных значений, включая несопоставленные суррогаты.

Итак, у нас есть два строковых типа: `str` – для строк Юникода и `OsStr` – для всякой чепухи, на которую соглашается операционная система. Добавим еще один: `std::path::Path` – для имен файлов. Он введен только для удобства: по составу символов `Path` в точности совпадает с `OsStr`, но добавляет много методов, относящихся к работе с именами файлов, которые мы рассмотрим в следующем разделе. Тип `Path` используется как для абсолютных, так и для относительных имен. Каждая компонента пути представляется типом `OsStr`.

И наконец, каждому строковому типу соответствует «владеющий» тип; `String` владеет выделенным в куче значением типа `str`, `std::ffi::OsString` – выделенным в куче значением типа `OsStr`, а `std::path::PathBuf` – выделенным в куче значением типа `Path`.

	`Str`	`OsStr`	`Path`
Безразмерный тип, передается только по ссылке	да	да	да
Может содержать только текст в Юникоде	да	да	да
Выглядит в точности как UTF-8	да	да	да
Может содержать данные не в Юникоде	нет	да	да
Методы обработки текста	да	нет	нет
Соответствующий владеющий тип динамического размера, выделяемый в куче	`String`	`OsString`	`PathBuf`
Преобразование во владеющий тип	`.to_string()`	`.to_os_string()`	`.to_path_buf()`

Все три типа реализуют общую характеристику `AsRef<Path>`, поэтому нетрудно объявить универсальную функцию, принимающую в качестве аргумента «имя файла любого типа». При этом применяется техника, описанная в разделе «Характеристики AsRef и AsMut» главы 13:

```
use std::path::Path;
use std::io;

fn swizzle_file<P>(path_arg: P) -> io::Result<()>
    where P: AsRef<Path>
{
    let path = path_arg.as_ref();
    ...
}
```

Эта техника используется во всех стандартных функциях и методах, принимающих аргумент `path`, так что любому из них можно передавать строковые литералы.

Методы типов Path и PathBuf

Тип `Path` предлагает, в частности, следующие методы:

- **`Path::new(str)`** преобразует `&str` или `&OsStr` в `&Path`. Строка при этом не копируется: новое значение `&Path` указывает на те же байты, что исходное значение `&str` или `&OsStr`.

  ```
  use std::path::Path;
  let home_dir = Path::new("/home/fwolfe");
  ```

  (Аналогичный метод `OsStr::new(str)` преобразует `&str` в `&OsStr`.);
- **`path.parent()`** возвращает родительский каталог пути, если таковой существует. Возвращаемое значение имеет тип `Option<&Path>`.
  Путь при этом не копируется: родительский каталог `path` всегда является подстрокой `path`.

  ```
  assert_eq!(Path::new("/home/fwolfe/program.txt").parent(),
             Some(Path::new("/home/fwolfe")));
  ```

- **`path.file_name()`** возвращает последнюю компоненту `path`, если таковая существует. Возвращаемое значение имеет тип `Option<&OsStr>`.
  В типичном случае, когда `path` включает каталог, знак косой черты и имя файла, возвращается имя файла.

  ```
  assert_eq!(Path::new("/home/fwolfe/program.txt").file_name(),
             Some(OsStr::new("program.txt")));
  ```

- методы **`path.is_absolute()`** и **`path.is_relative()`** говорят, является ли путь абсолютным, как, например, `/usr/bin/advent` в Unix или `C:\Program Files` в Windows, или относительным, как, например, `src/main.rs`;
- **`path1.join(path2)`** соединяет два пути и возвращает новое значение типа `PathBuf`.

  ```
  let path1 = Path::new("/usr/share/dict");
  assert_eq!(path1.join("words"),
             Path::new("/usr/share/dict/words"));
  ```

  Если путь `path2` абсолютный, то возвращается копия `path2`, так что этот метод можно использовать для преобразования любого пути в абсолютный:

  ```
  let abs_path = std::env::current_dir()?.join(any_path);
  ```

- **path.components()** возвращает итератор для обхода компонент заданного пути слева направо. Тип объекта итератора – `std::path::Component`, перечисление, позволяющее представить различные части имени файла:

```
pub enum Component<'a> {
    Prefix(PrefixComponent<'a>),  // только Windows: буква диска или разделяемая папка
    RootDir,                      // корневой каталог, `/` или `\`
    CurDir,                       // специальный каталог `.`
    ParentDir,                    // специальный каталог `..`
    Normal(&'a OsStr)             // простой файл или каталог
}
```

  Например, в Windows путь `\\venice\Music\A Love Supreme\04-Psalm.mp3` состоит из компоненты `Prefix`, представляющей `\\venice\Music`, за которой следует компонента `RootDir`, а затем две компоненты `Normal`, представляющие `A Love Supreme` и `04-Psalm.mp3`.

Детали смотрите в документации.

Описанные выше методы работают со строками в памяти. Но тип `Path` располагает также методами, которые опрашивают файловую систему: `.exists()`, `.is_file()`, `.is_dir()`, `.read_dir()`, `.canonicalize()` и т. д. Дополнительные сведения смотрите в документации.

Существуют три метода для преобразования `Path` в строку. Все они допускают возможность, что `Path` – недопустимая последовательность в смысле UTF-8:

- **path.to_str()** преобразует `Path` в строку и возвращает значение типа `Option<&str>`. Если `path` – недопустимая последовательность в смысле UTF-8, то возвращается `None`.

```
if let Some(file_str) = path.to_str() {
    println!("{}", file_str);
} // ...иначе пропустить файл со странным именем
```

- **path.to_string_lossy()**, по существу, делает то же самое, но возвращает какую-то строку в любом случае. Если `path` – недопустимая последовательность в смысле UTF-8, то этот метод создает копию, в которой все некорректные последовательности байтов заменены символом U+FFFD (‘**&#xFFFD;**’) (символ замены в Юникоде).

  Возвращаемое значение имеет тип `std::borrow::Cow<str>`, т. е. является заимствованной или принадлежащей владельцу строкой. Чтобы получить из него строку типа `String`, вызовите метод `.to_owned()`.

- **path.display()** предназначен для печати путей.

```
println!("Download found. You put it in: {}", dir_path.display());
```

  Возвращаемое значение – не строка, но реализует характеристику `std::fmt::Display`, так что его можно использовать в макросах `format!()`, `println!()` и т. п. Если путь – недопустимая последовательность в смысле UTF-8, то результат может содержать символ ‘**&#xFFFD;**’.

Функции доступа к файловой системе

В табл. 18.1 перечислены несколько функций из модуля `std::fs` и их приблизительные эквиваленты в Unix и Windows.

Все эти функции возвращают значения типа `io::Result`, содержащие `Result<()>`, если явно не оговорено противное.

Таблица 18.1. Сводка функций доступа к файловой системе

Функция Rust	Unix	Windows
Создание и удаление		
`create_dir(path)`	`mkdir()`	`CreateDirectory()`
`create_dir_all(path)`	Как `mkdir -p`	Как `mkdir`
`remove_dir(path)`	`rmdir()`	`RemoveDirectory()`
`remove_dir_all(path)`	Как `rm -r`	Как `rmdir /s`
`remove_file(path)`	`unlink()`	`DeleteFile()`
Копирование, перемещение, создание ссылки		
`copy(src_path, dest_path) -> Result<u64>`	Как `cp -p`	`CopyFileEx()`
`rename(src_path, dest_path)`	`rename()`	`MoveFileEx()`
`rename(src_path, dest_path)`	`link()`	`CreateHardLink()`
Инспекция		
`canonicalize(path) -> Result<PathBuf>`	`realpath()`	`GetFinalPathNameByHandle()`
`metadata(path) -> Result<Metadata>`	`stat()`	`GetFileInformationByHandle()`
`symlink_metadata(path) -> Result<Metadata>`	`lstat()`	`GetFileInformationByHandle()`
`read_dir(path) -> Result<ReadDir>`	`opendir()`	`FindFirstFile()`
`read_link(path) -> Result<PathBuf>`	`readlink()`	`FSCTL_GET_REPARSE_POINT`
Права		
`set_permissions(path, perm)`	`chmod()`	`SetFileAttributes()`

(Функция `copy()` возвращает размер скопированного файла в байтах. О создании символических ссылок см. раздел «Платформенно-зависимые средства» ниже).

Как видим, Rust изо всех сил пытается предложить переносимые функции, которые предсказуемо работают в Windows, macOS, Linux и других Unix-системах.

Полное описание файловых систем выходит за рамки этой книги, но интересующиеся легко найдут информацию в сети. В следующем разделе мы приведем несколько примеров.

Все эти функции обращаются к операционной системе. Так, `std::fs::canonicalize(path)` не просто манипулирует строкой, исключая `.` и `..` из заданного пути `path`, а разрешает относительные пути с учетом текущего рабочего каталога и следует по символическим ссылкам. Если путь не существует, возвращается ошибка.

Тип `Metadata`, возвращаемый функциями `std::fs::metadata(path)` и `std::fs::symlink_metadata(path)`, содержит такую информацию, как тип и размер файла, права доступа и временные метки. Подробности, как обычно, смотрите в документации.

Для удобства некоторые из этих функций включены в тип `Path` в качестве методов, например `path.metadata()` – то же самое, что `std::fs::metadata(path)`.

Чтение каталогов

Для получения содержания каталога пользуйтесь функцией `std::fs::read_dir` или эквивалентным ей методом `.read_dir()` типа `Path`.

```
for entry_result in path.read_dir()? {
    let entry = entry_result?;
    println!("{}", entry.file_name().to_string_lossy());
}
```

Обратите внимание на два знака `?` в этом фрагменте. В первой строчке проверяются ошибки при открытии каталога, а во второй – при чтении следующего элемента каталога.

Значение `entry` имеет тип `std::fs::DirEntry`, это структура с несколькими методами:

- **`entry.file_name()`** возвращает имя файла или каталога в виде `OsString`;
- **`entry.path()`** делает то же самое, но присоединяет исходный путь, возвращая значение типа `PathBuf`. Если мы запрашиваем содержание каталога `"/home/jimb"` и `entry.file_name()` равно `".emacs"`, то `entry.path()` вернет значение `PathBuf::from("/home/jimb/.emacs")`;
- **`entry.file_type()`** возвращает значение типа `io::Result<FileType>`. В типе `FileType` определены методы `.is_file()`, `.is_dir()` и `.is_symlink()`;
- **`entry.metadata()`** возвращает остальные метаданные элемента каталога.

Специальные каталоги `.` и `..` *не* включаются в содержание каталога.

Ниже приведен более содержательный пример. Программа рекурсивно копирует дерево каталога из одного места в другое.

```
use std::fs;
use std::io;
use std::path::Path;

/// Копирует ссуществующий каталог `src` в `dst`.
fn copy_dir_to(src: &Path, dst: &Path) -> io::Result<()> {
    if !dst.is_dir() {
        fs::create_dir(dst)?;
    }

    for entry_result in src.read_dir()? {
        let entry = entry_result?;
        let file_type = entry.file_type()?;
        copy_to(&entry.path(), &file_type, &dst.join(entry.file_name()))?;
    }

    Ok(())
}
```

Для копирования отдельных элементов каталога вызывается функция `copy_to`.

```
/// Копирует элемент `src` в `dst`.
fn copy_to(src: &Path, src_type: &fs::FileType, dst: &Path) -> io::Result<()> {
    if src_type.is_file() {
        fs::copy(src, dst)?;
    } else if src_type.is_dir() {
        copy_dir_to(src, dst)?;
    } else {
        return Err(io::Error::new(io::ErrorKind::Other,
                                  format!("не знаю, как скопировать: {}",
                                  src.display())));
    }
    Ok(())
}
```

Платформенно-зависимые средства

Написанная выше функция `copy_to` умеет копировать файлы и каталоги. Но допустим, что мы хотим поддержать еще и символические ссылки в Unix.

Не существует переносимого способа создания символических ссылок, который работал бы в Unix и в Windows, но стандартная библиотека предоставляет функцию `symlink` для Unix

```
use std::os::unix::fs::symlink;
```

и с ее помощью наша задача упрощается. Нужно только добавить ветвь в выражение `if` в функции `copy_to`:

```
...
} else if src_type.is_symlink() {
    let target = src.read_link()?;
    symlink(target, dst)?;
...
```

Этот код будет работать, если программа компилируется только для систем на основе Unix, например Linux и macOS.

Модуль `std::os` содержит различные платформенно-зависимые вещи, в т. ч. функцию `symlink`. Тело `std::os` в стандартной библиотеке выглядит как стихотворение:

```
//! Функциональность, зависящая от ОС.

#[cfg(unix)]                    pub mod unix;
#[cfg(windows)]                 pub mod windows;
#[cfg(target_os = "ios")]       pub mod ios;
#[cfg(target_os = "linux")]     pub mod linux;
#[cfg(target_os = "macos")]     pub mod macos;
...
```

Атрибут `#[cfg]` означает условную компиляцию: каждый из этих модулей существует только на некоторых платформах. Поэтому наша модифицированная программа, в которой используется модуль `std::os::unix`, будет успешно компилироваться только для Unix: на других платформах этот модуль не существует.

Если мы хотим, чтобы код компилировался на всех платформах и поддерживал символические ссылки в Unix, то должны использовать директиву `#[cfg]` и в своей программе. В данном случае проще всего импортировать `symlink` на платформе Unix, определив свою заглушку `symlink` для других систем:

```
#[cfg(unix)]
use std::os::unix::fs::symlink;

/// Реализация заглушки `symlink` для платформ, где ее не существует.
#[cfg(not(unix))]
fn symlink<P: AsRef<Path>, Q: AsRef<Path>>(src: P, _dst: Q)
    -> std::io::Result<()>
{
    Err(io::Error::new(io::ErrorKind::Other,
                       format!("не умею копировать символические ссылки: {}",
                               src.as_ref().display())))
}
```

На момент написания этой книги документация на странице https://doc.rust-lang.org/std была сгенерирована путем прогона `rustdoc` для стандартной библиотеки – *в Linux*. Это означает, что системно-зависимая функциональность для macOS, Windows и других платформ в документации не отражена. Самый лучший способ найти ее – воспользоваться командой `rustup doc`, чтобы увидеть документацию для своей платформы в формате HTML. Есть, конечно, и другой вариант – заглянуть в исходный код, доступный по адресу https://github.com/rust-lang/rust/blob/master/src/libstd/sys/windows/ext/fs.rs.

Как выясняется, `symlink` – в каком-то смысле особый случай. Большинство относящихся к Unix возможностей – не автономные функции, а расширяющие характеристики, которые добавляют новые методы в стандартные библиотечные типы. (Расширяющие характеристики рассматриваются в разделе «Характеристики и сторонние типы» главы 11.) Существует модуль `prelude`, который позволяет включить все эти характеристики сразу:

```
use std::os::unix::prelude::*;
```

В Unix при этом появится, например, метод `.mode()` в типе `std::fs::Permissions`, дающий доступ к значению типа `u32`, которое представляет права в Unix. А тип `std::fs::Metadata` будет дополнен акцессорами для полей структуры `struct stat`, например появится метод `.uid()`, возвращающий идентификатор пользователя для владельца файла.

Но в общем и целом модуль `std::os` содержит только базовые средства. Гораздо больше платформенно-зависимой функциональности можно найти в сторонних крейтах, например `winreg` для доступа к реестру Windows.

Средства сетевого программирования

Подробное пособие по сетевому программированию выходит за рамки этой книги. Но если вы уже что-то знаете, то этот раздел поможет вам получить представление о том, как сетевое программирование устроено в Rust.

Для знакомства с низкоуровневым сетевым кодом начните с модуля `std::net`, который предоставляет кросс-платформенную поддержку протоколов TCP и UDP. Крейт `native_tls` включает поддержку SSL/TLS.

Эти модули содержат строительные блоки для прямолинейного блокирующего сетевого ввода-вывода. Простой сервер можно создать, написав всего несколько строчек кода с использованием модуля `std::net`. Программа будет запускать новый поток для обслуживания каждого подключения. Вот, например, код эхо-сервера:

```
use std::net::TcpListener;
use std::io;
use std::thread::spawn;

/// Принимать запросы на подключение и для каждого запускать новый поток.
fn echo_main(addr: &str) -> io::Result<()> {
    let listener = TcpListener::bind(addr)?;
    println!("listening on {}", addr);
    loop {
        // Ждать запрос на подключение от клиента.
        let (mut stream, addr) = listener.accept()?;
```

```
        println!("получен запрос от {}", addr);

        // Запустить поток для обслуживания этого клиента.
        let mut write_stream = stream.try_clone()?;
        spawn(move || {
            // Вернуть все полученное из потока `stream` в него же.
            io::copy(&mut stream, &mut write_stream)
                .expect("ошибка в потоке клиента: ");
            println!("соединение закрыто");
        });
    }
}

fn main() {
    echo_main("127.0.0.1:17007").expect("error: ");
}
```

Эхо-сервер просто повторяет все, что ему отправили. Этот код не сильно отличается от того, что вы написали бы на Java или на Python. (Функцию `std::thread::spawn()` мы рассмотрим в следующей главе.)

Но для высокопроизводительных серверов следует использовать асинхронный ввод-вывод. Необходимая поддержка имеется в крейте `mio`. MIO – код очень низкого уровня. Он предоставляет всего лишь простой цикл обработки событий и методы для асинхронного чтения, записи, подключения и приема запросов на подключение, т. е. асинхронный вариант всего сетевого API. По завершении асинхронной операции MIO передает событие написанному вами методу обработки событий.

Существует также экспериментальный крейт `tokio`, который обертывает цикл обработки событий `mio` слоем API, основанным на будущих объектах, отдаленно похожих на обещания в JavaScript.

Протоколы более высокого уровня поддерживаются сторонними крейтами. Например, крейт `reqwest` предлагает элегантный API для HTTP-клиентов. Вот полный код командной утилиты, которая скачивает произвольный документ по протоколу `http:` или `https:` и выводит его на терминал. При написании кода использовалась версия `reqwest = "0.5.1"`.

```
extern crate reqwest;

use std::error::Error;
use std::io::{self, Write};

fn http_get_main(url: &str) -> Result<(), Box<Error>> {
    // Отправить HTTP-запрос и получить ответ.
    let mut response = reqwest::get(url)?;
    if !response.status().is_success() {
        Err(format!("{}", response.status()))?;
    }

    // Прочитать тело ответа и вывести его на stdout.
    let stdout = io::stdout();
    io::copy(&mut response, &mut stdout.lock())?;

    Ok(())
}
```

```
fn main() {
    let args: Vec<String> = std::env::args().collect();
    if args.len() != 2 {
        writeln!(io::stderr(), "usage: http-get URL").unwrap();
    return;
    }

    if let Err(err) = http_get_main(&args[1]) {
        writeln!(io::stderr(), "error: {}", err).unwrap();
    }
}
```

Каркас `iron` для разработки HTTP-серверов предлагает такие высокоуровневые возможности, как характеристики `BeforeMiddleware` и `AfterMiddleware`, которые помогают разложить приложение на сменные компоненты. Крейт `websocket` реализует протокол WebSocket. И так далее. Rust – молодой язык с активно развивающейся экосистемой с открытым исходным кодом. Поддержка сетевого программирования быстро эволюционирует.

Глава 19

Конкурентность

> Вообще говоря, не рекомендуется писать большие конкурентные программы на машинно-ориентированных языках, допускающих неограниченный доступ к памяти по конкретным адресам. Мы просто не сумеем сделать такие программы надежными (даже с помощью хитроумных аппаратных механизмов).
>
> *— Пер Бринч Хансен (1977)*

> Паттерны коммуникации – это паттерны параллелизма.
>
> *— Уит Моррисс*

Если ваше отношение к конкурентности менялось на протяжении карьеры, то вы не одиноки. Это обычное дело.

Поначалу написание конкурентного кода кажется простым и увлекательным занятием. Инструментарий – потоки, блокировки, очереди и т. д. – просто прелесть: пользуйся в свое удовольствие. Ну, есть, конечно, масса подвохов, но вы-то знаете, где они прячутся, и не делаете ошибок.

Но наступает момент, когда приходится отлаживать чужой многопоточный код, и вы приходите к выводу, что *кое-кому* не стоило бы пользоваться этими инструментами.

А еще через какое-то время вы оказываетесь перед необходимостью отлаживать собственный многопоточный код.

Опыт вселяет здоровый скептицизм, если не откровенно циничное отношение к любому многопоточному коду. И этому способствует случайно попавшаяся на глаза статья, в которой с мельчайшими деталями объясняется, почему, на первый взгляд, правильная идиома многопоточности вообще не работает (это я о «модели памяти»). Но в конце концов вы находите подход к конкурентности, который вроде бы позволяет писать код, не делая постоянно ошибок. В эту идиому можно впихнуть практически что угодно, и вы учитесь говорить «нет» лишней сложности (если урок *действительно* пошел впрок).

Разумеется, есть много различных идиом. Перечислим несколько излюбленных системными программистами подходов:

- *фоновый поток*, у которого есть единственная задача и который периодически просыпается, чтобы ее выполнить;
- *пул рабочих потоков* общего назначения, которые взаимодействуют с клиентами посредством *очереди задач*;

- *конвейеры*, по которым данные передаются от одного потока к другому и каждый поток выполняет небольшую часть работы;
- *распараллеливание по данным*, когда предполагается (обоснованно или нет), что компьютер занят в основном одним большим вычислением, которое разбивается на *n* частей и распределяется между *n* потоками в надежде, что все *n* процессорных ядер будут работать одновременно;
- *море синхронизированных объектов*, когда несколько потоков имеет доступ к одним и тем же данным, а для предотвращения гонок используются специальные схемы *блокировки*, основанные на низкоуровневых примитивах типа мьютексов. (В Java встроена поддержка этой модели, которая была очень популярна в 1990-е и 2000-е годы.);
- *атомарные операции с целыми числами*, когда взаимодействие осуществляется посредством передачи информации в полях размером в одно машинное слово. (Эту схему еще труднее правильно реализовать, чем все остальные, если только передаваемые данные не являются буквально целыми значениями. На практике это обычно указатели.)

Со временем вы, возможно, научитесь применять некоторые из этих подходов и безопасно комбинировать их. Вы станете мастером. И все было бы замечательно, если бы никогда никто, кроме вас, не пытался вносить хотя бы малейшие изменения в систему. В программах, где используются потоки, полно неписаных правил.

Rust предлагает другой подход к конкурентности, он не заставляет придерживаться единого стиля во всех программах (для системных программистов это в любом случае не решение), а безопасно поддерживает разные стили. Неписаные правила становятся писаными – в коде, – и за их соблюдением следит компилятор.

Вы слыхали, что Rust позволяет писать безопасные и быстрые конкурентные программы. А в этой главе мы покажем, как это делается. Мы рассмотрим три способа использования потоков в Rust:

- вилочный параллелизм;
- каналы;
- разделяемое изменяемое состояние.

По ходу дела мы воспользуемся всем, что уже узнали о языке Rust. Внимание, с которым Rust подходит к ссылкам, изменяемости и времени жизни, достаточно важно и в однопоточных программах, но именно в конкурентном программировании раскрывается истинная ценность этих правил. Благодаря им мы можем расширять свой инструментарий, быстро и корректно применять различные стили написания многопоточного кода – без скепсиса, без цинизма, без страха.

Вилочный параллелизм

Простейший случай использования потоков возникает, когда имеется несколько абсолютно независимых задач, которые хотелось бы решать одновременно.

Пусть, например, мы применяем алгоритмы обработки естественного языка к большому корпусу документов. Можно было бы написать такой цикл:

```
fn process_files(filenames: Vec<String>) -> io::Result<()> {
    for document in filenames {
        let text = load(&document)?;  // читать исходный файл
```

```
        let results = process(text);  // вычислить статистические показатели
        save(&document, results)?;    // записать выходной файл
    }
    Ok(())
}
```

Эта программа будет работать, как показано на рис. 19.1.

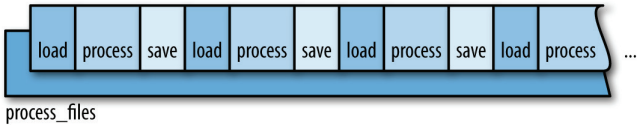


Рис. 19.1 ❖ Однопоточное выполнение функции `process_files()`

Поскольку каждый документ обрабатывается отдельно от других, решение сравнительно легко ускорить, разделив весь корпус на порции и поручив обработку каждой порции отдельному потоку (см. рис. 19.2).

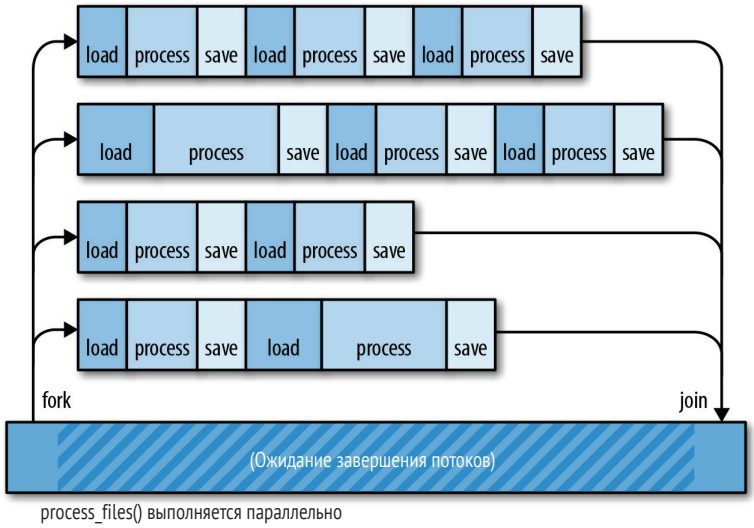


Рис. 19.2 ❖ Многопоточная обработка файлов с применением вилочного параллелизма

Этот подход называется *вилочным параллелизмом* (fork-join parallelism). «Fork» означает разветвление, т. е. запуск нового потока, а «join» – соединение, т. е. ожидание завершения этого потока. Этой техникой мы уже пользовались, чтобы ускорить построение множества Мандельброта в главе 2.

Вилочный параллелизм привлекателен по нескольким причинам:

- он прост до безобразия. Разветвление и соединение легко реализовать, а Rust гарантирует правильность реализации;
- в нем нет узких мест. Никакие разделяемые ресурсы при таком подходе не блокируются. Каждый поток должен ждать только завершения других по-

токов, и это происходит уже после того, как он выполнит свою работу. А до того все потоки работают без помех. В результате накладные расходы на контекстное переключение сводятся к минимуму;

- математика оценки производительности проста. В лучшем случае, запустив четыре потока, мы закончим работу в четыре раза быстрее. На рис. 19.2 показана одна причина, по которой не следует ожидать такого идеального ускорения: не всегда возможно равномерно распределить работу между потоками. Другая причина не испытывать чрезмерного оптимизма – тот факт, что иногда программы с вилочным параллелизмом должны потратить некоторое время после соединения потоков на *объединение* вычисленных ими результатов. Но все равно в любой счетной программе выделение изолированных единиц работы, как правило, дает заметное ускорение;
- о корректности программы легко рассуждать. Программы с вилочным параллелизмом *детерминированы*, коль скоро все потоки действительно изолированы, как счетные потоки в программе построения множества Мандельброта. Это значит, что программа всегда порождает один и тот же результат вне зависимости от вариаций в скорости работы потоков. Это модель конкурентности без гонки.

Основной недостаток вилочного параллелизма заключается в том, что единицы работы должны быть изолированы. Ниже в этой главе мы встретим задачи, в которых такое четкое разделение невозможно.

А пока продолжим пример с обработкой документов на естественном языке. Мы покажем несколько способов применить вилочный параллелизм к функции `process_files`.

Функции spawn и join

Функция `std::thread::spawn` запускает новый поток.

```
spawn(|| {
    println!("привет от дочернего потока");
})
```

Она принимает один аргумент – замыкание или функцию, реализующие характеристику `FnOnce`. Rust запускает новый поток для выполнения кода этого замыкания или функции. Новый поток – это настоящий поток операционной системы со своим стеком, как потоки в C++, C# и Java.

Ниже приведен более содержательный пример, в котором функция `spawn` используется для реализации параллельной версии функции `process_files`.

```
use std::thread::spawn;

fn process_files_in_parallel(filenames: Vec<String>) -> io::Result<()> {
    // Разбить всю работу на несколько порций.
    const NTHREADS: usize = 8;
    let worklists = split_vec_into_chunks(filenames, NTHREADS);

    // Разветвление: запустить поток для обработки каждой порции.
    let mut thread_handles = vec![];
    for worklist in worklists {
        thread_handles.push(
```

```
            spawn(move || process_files(worklist))
        );
    }

    // Соединение: ждать завершения всех потоков.
    for handle in thread_handles {
        handle.join().unwrap()?;
    }

    Ok(())
}
```

Разберем этот код по частям.

```
fn process_files_in_parallel(filenames: Vec<String>) -> io::Result<()> {
```

Новая функция `process_files` имеет такую же сигнатуру, как первоначальная, т. е. они взаимозаменяемы.

```
    // Разбить всю работу на несколько порций.
    const NTHREADS: usize = 8;
    let worklists = split_vec_into_chunks(filenames, NTHREADS);
```

Мы используем вспомогательную функцию `split_vec_into_chunks`, которая здесь не показана, чтобы разбить работу на порции. Результатом является вектор векторов `worklists`. Он содержит восемь срезок исходного вектора `filenames` одинакового размера.

```
    // Разветвление: запустить поток для обработки каждой порции.
    let mut thread_handles = vec![];
    for worklist in worklists {
        thread_handles.push(
            spawn(move || process_files(worklist))
        );
    }
```

Запускаем по одному потоку для каждой порции `worklist`. Функция `spawn()` возвращает значение типа `JoinHandle`, которое нам понадобится позднее. А пока помещаем все описатели `JoinHandle` в вектор.

Обратите внимание, список имен файлов передается рабочему потоку:

- `worklist` определяется и заполняется в цикле `for` в родительском потоке;
- сразу после создания `move`-замыкания ему передается `worklist`;
- затем `spawn` передает замыкание (вместе с вектором `worklist`) новому дочернему потоку.

Эти передачи обходятся дешево. Как и при передаче вектора `Vec<String>`, обсуждавшейся в главе 4, строки `String` не клонируются. На самом деле никакого выделения и освобождения памяти вообще не происходит. Единственное, что физически передается от одного владельца другому, – сама структура `Vec`, т. е. три машинных слова.

Практически всем создаваемым вами потокам нужен какой-то код и данные, чтобы начать работу. Замыкания Rust как раз и содержат и код, и данные.

Идем дальше:

```
    // Соединение: ждать завершения всех потоков.
    for handle in thread_handles {
```

```
        handle.join().unwrap()?;
    }
```

Мы используем метод `.join()` запомненных ранее значений `JoinHandle`, чтобы дождаться завершения всех восьми потоков. Соединение потоков часто необходимо для правильной работы программы, поскольку Rust-программа завершается сразу, как только `main` возвращает управление, даже если еще есть работающие потоки. Деструкторы не вызываются, дополнительные потоки просто снимаются. Если такое поведение вас не устраивает, не забудьте присоединить интересующие вас потоки до возврата из `main`.

Если цикл выполнился до конца, значит, все восемь потоков успешно завершились. Тогда наша функция тоже завершается и возвращает `Ok(())`:

```
    Ok(())
}
```

Обработка ошибок в потоках

Код присоединения дочерних потоков сложнее, чем кажется, из-за необходимости обрабатывать ошибки. Вернемся к следующей строчке:

```
handle.join().unwrap()?;
```

Метод `.join()` делает две вещи.

Во-первых, `handle.join()` возвращает значение типа `std::thread::Result`, которое содержит код ошибки, *если в дочернем потоке возникла паника*. Благодаря этому многопоточность в Rust неизмеримо стабильнее, чем в C++. В C++ выход за границы массива – неопределенное поведение, и вся система никак не защищена от последствий этой ошибки. В Rust паника безопасна и ограничена одним потоком. Границы между потоками служат переборкой для паники, паника не распространяется автоматически из одного потока в другие, зависящие от него, просто уведомление о ней передается другим потокам в виде ошибочного значения `Result`. Программа в целом легко восстанавливается после ошибки.

Но в этой программе мы не пытаемся как-то хитро обрабатывать панику, а сразу вызываем `.unwrap()` для полученного `Result`, утверждая, что он содержит `Ok`, а не `Err`. Если в дочернем потоке имела место паника, то это утверждение окажется неверным, поэтому родительский поток тоже паникует. Таким образом, мы явно распространяем панику от родительских потоков дочернему.

Далее `handle.join()` передает родительскому потоку значение, возвращенное дочерним потоком. Значение, возвращаемое замыканием, которое мы передали функции `spawn`, имеет тип `io::Result<()>`, поскольку именно такой тип возвращает `process_files`. Это возвращенное значение не отбрасывается. Когда дочерний поток завершается, возвращенное им значение сохраняется, и метод `JoinHandle::join()` передает его родительскому потоку.

Полный тип, возвращенный методом `handle.join()` в этой программе, имеет вид `std::thread::Result<std::io::Result<()>>`. Здесь `thread::Result` – часть API `spawn/join`, а `io::Result` – часть нашего приложения.

В нашем случае после развертывания `thread::Result` оператор `?`, применяемый к `io::Result`, явно распространяет ошибки ввода-вывода, возникшие в дочерних потоках, родительскому потоку.

Всё это может показаться довольно запутанным. Но примите во внимание, что речь идет всего об одной строчке кода, и сравните с другими языками. В Java и C# исключения в дочерних потоках по умолчанию печатаются на терминале и игнорируются. В C++ по умолчанию завершается весь процесс. В Rust ошибки – это значения типа `Result` (данные), а не исключения (поток управления). Они передаются между потоками, как любые другие значения. Когда вы пользуетесь низкуровневыми потоковыми API, код обработки ошибок нужно писать очень тщательно, но *раз уж писать его все равно придется*, иметь под рукой `Result` очень полезно.

Разделение неизменяемых данных между потоками

Предположим, что для выполняемого нами анализа нужна большая база английских слов и фраз.

```
// до
fn process_files(filenames: Vec<String>)

// после
fn process_files(filenames: Vec<String>, glossary: &GigabyteMap)
```

Глоссарий `glossary` будет большим, так что передаем его по ссылке. Как изменить функцию `process_files_in_parallel`, чтобы глоссарий передавался рабочим потокам?

Очевидная попытка не работает:

```
fn process_files_in_parallel(filenames: Vec<String>,
                             glossary: &GigabyteMap)
    -> io::Result<()>
{
    ...
    for worklist in worklists {
        thread_handles.push(
            spawn(move || process_files(worklist, glossary))  // ошибка
        );
    }
    ...
}
```

Мы просто добавили в функцию `process_files` аргумент `glossary` и передали его. Но Rust недоволен:

```
error[E0477]: the type `[closure@...]` does not fulfill the required lifetime
  --> concurrency_spawn_lifetimes.rs:35:13
   |
35 |             spawn(move || process_files(worklist, glossary)) // ошибка
   |             ^^^^^
   |
   = note: type must satisfy the static lifetime
```

Компилятор ругается на время жизни замыкания, переданного `spawn`.

Функция `spawn` запускает независимые потоки. Rust ничего не знает о том, сколько времени будет работать дочерний поток, поэтому предполагает худшее: что он может продолжать работать и после того, как родительский поток завер-

шится и все значения в нем будут уничтожены. Очевидно, что если дочерний поток будет существовать так долго, то и исполняемое им замыкание должно жить не меньше. Но у этого замыкания время жизни ограничено: оно зависит от ссылки `glossary`, а ссылки вечно не живут.

Заметим, что Rust абсолютно прав, отвергая этот код! Сейчас функция написана так, что действительно может случиться, что в каком-то потоке возникнет ошибка ввода-вывода, после чего `process_files_in_parallel` вылетит, а остальные потоки будут работать. Дочерние потоки могли бы попытаться использовать глоссарий, после того как главный поток освободил занятую им память. Имеем гонку – а призом в ней будет неопределенное поведение, если победит главный поток. Rust не может этого допустить.

Похоже, функция `spawn` недостаточно определенная, чтобы поддержать разделение ссылок между потоками. Вообще-то, мы уже встречались с такой ситуацией в разделе «Замыкания с кражей» главы 14. Там решение состояло в том, чтобы передать владение данными новому потоку, используя `move`-замыкание. Но здесь оно работать не будет, потому что у нас несколько потоков, и всем нужны одни и те же данные. Безопасная альтернатива – клонировать весь глоссарий для каждого потока, но он слишком большой, так что мы хотели этого избежать. К счастью, стандартная библиотека предлагает другой путь: атомарный подсчет ссылок.

Мы описывали тип `Arc` в разделе «Rc и Arc: совместное владение» главы 4. Самое время приспособить его к делу:

```
use std::sync::Arc;

fn process_files_in_parallel(filenames: Vec<String>,
                             glossary: Arc<GigabyteMap>)
    -> io::Result<()>
{
    ...
    for worklist in worklists {
        // Этот вызов .clone() клонирует только Arc и увеличивает счетчик
        // ссылок на 1. Он не клонирует GigabyteMap.
        let glossary_for_child = glossary.clone();
        thread_handles.push(
            spawn(move || process_files(worklist, &glossary_for_child))
        );
    }
    ...
}
```

Мы изменили тип `glossary`: для параллельного анализа вызывающая сторона должна передать `Arc<GigabyteMap>`, интеллектуальный указатель на `GigabyteMap`, который был перемещен в кучу посредством вызова `Arc::new(giga_map)`.

Вызывая `glossary.clone()`, мы создаем копию интеллектуального указателя `Arc`, но не всего `GigabyteMap`. Это сводится к увеличению счетчика ссылок.

После такого изменения программа компилируется и выполняется без ошибок, но теперь она не зависит от времени жизни ссылок. Если *хотя бы один* поток владеет указателем `Arc<GigabyteMap>`, то глоссарий не будет уничтожен, даже если родительский поток завершится преждевременно. И никакой гонки за данные тоже нет, т. к. данные в `Arc` неизменяемы.

Rayon

Стандартная библиотечная функция `spawn` – важный примитив, но она не рассчитана специально на вилочный параллелизм. Над ней надстроены более удобные API разветвления-соединения. Например, в главе 2 мы использовали библиотеку Crossbeam для разделения работы между 8 потоками. Введенные в Crossbeam «потоки в области видимости» вполне естественно поддерживают вилочный параллелизм.

Другой пример – библиотека Rayon Нико Мацакиса (Niko Matsakis). Она предлагает два способа параллельного выполнения задач:

```
extern crate rayon;
use rayon::prelude::*;

// "делать 2 дела параллельно"
let (v1, v2) = rayon::join(fn1, fn2);

// "делать N дел параллельно"
giant_vector.par_iter().for_each(|value| {
    do_thing_with_value(value);
});
```

Функция `rayon::join(fn1, fn2)` просто вызывает обе функции и возвращает оба результата. А метод `.par_iter()` создает `ParallelIterator` – значение, имеющее методы `map`, `filter` и другие, как у обычного итератора Rust (типа `Iterator`). В обоих случаях Rayon использует собственный пул рабочих потоков, чтобы распределить работу, когда это возможно. Мы просто говорим Rayon, что задачи *можно* выполнять параллельно, а Rayon сам управляет потоками и распределяет работу оптимальным образом.

На рис. 19.3 показаны два способа интерпретации вызова `giant_vector.par_iter().for_each(...)`. (a) Rayon действует так, будто запускает один поток на каждый элемент вектора. (b) За кулисами Rayon запускает по одному рабочему потоку на процессорное ядро, что более эффективно. Этот пул рабочих потоков разделяется всеми потоками программы. Если сразу поступают тысячи задач, Rayon делит работу на части.

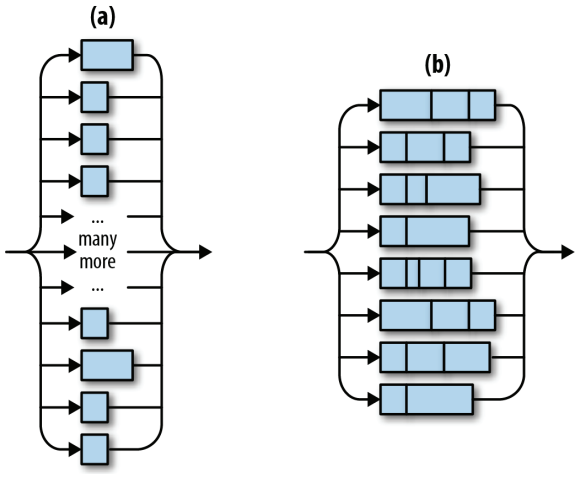


Рис. 19.3 ❖ Rayon в теории и на практике

Ниже приведен вариант функции process_files_in_parallel с использованием Rayon:

```
extern crate rayon;

use rayon::prelude::*;

fn process_files_in_parallel(filenames: Vec<String>, glossary: &GigabyteMap)
    -> io::Result<()>
{
    filenames.par_iter()
        .map(|filename| process_file(filename, glossary))
        .reduce_with(|r1, r2| {
            if r1.is_err() { r1 } else { r2 }
        })
        .unwrap_or(Ok(()))
}
```

Этот код короче и проще, чем версия на основе std::thread::spawn. Разберем его построчно:

- сначала мы вызываем filenames.par_iter(), чтобы создать параллельный итератор;
- с помощью .map() вызываем process_file для каждого имени файла. В результате ParallelIterator обходит последовательность значений io::Result<()>;
- объединяем результаты с помощью .reduce_with(). В данном случае мы оставляем первую ошибку (если ошибки были) и отбрасываем остальные. Если бы мы хотели аккумулировать или напечатать все ошибки, то могли бы сделать это здесь.
  Метод .reduce_with() удобен, когда мы передаем методу .map() замыкание, возвращающее полезное значение в случае успеха. Тогда мы можем передать .reduce_with() замыкание, которое знает, как объединить два успешных результата;
- reduce_with возвращает значение типа Option, равное None, только если вектор filenames был пуст. Мы используем метод .unwrap_or() этого значения, чтобы в этом случае вернуть результат Ok(()).

За кулисами Rayon динамически балансирует нагрузку между потоками, применяя так называемый *перехват работы* (work-stealing). В типичной ситуации он лучше справляется с задачей равномерной загрузки всех процессоров, чем мы могли бы сделать вручную, разбив работу на части заранее, как в разделе «Функции spawn и join» выше.

В качестве бонуса Rayon поддерживает разделение ссылок между потоками. Любая параллельная обработка, производимая за кулисами, гарантированно завершается к моменту возврата из метода reduce_with. Это объясняет, почему мы смогли передать glossary функции process_file, несмотря на то что замыкание вызывается в нескольких потоках. (Кстати говоря, то, что методы названы map и reduce, – не случайность. Модель программирования MapReduce, популяризуемая Google и Apache Hadoop, имеет много общего с вилочным параллелизмом. Ее можно рассматривать как вилочный подход к опросу распределенных данных.)

И снова о множестве Мандельброта

В главе 2 мы применили вилочный параллелизм к отрисовке множества Мандельброта. В результате удалось ускорить отрисовку в четыре раза – результат впечатляющий, но мог бы быть и лучше, принимая во внимание, что программа запускала восемь рабочих потоков и работала на восьмиядерном процессоре.

Проблема в том, что нагрузка распределена неравномерно. Для вычисления одного пикселя изображения нужно выполнить цикл (см. раздел «Что такое множество Мандельброта» главы 2). Оказывается, что бледные части изображения, в которых цикл быстро завершается, обсчитываются гораздо быстрее черных, где выполняются все 255 итераций цикла. Поэтому хотя мы и разбили область на равные горизонтальные полосы, нагрузка на потоки не равна, как видно из рис. 19.4.

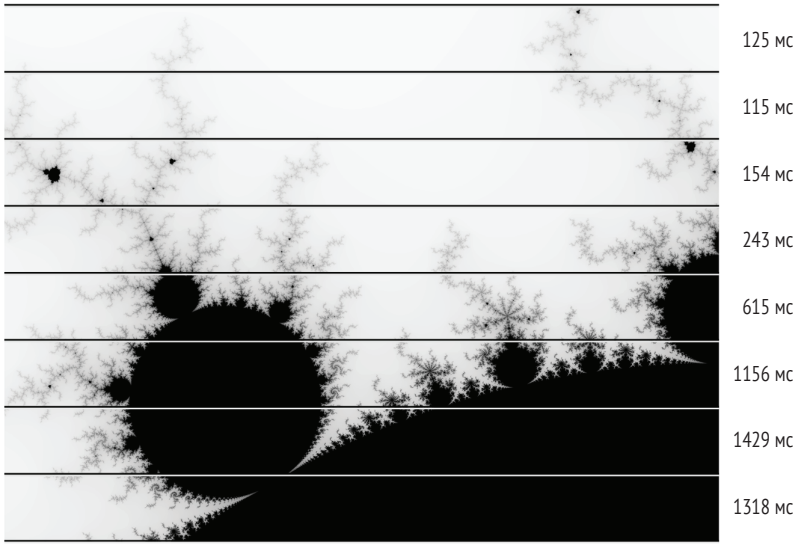


Рис. 19.4 ❖ Неравномерное распределение нагрузки в программе построения множества Мандельброта

Это легко позволяет исправить Rayon. Мы можем запустить параллельную задачу для каждой строки пикселей выходного изображения. При этом будет создано несколько сотен задач, которые Rayon распределит между потоками. Благодаря перехвату работы не имеет значения, что размеры задач разнятся. Rayon будет динамически балансировать работу.

Код приведен ниже. Первая и последняя строчки – часть функции `main`, показанной в разделе «Конкурентная программа рисования множества Мандельброта» главы 2, а находящийся между ними код отрисовки мы изменили.

```
let mut pixels = vec![0; bounds.0 * bounds.1];

// Разбиение `pixels` на горизонтальные полосы.
{
    let bands: Vec<(usize, &mut [u8])> = pixels
        .chunks_mut(bounds.0)
```

```
        .enumerate()
        .collect();

    bands.into_par_iter()
        .weight_max()
        .for_each(|(i, band)| {
            let top = i;
            let band_bounds = (bounds.0, 1);
            let band_upper_left = pixel_to_point(bounds, (0, top),
                                                 upper_left, lower_right);
            let band_lower_right = pixel_to_point(bounds, (bounds.0, top + 1),
                                                  upper_left, lower_right);
            render(band, band_bounds, band_upper_left, band_lower_right);
        });
}

write_bitmap(&args[1], &pixels, bounds).expect("ошибка при записи PNG-файла");
```

Сначала мы создаем `bands`, коллекцию задач, передаваемых Rayon. Каждая задача описывается кортежем `(usize, &mut [u8])`: номер строки пикселей, поскольку он необходим для вычисления, и подлежащая заполнению срезка массива `pixels`. Для разбиения буфера изображения на строки используется метод `chunks_mut`, для присоединения порядкового номера к каждой строке – метод `enumerate`, а для собирания все пар номер-срезка в вектор – метод `collect`. (Нам необходим вектор, потому что Rayon умеет создавать параллельные итераторы только для обхода массивов и векторов.)

Затем мы преобразуем `bands` в параллельный итератор, вызываем метод `.weight_max()`, чтобы сообщить Rayon о том, что эти задачи активно используют процессор, и с помощью метода `.for_each()` говорим, что хотим сделать.

Чтобы можно было использовать Rayon, необходимо включить в `main.rs` такие строчки:

```
extern crate rayon;
use rayon::prelude::*;
```

а в файл `Cargo.toml` – строчки:

```
[dependencies]
rayon = "0.4"
```

После этих изменений программы использует 7.75 ядра на 8-ядерном процессоре. Она стала на 75% быстрее, чем раньше, когда мы разбивали работу на части вручную. А код стал немного короче, что наглядно демонстрирует преимущества использования крейта вместо работы (распределения нагрузки) вручную.

Каналы

Канал – это односторонний тракт передачи данных от одного потока другому. Иными словами, это потокобезопасная очередь.

На рис. 19.5 показано, как используются каналы. Это что-то вроде каналов (pipe) в Unix: с одной стороны данные передаются, с другой принимаются. Обычно концы канала принадлежат разным потокам. Но если в Unix каналы используются для передачи байтов, то в Rust – для передачи значений. Метод `sender.send(item)`

помещает в канал одно значение, а метод `receiver.recv()` одно значение изымает. Владение передается от передающего потока принимающему. Если канал пуст, то `receiver.recv()` блокирует выполнение потока до момента передачи значения.

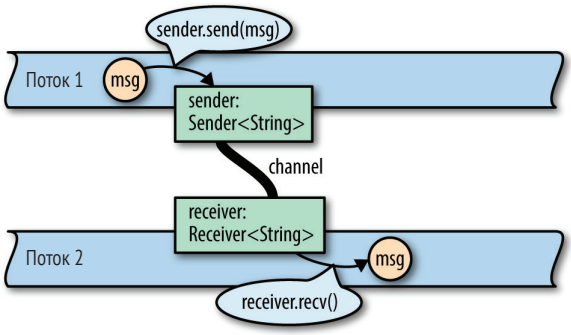


Рис. 19.5 ❖ Канал для передачи строк.
Владение строкой `msg` передается от потока 1 потоку 2

С помощью каналов потоки могут взаимодействовать, передавая значения друг другу. Это очень простой способ совместной работы, не требующей ни блокировки, ни разделяемой памяти.

Эта техника не нова. В языке Erlang изолированные процессы и передача сообщений используются вот уже тридцать лет. Каналам Unix почти пятьдесят лет. Мы привыкли считать, что каналы обеспечивают гибкость и возможность композиции, а не конкурентность, однако на самом деле это им тоже по плечу. На рис. 19.6 показан пример конвейера в Unix. Нет никаких сомнений, что все три программы могут работать одновременно.

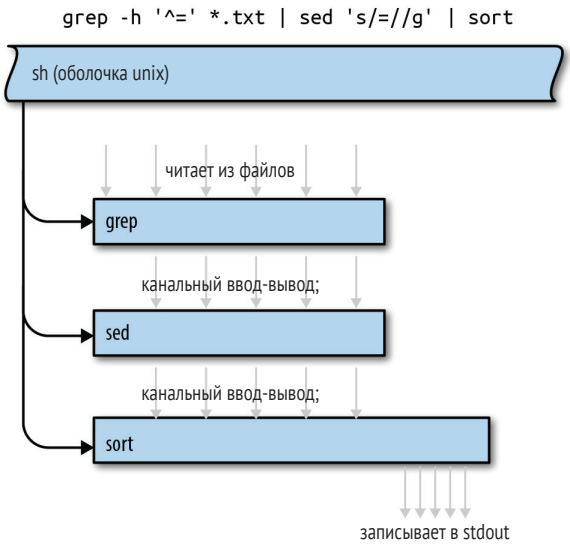


Рис. 19.6 ❖ Выполнение конвейера программ
*grep -h '^=' *.txt | sed 's/=//g' | sort* в Unix

Каналы в Rust работают быстрее, чем в Unix. При отправке значения происходит передача владения, а не копирование, а эта делается быстро, даже когда размер передаваемой структуры измеряется мегабайтами.

Отправка значений

В следующих разделах мы будем использовать каналы для построения конкурентной программы, создающей *инвертированный индекс* – один из основных ингредиентов поисковой системы. Любая поисковая система работает с определенным набором документов. Инвертированный индекс – это база данных, которая говорит, где какое слово встречается.

Мы покажем части кода, имеющие отношение к потокам и каналам. Полный код программы можно найти по адресу https://github.com/ProgrammingRust/fingertips. Он небольшой, примерно тысяча строка на всё про всё.

Наша программа устроена в виде конвейера, показанного на рис. 19.7. Конвейер – лишь один из многих способов применения каналов (другие мы обсудим ниже), но это самый прямой путь сделать конкурентной существующую однопоточную программу.

Всего мы будем использовать пять потоков, решающих различные задачи. Каждый поток на протяжении всей работы программы непрерывно порождает данные. Например, первый поток просто читает исходные документы с диска в память, один за другим. (Мы хотим поручить эту задачу потоку, потому что пишем самый простой код с использованием блокирующих функций `File::open` и `read_to_string`. Мы не хотим, чтобы процессор простаивал, пока работает диск.) На выходе этого этапа получается одна длинная строка, представляющая весь документ, так что этот поток соединен со следующим каналом строк `String`.

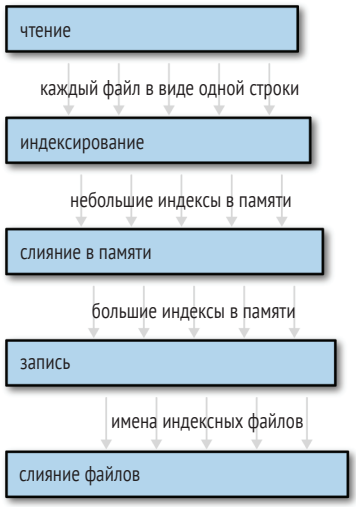


Рис. 19.7. Конвейер построителя индекса.
Стрелки представляют значения,
передаваемые по каналу от одного значения другому.
Дисковый ввод-вывод не показан

Первым делом программа запускает поток, читающий файлы. Обозначим `documents` вектор имен файлов типа `Vec<PathBuf>`. Ниже приведен код, запускающий поток чтения файлов:

```
use std::fs::File;
use std::io::prelude::*;  // для `Read::read_to_string`
use std::thread::spawn;
use std::sync::mpsc::channel;

let (sender, receiver) = channel();

let handle = spawn(move || {
    for filename in documents {
        let mut f = File::open(filename)?;
        let mut text = String::new();
        f.read_to_string(&mut text)?;

        if sender.send(text).is_err() {
            break;
        }
    }
    Ok(())
});
```

Каналы – часть модуля `std::sync::mpsc`. Что означает это имя, мы объясним ниже, но сначала посмотрим, как код работает. Все начинается с создания канала:

```
let (sender, receiver) = channel();
```

Функция `channel` возвращает два значения: отправитель и получатель. Лежащая в основе канала очередь – деталь реализации, которую стандартная библиотека не раскрывает.

Каналы типизированы. Мы собираемся использовать этот канал для отправки текста каждого файла, поэтому `sender` имеет тип `Sender<String>`, а `receiver` – тип `Receiver<String>`. Можно было бы явно запросить канал строк, написав `channel::<String>()`, но мы поручили это механизму вывода типов.

```
let handle = spawn(move || {
```

Как и раньше, для запуска потока используется функция `std::thread::spawn`. Владение `sender` (но не `receiver`) передается новому потоку посредством `move`-замыкания.

Далее просто читаются файлы с диска:

```
    for filename in documents {
        let mut f = File::open(filename)?;
        let mut text = String::new();
        f.read_to_string(&mut text)?;
```

После успешного чтения файла его текст отправляется в канал:

```
        if sender.send(text).is_err() {
            break;
        }
    }
```

Метод `sender.send(text)` передает значение `text` в канал. В конце оно будет передано еще раз – получателю значения. И не важно, какого размера `text` – десять строчек или десять мегабайтов, – в процессе этой операции копируются всего три машинных слова (размер `String`), и соответствующий метод `receiver.recv()` также скопирует три слова.

Методы `send` и `recv` возвращают значение типа `Result`, но неудачно завершиться они могут, только если другая сторона канала уничтожена. Вызов `send` завершается неудачно, если был уничтожен получатель `Receiver`, потому что в противном случае значение осталось бы в канале навечно, ведь в отсутствие получателя его некому было бы получить. Аналогично вызов `recv` завершается неудачно, если был уничтожен отправитель `Sender`, потому что иначе `recv` ждал бы вечно, т. к. в отсутствие отправителя никакой поток не сможет отправить следующего значения. Уничтожение своей стороны каналы – обычный способ «повесить трубку», т. е. закрыть ставшее ненужным соединение.

В нашей программе `sender.send(text)` завершается неудачно, только если поток получателя преждевременно завершился. Это типичная ситуация в коде, где используются каналы. Не важно, произошло это специально или вследствие ошибки, поток читателя может без суеты завершиться сам.

Если случилась такая неприятность или если поток закончил читать документы, он возвращает `Ok(())`:

```
    Ok(())
});
```

Отметим, что это замыкание возвращает `Result`. Если в потоке возникает ошибка ввода-вывода, то он завершается немедленно, а ошибка сохраняется в значении `JoinHandle` этого потока.

Разумеется, как и любой язык программирования, Rust допускает много других стратегий обработки ошибок. Мы могли бы просто напечатать сообщение об ошибке с помощью `println!` и перейти к обработке следующего файла. Или можно было бы передавать ошибки по тому же каналу, что и данные, сделав его каналом значений типа `Result`. А можно было бы создать второй канал специально для ошибок. Выбранный нами подход экономный и в то же время ответственный: оператор `?` позволяет избежать нагромождения трафаретного кода или даже явных блоков `try/catch`, встречающихся в Java, но тем не менее ошибки не «проглатываются» молча.

Для удобства мы обернули весь приведенный выше код функцией, которая возвращает `receiver` (мы его еще не использовали) и `JoinHandle` нового потока:

```
fn start_file_reader_thread(documents: Vec<PathBuf>)
    -> (Receiver<String>, JoinHandle<io::Result<()>>)
{
    let (sender, receiver) = channel();

    let handle = spawn(move || {
        ...
    });

    (receiver, handle)
}
```

Заметим, что эта функция запускает новый поток и сразу же возвращает управление. Подобные функции мы напишем для каждого этапа конвейера.

Получение значений

Теперь у нас имеется поток, в котором работает цикл отправки значений. Мы можем запустить второй поток, где в цикле вызывается метод `receiver.recv()`:

```
while let Ok(text) = receiver.recv() {
    do_something_with(text);
}
```

Но значения `Receiver` итерируемые, поэтому красивее будет написать так:

```
for text in receiver {
    do_something_with(text);
}
```

Оба цикла эквивалентны. В любом случае, если канал окажется пуст, когда управление дойдет до начала цикла, то принимающий поток будет блокирован, до тех пор пока какой-нибудь другой поток не отправит значение. Цикл завершается нормально, если канал пуст и отправитель `Sender` уничтожен. В нашей программе это происходит естественно, когда поток-читатель завершается. Этот поток выполняет замыкание, владеющее переменной `sender`; когда замыкание завершается, `sender` уничтожается.

Теперь можно написать код второго этапа конвейера.

```
fn start_file_indexing_thread(texts: Receiver<String>)
    -> (Receiver<InMemoryIndex>, JoinHandle<()>)
{
    let (sender, receiver) = channel();

    let handle = spawn(move || {
        for (doc_id, text) in texts.into_iter().enumerate() {
            let index = InMemoryIndex::from_single_document(doc_id, text);
            if sender.send(index).is_err() {
                break;
            }
        }
    });
    (receiver, handle)
}
```

Эта функция запускает поток, который получает значения типа `String` из одного канала (`texts`) и отправляет значения типа `InMemoryIndex` в другой канал (`sender`/`receiver`). Задача этого потока – взять каждый файл, загруженный на первом этапе, и преобразовать документ в небольшой инвертированный индекс в памяти, относящийся только к одному файлу.

Главный цикл этого потока прямолинеен. Вся работа по индексированию документа выполняется в функции `make_single_file_index`. Мы не приводим здесь ее исходный код, но смысл его в том, чтобы разбить входную строку на слова, а затем создать отображение, сопоставляющее каждому слову список его позиций в документе.

На этом этапе нет никакого ввода-вывода, поэтому и ошибок `io::Error` быть не может. Так что функция возвращает не `io::Result<()>`, а `()`.

Выполнение конвейера

Остальные три этапа похожи на описанные выше. Каждый потребляет значение `Receiver`, созданное на предыдущем этапе. Цель оставшейся части конвейера – слить все небольшие индексы в один большой файл на диске. Самый быстрый способ, который мы сумели отыскать, состоит из трех этапов. Не станем приводить код целиком (он есть в сети), а покажем лишь сигнатуры соответствующих функций.

Сначала сливаем индексы в памяти, пока размер получившегося индекса не станет слишком велик (этап 3).

```
fn start_in_memory_merge_thread(file_indexes: Receiver<InMemoryIndex>)
    -> (Receiver<InMemoryIndex>, JoinHandle<()>)
```

Эти большие индексы записываем на диск (этап 4).

```
fn start_index_writer_thread(big_indexes: Receiver<InMemoryIndex>,
                             output_dir: &Path)
    -> (Receiver<PathBuf>, JoinHandle<io::Result<()>>)
```

Наконец, если образовалось несколько больших файлов, то мы сливаем их, применяя алгоритм слияния файлов (этап 5).

```
fn merge_index_files(files: Receiver<PathBuf>, output_dir: &Path)
    -> io::Result<()>
```

Последний этап не возвращает `Receiver`, поскольку это уже конец цепочки, а просто создает один выходной файл на диске. Он не возвращает `JoinHandle`, т. к. на этом этапе нет смысла запускать новый поток. Вся работа выполняется в вызывающем потоке.

Вот мы и дошли до кода, в котором запускаются потоки и проверяются ошибки.

```
fn run_pipeline(documents: Vec<PathBuf>, output_dir: PathBuf)
    -> io::Result<()>
{
    // Запустить все пять этапов конвейера.
    let (texts,   h1) = start_file_reader_thread(documents);
    let (pints,   h2) = start_file_indexing_thread(texts);
    let (gallons, h3) = start_in_memory_merge_thread(pints);
    let (files,   h4) = start_index_writer_thread(gallons, &output_dir);
    let result = merge_index_files(files, &output_dir);

    // Ждем завершения потоков и сохраняем ошибки, о которых они сообщили.
    let r1 = h1.join().unwrap();
    h2.join().unwrap();
    h3.join().unwrap();
    let r4 = h4.join().unwrap();

    // Возвращаем первую встретившуюся ошибку, если таковые были.
    // (На самом деле h2 и h3 не могут завершиться с ошибкой: эти потоки
    // заняты обработкой данных только в памяти.)
```

```
    r1?;
    r4?;
    result
}
```

Как и раньше, для явного распространения паники от дочерних потоков родительскому используется метод `.join().unwrap()`. И есть только одна необычная вещь – вместо того чтобы использовать оператор `?` сразу же, мы сохраняем значения типа `io::Result`, до тех пор пока все четыре потока не соединятся.

Этот конвейер на 40% быстрее эквивалентной однопоточной программы. Неплохо для работы на полдня, но просто пустяк по сравнению с ускорением на 675%, которого мы добились в случае программы отрисовки множества Мандельброта. Очевидно, что мы не загрузили на 100% ни оборудования ввода-вывода, ни даже все процессорные ядра. В чем же дело?

Конвейеры подобны сборочным линиям на производстве: их производительность ограничена производительностью самого медленного этапа. Совершенно новая, ненастроенная сборочная линия может оказаться не быстрее штучного изготовления, но сборочные линии благодарно отзываются на целенаправленную настройку. В нашем случае измерения показывают, что узкое место – второй этап. В потоке индексирования используются функции `.to_lowercase()` и `.is_alphanumeric()`, а значит, он тратит много времени на поиск в таблицах Юникода. Следующие за ним потоки большую часть времени спят в вызове `Receiver::recv`, ожидая входных данных.

Следовательно, мы могли бы добиться большего. Расшивая узкие места, мы увеличиваем степень параллелизма. Теперь, когда мы знаем, как работают каналы, а наша программа состоит из изолированных кусков кода, легко понять, как устранить это первое бутылочное горлышко. Можно было бы вручную оптимизировать код второго этапа, как и любой код, или разбить работу на большее число этапов, или просто запустить одновременно несколько потоков индексирования файлов.

Возможности и производительность каналов

Часть `mpsc` имени `std::sync::mpsc` означает *multi-producer, single-consumer* (несколько производителей, один потребитель). Это краткое описание того вида взаимодействия, который реализуют каналы Rust.

В нашей простой программе каналы переносят значения от одного отправителя одному получателю. Это довольно распространенный случай. Но каналы Rust поддерживают и несколько отправителей, и это может пригодиться, например, когда один поток обрабатывает запросы от нескольких клиентских потоков, как показано на рис. 19.8.

Тип `Sender<T>` реализует характеристику `Clone`. Чтобы получить канал с несколькими отправителями, нужно просто создать обычный канал и клонировать отправителя столько раз, сколько захочется. Значения от разных отправителей можно передавать в разные потоки.

Клонировать значение типа `Receiver<T>` нельзя, поэтому если требуется, чтобы несколько потоков получало значения из одного канала, то понадобится `Mutex`. Как это сделать, мы покажем ниже в этой главе.

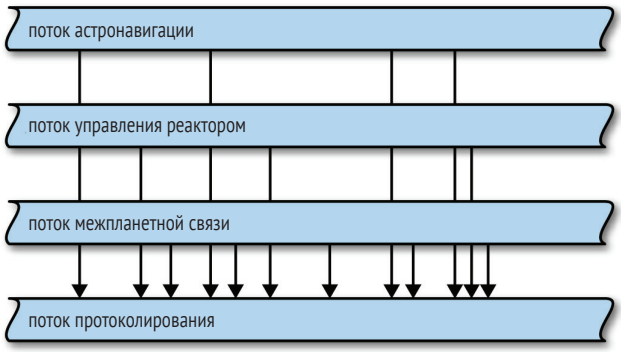


Рис. 19.8 ❖ Один канал может получать запросы от нескольких отправителей

Каналы Rust тщательно оптимизированы. В момент создания канала Rust использует специальную «одноразовую» реализацию очереди. Если вы отправите по каналу только один объект, то накладные расходы будут минимальны. При отправке второго значения Rust переключается на другую реализацию очереди, готовя канал к длительной работе для передачи большого числа значений с минимальными издержками на выделение памяти. А если `Sender` клонируется, Rust вынужден перейти на третью реализацию, обеспечивающую безопасную работу в условиях, когда несколько потоков пытаются одновременно отправлять значения. Но даже в самой медленной реализации используется очередь без блокировок, так что отправка и получение значения в худшем случае включают несколько атомарных операций и выделение из кучи плюс саму передачу значения. Системные вызовы необходимы только в случае, когда очередь пуста, и, следовательно, принимающий поток должен перевести себя в состояние сна. Но тогда трафик по каналу в любом случае не достигает максимума.

Несмотря на всю эту оптимизацию, есть одна ошибка, относящаяся к производительности канала, которую очень легко допустить: отправка значений быстрее, чем их можно получить и обработать. В результате накапливается никогда не рассасывающаяся очередь на входе в канал. Например, мы обнаружили, что в нашей программе поток чтения файлов (этап 1) может загружать файлы гораздо быстрее, чем поток на этапе 2 способен их индексировать. А в итоге очередь сразу же забивается сотнями мегабайтов исходных данных.

Такое поведение приводит к перерасходу памяти и негативно отражается на локальности обращений. Хуже того, отправляющий поток продолжает работать, потребляя процессор и другие системные ресурсы, чтобы отправлять все новые и новые значения как раз тогда, когда эти ресурсы остро необходимы принимающей стороне.

И тут Rust снова подражает каналам Unix. В Unix применяется элегантный трюк для поддержания противодавления (backpressure) в канале, чтобы принудить быстрых отправителей к замедлению: у каждого канала в Unix размер фиксирован, и если процесс пытается писать в канал, который в данный момент заполнен, то система попросту блокирует этот процесс до тех пор, пока в канале не освободится место. Эквивалентный механизм в Rust называется *синхронным каналом*.

```
use std::sync::mpsc::sync_channel;

let (sender, receiver) = sync_channel(1000);
```

Синхронный канал похож на обычный во всем, кроме одного нюанса: при его создании указывается, сколько значений он может содержать. Для синхронного канала вызов `sender.send(value)` – потенциально блокирующая операция. Ведь идея в том и состоит, что блокирование не всегда плохо. В нашей программе замена `channel` в функции `start_file_reader_thread` на `sync_channel` с лимитом в 32 значения сокращает потребление памяти на две трети (на нашем эталонном наборе данных) без снижения производительности.

Потокобезопасность: Send и Sync

До сих мы вели себя так, будто любые значения можно беспрепятственно передавать и разделять между потоками. В большинстве случаев это верно, но вся поддержка потокобезопасности в Rust зиждется на двух встроенных характеристиках: `std::marker::Send` and `std::marker::Sync`:

- типы, реализующие `Send`, безопасно передавать в другой поток по значению. Владение ими может переходить от потока к потоку;
- типы, реализующие `Sync`, безопасно передавать в другой поток по неизменяемой ссылке. Их можно разделять между потоками.

Слово «безопасно» здесь означает то же, что и везде: отсутствие гонки за данные и других видов неопределенного поведения.

Так, в функции `process_files_in_parallel` мы использовали замыкание для передачи вектора `Vec<String>` из родительского потока в каждый дочерний. Тогда мы не заостряли на этом внимания, но это означает, что память для вектора и хранящихся в нем строк выделена в родительском потоке, а освобождена в дочернем. Это допустимо только благодаря тому, то `Vec<String>` реализует характеристику `Send`: распределитель памяти, которым внутри себя пользуются типы `Vec` и `String`, потокобезопасен.

(Если бы вы захотели написать собственные типы `Vec` и `String` с быстрыми, но не потокобезопасными распределителями памяти, то должны были бы сделать это с помощью типов, не реализующих характеристику `Send`, например небезопасных указателей. Тогда Rust вывел бы, что ваши типы `NonThreadSafeVec` и `NonThreadSafeString` не реализуют `Send`, и ограничил бы их использование только однопоточными программами. Но это редкий случай.)

Как видно по рис. 19.9, оба типа реализуют `Send` и `Sync`. Чтобы реализовать эти характеристики в структурах и перечислениях, не понадобится даже атрибут `#[derive]`. Rust все сделает за вас. Структура или перечисление реализует `Send`, если все ее поля реализуют `Send`, и то же самое справедливо в отношении `Sync`.

Немногие типы, которые не реализуют `Send` и `Sync`, – это в основном те, в которых изменяемость используется непотокобезопасным способом. Например, рассмотрим тип `std::rc::Rc<T>` интеллектуального указателя с подсчетом ссылок.

Что случилось бы, если бы было разрешено разделять `Rc<String>` между потоками? Если бы два потока попытались одновременно клонировать `Rc`, как показано на рис. 19.10, то возникла бы гонка за данные, поскольку оба увеличивают разделяемый счетчик ссылок. В результате счетчик ссылок мог бы оказаться неправильным, что впоследствии привело бы к использованию после освобождения или к двойному освобождению, т. е. в любом случае к неопределенному поведению.

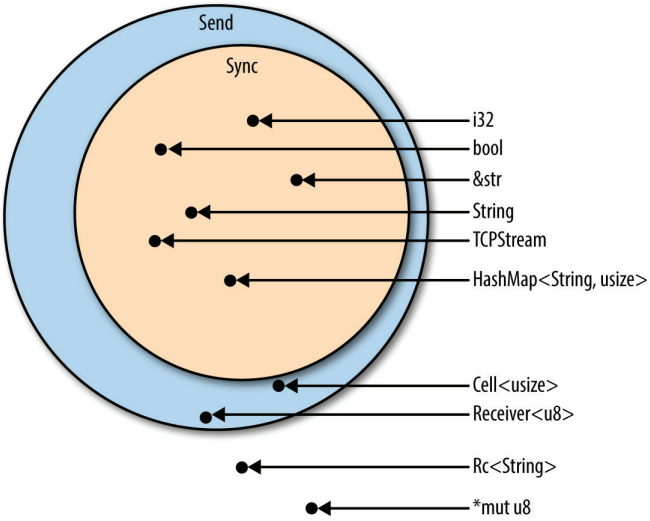


Рис. 19.9 ❖ Типы, реализующие `Send` и `Sync`

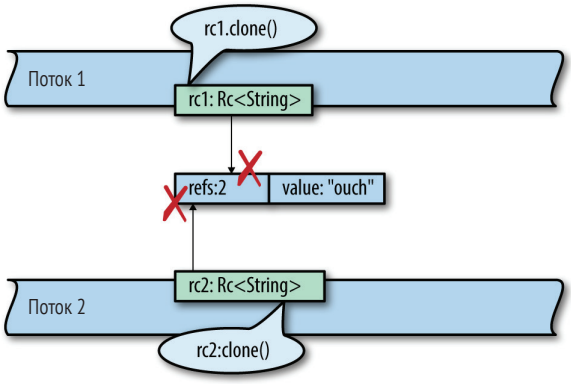


Рис. 19.10 ❖ Почему `Rc<String>` не реализует ни `Sync`, ни `Send`

Конечно же, Rust это предотвращает. Вот код, ведущий к такой гонке за данные:

```
use std::thread::spawn;
use std::rc::Rc;

fn main() {
    let rc1 = Rc::new("hello threads".to_string());
    let rc2 = rc1.clone();
    spawn(move || {  // ошибка
        rc2.clone();
    });
    rc1.clone();
}
```

Rust отказывается его компилировать, выдавая такое сообщение об ошибке:

```
error[E0277]: the trait bound `Rc<String>: std::marker::Send` is not satisfied
              in `[closure@...]`
  --> concurrency_send_rc.rs:10:5
   |
10 |     spawn(move || { // error
   |     ^^^^^ within `[closure@...]`, the trait `std::marker::Send` is not
   |           implemented for `Rc<String>`
   |
   = note: `Rc<String>` cannot be sent between threads safely
   = note: required because it appears within the type `[closure@...]`
   = note: required by `std::thread::spawn`
```

Теперь мы видим, как `Send` и `Sync` помогают Rust обеспечивать потокобезопасность. Они выступают в роли ограничений в сигнатуре функций типа `spawn`, которые передают данные через границы потоков. При запуске потока функцией `spawn` переданное ей замыкание должно реализовывать `Send`, а это означает, что все значения внутри него должны реализовывать `Send`. Аналогично любые значения, передаваемые по каналу в другой поток, должны реализовывать `Send`.

Отправка объектов почти любого итератора по каналу

Наш построитель инвертированного индекса выполнен в виде конвейера. Код вполне прозрачен, но нам пришлось настраивать каналы и запускать потоки вручную. С другой стороны, итераторные конвейеры, которые мы строили в главе 15, как-то ухитряются втиснуть куда больше работы в несколько строчек кода. Нельзя ли построить нечто подобное для потоковых конвейеров?

Вообще, было бы хорошо унифицировать итераторные и потоковые конвейеры. Тогда построитель индекса можно было бы записать в виде итераторного конвейера. Начинаться он мог бы как-то так:

```
documents.into_iter()
    .map(read_whole_file)
    .errors_to(error_sender)   // отфильтровать ошибочные результаты
    .off_thread()              // запустить поток, выполняющий предшествующую работу
    .map(make_single_file_index)
    .off_thread()              // запустить другой поток для этапа 2
    ...
```

Характеристики позволяют добавлять методы в стандартные библиотечные типы, так что в этом нет ничего невозможного. Для начала напишем характеристику с объявлением нужного нам метода:

```
use std::sync::mpsc;

pub trait OffThreadExt: Iterator {
    /// Преобразует этот итератор в итератор с запуском потока: вызов
    /// `next()` выполняется в отдельном рабочем потоке, так что итератор
    /// и тело вашего цикла работают параллельно.
    fn off_thread(self) -> mpsc::IntoIter<Self::Item>;
}
```

Затем реализуем эту характеристику для итераторных типов. Удачно, что тип `mpsc::Receiver` уже является итерируемым.

```
use std::thread::spawn;

impl<T> OffThreadExt for T
    where T: Iterator + Send + 'static,
          T::Item: Send + 'static
{
    fn off_thread(self) -> mpsc::IntoIter<Self::Item> {
        // Создать канал для передачи объектов из рабочего потока.
        let (sender, receiver) = mpsc::sync_channel(1024);

        // Передать этот итератор новому рабочему потоку и выполнять его там.
        spawn(move || {
            for item in self {
                if sender.send(item).is_err() {
                    break;
                }
            }
        });

        // Вернуть итератор, который вытягивает значения из канала.
        receiver.into_iter()
    }
}
```

Как должна выглядеть фраза `where` в этом коде, мы определили с помощью процедуры, очень похожей на описанную в разделе «Обратное конструирование ограничений» главы 11. Сначала у нас было только

```
impl<T: Iterator> OffThreadExt for T
```

То есть мы хотели, чтобы реализация работала для всех итераторов. Rust об этом даже слышать не желал. Поскольку мы используем `spawn` для передачи итератора типа `T` новому потоку, то должны задать `T: Iterator + Send + 'static`. Поскольку мы отправляем объекты по каналу, то должно быть `T::Item: Send + 'static`. Этими изменениями Rust остался удовлетворен.

В этом весь характер Rust: мы вправе добавить поддержку конкурентности почти для любого итератора – но не раньше, чем поймем и документируем ограничения, при которых его использование безопасно.

За пределами конвейеров

В примерах из этого раздела мы использовали конвейеры, потому что это изящное и очевидное применение каналов. Конвейеры всем понятны – материальные, практичные и детерминированные.

Но каналы полезны не только для организации конвейеров. Это еще и быстрый и легкий способ предоставить асинхронный сервис другим потокам в том же процессе.

Предположим, к примеру, что мы хотим вести протоколирование в отдельном потоке, как на рис. 19.8. Другие потоки могли бы посылать сообщения протоколирующему потоку по каналу, а поскольку мы можем клонировать отправителя `Sender`, то в нескольких клиентских потоках могут присутствовать отправители, доставляющие сообщения одному протоколирующему потоку.

Выполнение сервиса наподобие протоколирования в отдельном потоке имеет свои преимущества. Протоколирующий поток может в нужный момент ротировать файлы журналов. Ему не нужна хитроумная координация с другими потоками. Другие потоки не будут блокироваться. Сообщения спокойно накапливаются в канале в ожидании момента, когда протоколирующий поток сможет их обработать.

Каналы можно использовать и в случаях, когда один поток отправляет запрос другому и хочет получить какой-то ответ. Запрос первого потока может представлять собой структуру или кортеж, включающий поле типа `Sender`, – своего рода конверт с адресом, который второй поток использует для отправки ответа. Это не означает, что взаимодействие должно быть синхронным. Первый поток сам решает, следует ли блокировать выполнение и дождаться ответа или воспользоваться методом `.try_recv()` для опроса.

Описанных выше инструментов – вилочного параллелизма для вычислений с высокой степенью параллелизма, каналов для организации слабо связанных компонентов – достаточно для разработки широкого спектра приложений. Но это еще не все.

Разделяемое изменяемое состояние

За месяцы, прошедшие с момента прочтения главы 8, ваша программа моделирования папоротников обрела новую жизнь. Вы уже работаете над многопользовательской стратегической игрой реального времени, в которой восемь игроков соревнуются в выращивании самых аутентичных папоротников в смоделированном ландшафте Юрского периода. В качестве сервера игры выступает массово-параллельное приложение, в котором обработкой запросов занято много потоков. Как эти потоки координируются, чтобы начать игру, как только соберутся восемь игроков?

Проблема заключается в том, что имеется несколько потоков, которым нужен доступ к разделяемому списку игроков, желающих присоединиться к игре. Эти данные по необходимости должны быть изменяемыми и разделяемыми между всеми потоками. Но ведь в Rust не бывает разделяемого изменяемого состояния. Так что же делать?

Для решения задачи можно было бы создать новый поток с единственной целью: управлять этим списком. Другие потоки взаимодействовали бы с ним посредством каналов. Разумеется, при этом придется понести накладные расходы на поддержку дополнительного потока операционной системой.

Другой вариант – воспользоваться средствами, которые Rust предоставляет для безопасного разделения изменяемых данных. Это низкоуровневые примитивы, знакомые любому системному программисту, имевшему дело с потоками. В этом разделе мы рассмотрим мьютексы, блокировки чтения-записи, условные переменные и атомарные целые числа. И напоследок покажем, как реализовать в Rust глобальные изменяемые переменные.

Что такое мьютекс?

Мьютекс (или «блокировка») применяется для того, чтобы принудить несколько потоков соблюдать очередность при доступе к определенным данным.

С мьютексами Rust мы познакомимся в следующем разделе. А сначала имеет смысл вспомнить, как выглядят мьютексы в других языках. Вот пример простого использования мьютекса в C++:

```
// Код на C++, в не на Rust
void FernEngine::JoinWaitingList(PlayerId player) {
    mutex.Acquire();

    waitingList.push_back(player);

    // Начать игру, если собралось достаточно игроков.
    if (waitingList.length() >= GAME_SIZE) {
        vector<PlayerId> players;
        waitingList.swap(players);
        StartGame(players);
    }

    mutex.Release();
}
```

Вызовы `mutex.Acquire()` и `mutex.Release()` отмечают начало и конец *критической секции*. Для каждого мьютекса в программе в каждый момент времени только один поток может войти в защищаемую им критическую секцию. Если какой-то поток уже находится в критической секции, то вызов `mutex.Acquire()` в другом потоке блокирует его выполнение, до тех пор пока первый поток не выполнит `mutex.Release()`.

Мы говорим, что мьютекс *защищает* данные: в данном случае `mutex` защищает `waitingList`. Но следить за тем, что любой поток захватывает мьютекс перед доступом к данным и освобождает его по завершении работы с ними, должен программист.

Мьютексы полезны по несколькими причинам:

- они предотвращают *гонку за данные*, т. е. ситуацию, когда несколько потоков одновременно читает и записывает одну и ту же область памяти. Гонка за данные считается неопределенным поведением в C++ и в Go. Управляемые языки типа Java и C# обещают, что аварийного завершения не будет, но результат гонки за данные все равно не определен;
- даже если бы гонок за данные не было, даже если бы все операции чтения и записи выполнялись строго поочередно, все равно в отсутствие мьютекса действия различных потоков могли бы чередоваться в произвольном порядке. Представьте себе попытку написать код, который работает, даже если другие потоки изменяют его данные. И представьте отладку такого кода. Выглядит это так, будто вашу программу кто-то преследует;
- мьютексы поддерживают программирование с *инвариантами* – утверждениями о защищенных данных, которые по построению верны в начальный момент и поддерживаются в каждой критической секции.

Конечно, на самом деле всё это – одна и та же причина: неконтролируемые состояния гонки делают программирование невозможным. Мьютексы вносят некоторый порядок в этот хаос (хотя и меньший, чем в случае каналов или вилочного параллелизма).

Однако в большинстве языков с мьютексами очень легко запутаться. В C++, как и во многих других языках, данные и блокировки – это разные объекты. В лучшем

случае можно надеяться на комментарий, который объясняет, что каждый поток должен захватить мьютекс, прежде чем обращаться к данным:

```
class FernEmpireApp {
...

private:
    // Список игроков, желающих вступить в игру. Защищен мьютексом `mutex`.
    vector<PlayerId> waitingList;

    // Мьютекс, который следует захватить перед чтением или изменением `waitingList`.
    Mutex mutex;

    ...
};
```

Но как бы ни были хороши комментарии, компилятор не может гарантировать безопасности доступа. Если какая-то часть программы забудет захватить мьютекс, то мы получим неопределенное поведение, а на практике – ошибки, которые чрезвычайно трудно воспроизвести и исправить.

Даже в Java, где имеется некоторая теоретическая ассоциация между объектами и мьютексами, она не слишком глубока. Компилятор не делает никаких попыток навязать ее, а на практике редко бывает так, что данные, защищенные мьютексом, в точности совпадают с полями ассоциированного объекта. Часто защищать необходимо данные в нескольких объектах. Так что правила блокировки остаются хитроумными, и комментарии – по-прежнему основное средство их соблюдения.

Мьютексы в Rust

Теперь мы покажем реализацию списка ожидающих игроков в Rust. На сервере нашей игры Fern Empire каждому игроку присваивается уникальный идентификатор:

```
type PlayerId = u32;
```

Список ожидания – это просто коллекция игроков:

```
const GAME_SIZE: usize = 8;

/// Размер списка ожидания никогда не бывает больше GAME_SIZE.
type WaitingList = Vec<PlayerId>;
```

Список ожидания хранится в поле структуры `FernEmpireApp` – синглтона, который записывается в указатель типа `Arc` на этапе инициализации сервера. В каждом потоке имеется `Arc`-указатель на него. Эта структура содержит всю разделяемую конфигурацию и прочие вещи, необходимые программе. Большая часть ее полей постоянна. Но список ожидания разделяемый и изменяемый, поэтому должен быть защищен мьютексом:

```
use std::sync::Mutex;

/// Все потоки имеют общий доступ к этой большой структуре, содержащей контекст.
struct FernEmpireApp {
    ...
    waiting_list: Mutex<WaitingList>,
    ...
}
```

В отличие от C++, в Rust защищаемые данные хранятся *внутри* мьютекса. Инициализация мьютекса выглядит так:

```
let app = Arc::new(FernEmpireApp {
    ...
    waiting_list: Mutex::new(vec![]),
    ...
});
```

Создание нового `Mutex` похоже на создание `Box` или `Arc`, но если `Box` и `Arc` требуют выделения памяти в куче, то `Mutex` связан только с блокировкой. Если мы хотим, чтобы `Mutex` был выделен в куче, то надо об этом прямо сказать. Именно так мы здесь и поступили, использовав `Arc::new` для всего приложения, а `Mutex::new` – только для защищенных данных. Эти типы часто встречаются вместе: `Arc` удобен для разделения значений между потоками, а `Mutex` – для защиты изменяемых данных, совместно используемых несколькими потоками.

Теперь мы можем реализовать метод `join_waiting_list`, в котором этот мьютекс используется.

```
impl FernEmpireApp {
    /// Добавить игрока в список ожидания на следующую игру.
    /// Начать игру сразу, как только наберется достаточно игроков.
    fn join_waiting_list(&self, player: PlayerId) {
        // Захватить мьютекс и получить доступ к хранящимся внутри него данным.
        // Область видимости `guard` - критическая секция.
        let mut guard = self.waiting_list.lock().unwrap();

        // Дальше идет логика самой игры.
        guard.push(player);
        if guard.len() == GAME_SIZE {
            let players = guard.split_off(0);
            self.start_game(players);
        }
    }
}
```

Единственный способ добраться до данных – вызвать метод `.lock()`:

```
let mut guard = self.waiting_list.lock().unwrap();
```

Метод `self.waiting_list.lock()` блокирует поток, пока не будет захвачен мьютекс. Значение типа `MutexGuard<WaitingList>`, возвращенное этим методом, – тонкая обертка вокруг `&mut WaitingList`. Благодаря Deref-преобразованиям мы можем вызывать методы `WaitingList` прямо от имени стража `guard`:

```
guard.push(player);
```

Страж даже позволяет заимствовать прямые ссылки на обернутые им данные. Механизм времени жизни в Rust гарантирует, что эти ссылки не будут жить дольше самого стража. Не существует никакого способа получить доступ к данным внутри `Mutex`, предварительно не захватив его.

В момент уничтожения `guard` блокировка снимается. Обычно это происходит в конце блока, но можно уничтожить стража и вручную:

```
if guard.len() == GAME_SIZE {
    let players = guard.split_off(0);
    drop(guard);  // не удерживать блокировку списка в процессе запуска игры
    self.start_game(players);
}
```

mut и Mutex

Может показаться странным – нам, по крайней мере, поначалу так и показалось, – что метод `join_waiting_list` не принимает `self` по изменяемой ссылке. Вот его сигнатура:

```
fn join_waiting_list(&self, player: PlayerId)
```

Ведь обернутая коллекция `Vec<PlayerId>` действительно требует, чтобы ее методу `push` передавалась изменяемая ссылка. Его сигнатура такова:

```
pub fn push(&mut self, item: T)
```

Тем не менее код компилируется и работает. В чем же дело?

В Rust ключевое слово `mut` означает «монопольный доступ». А отсутствие `mut` означает «разделяемый доступ».

Мы привыкли, что типы передают `mut`-доступ от родителя к потомку, от контейнера к его элементам. Мы ожидаем, что сможем вызвать `mut`-методы значения `starships[id].engine`, только если изначально имеем изменяемую ссылку на `starships` (или владеем `starships` – тогда вы Илон Маск[1], с чем вас и поздравляем). Это режим по умолчанию, потому что если у нас нет монопольного доступа к родителю, то у Rust, вообще говоря, нет способа проверить наличие монопольного доступа к потомку.

Но у `Mutex` такой способ есть: блокировка. На самом деле мьютекс именно для этого и предназначен: предоставить *монопольный* (`mut`) доступ к хранящимся внутри него данным, несмотря на наличие *разделяемого* (не-`mut`) доступа к самому мьютексу со стороны нескольких потоков.

Система типов Rust говорит нам, что делает `Mutex`. Он динамически обеспечивает монопольный доступ – обычно это делается статически компилятором Rust.

(Вспомните, что тип `std::cell::RefCell` делает то же самое, только без поддержки нескольких потоков. И `Mutex`, и `RefCell` – вариации на тему внутренней изменяемости (см. главу 9)).

Почему мьютексы – не всегда хорошая идея

Прежде чем заняться мьютексами, мы представили несколько подходов к конкурентности, которые покажутся до странности простыми любому, кто раньше работал на C++. Это не случайно: эти подходы специально проектировались так, чтобы дать строгие гарантии защиты от наиболее сомнительных аспектов конкурентного программирования. Программы, в которых используется только вилочный параллелизм, детерминированы и защищены от взаимоблокировок. Программы на основе каналов обладают почти таким же корректным поведением. Если канал

[1] Starship – космический корабль. Их разработкой и занимается компания Илона Маска. – *Прим. перев.*

используется исключительно для организации конвейера, как в нашем построителе индекса, то программа детерминирована: время доставки сообщения может меняться, но на результате это не отражается. И так далее. Иметь гарантии поведения многопоточной программы здорово!

Дизайн типа `Mutex` в Rust почти наверняка заставит вас работать с мьютексами более систематически и разумно, чем раньше. Но имеет смысл сделать паузу и подумать о том, когда гарантии безопасности Rust могут помочь, а когда они бессильны.

В безопасном Rust-коде не может возникнуть *гонка за данные* – ошибка, при которой несколько потоков одновременно читают и записывают одну и ту же область памяти, что приводит к бессмысленным результатам. Это замечательно, поскольку гонка за данные всегда является ошибкой, и в реальных многопоточных программах они встречаются нередко.

Однако потоки, в которых используются мьютексы, подвержены и другим проблемам, которых Rust исправить не может:

- в допустимой Rust-программе не бывает гонок за данные, но могут встретиться другие *состояния гонки*— ситуации, когда поведение программы зависит от хронометража потоков и потому может меняться от выполнения к выполнению. Некоторые состояния гонки безобидны. А некоторые проявляются как общая нестабильность и ошибки, исправить которые невероятно трудно. Хаотическое использование мьютексов – прямой путь к состояниям гонки. А забота об их безобидности – ваша задача;
- разделяемое изменяемое состояние влияет и на структуру программы. Если каналы играют роль границ абстракций, так что программу легко разбить на изолированные компоненты, которые можно тестировать по отдельности, то мьютексы поощряют дизайн по принципу «просто добавь метод», а это может привести к монолитной глыбе взаимосвязанного кода:
- наконец, мьютексы не так просты, как кажется на первый взгляд, что станет понятно в следующих двух разделах.

Все эти проблемы – неотъемлемое свойство инструментария. Применяйте более структурированный подход, когда можете; пользуйтесь мьютексами, когда вынуждены.

Взаимоблокировка

Поток может заблокировать сам себя, когда пытается захватить блокировку, которую уже удерживает.

```
let mut guard1 = self.waiting_list.lock().unwrap();
let mut guard2 = self.waiting_list.lock().unwrap();  // взаимоблокировка
```

Предположим, что первый вызов `self.waiting_list.lock()` завершился успешно, т. е. блокировка захвачена. Второй вызов обнаруживает, что блокировка кем-то захвачена, и ждет ее освобождения. Ждать он будет вечно, поскольку именно ожидающий поток и удерживает блокировку.

Иными словами, блокировка в `Mutex` не является рекурсивной.

В данном случае ошибка очевидна. Но в реальной программе вызовы `lock()` могут находиться в двух разных методах, один из которых вызывает другой. Код каждого метода по отдельности выглядит нормально. Есть и другие способы на-

рваться на взаимоблокировку: когда имеется несколько потоков, пытающихся одновременно захватить несколько мьютексов. Система заимствования Rust не может защитить от взаимоблокировок. Лучшая защита – делать критические секции как можно меньше: вошел, сделал свою работу, вышел.

Взаимоблокировка возможна и при работе с каналами. Например, два потока могут взаимно заблокироваться, если будут ждать получения сообщений друг от друга. Чтобы такого не случалось, необходимо правильно проектировать программу. В конвейере, пример которого был приведен в программе построения инвертированного индекса, поток данных ациклический. В подобной программе, как и в конвейере оболочки Unix, взаимоблокировка невозможна.

Отравленные мьютексы

Метод `Mutex::lock()` возвращает `Result` по той же причине, что и метод `JoinHandle::join()`: чтобы обеспечить корректное поведение, если в другом потоке возникла паника. Запись `handle.join().unwrap()` означает, что Rust должен распространить панику из одного потока в другой. Идиома `mutex.lock().unwrap()` аналогична.

Если поток паникует, удерживая мьютекс, то Rust помечает значение `Mutex` как *отравленное*. Любая последующая попытка захватить отравленный мьютекс методом `lock` вернет ошибочный результат. Вызов `.unwrap()` говорит Rust, что при таком развитии событий нужно паниковать, распространив тем самым панику из другого потока в данный.

Плохо ли, когда возникает отравленный мьютекс? Само слово «отравленный» навевает мысли о смерти, но этот сценарий необязательно фатальный. В главе 7 мы уже говорили, что паника безопасна. Даже если один поток запаникует, остальная часть программа будет находиться в безопасном состоянии.

Таким образом, причина отравления мьютекса – не опасение неопределенного поведения. Скорее, она связана с технологией программирования с инвариантами. Раз программа запаниковала и вышла из критической секции, не закончив своих дел, то, возможно, она обновила одни поля защищенной структуры, не обновив других, что могло привести к нарушению инвариантов. Rust отравляет мьютекс, чтобы другие потоки не смогли по ошибке продолжить работу в некорректной ситуации, только усугубив ее. Но даже когда мьютекс отравлен, вы *можете* его захватить и получить доступ к внутренним данным с полной гарантией монопольности, см. документацию по методу `PoisonError::into_inner()`. Однако делать это вы будете не по ошибке, а с открытыми глазами.

Каналы с несколькими производителями и мьютексом

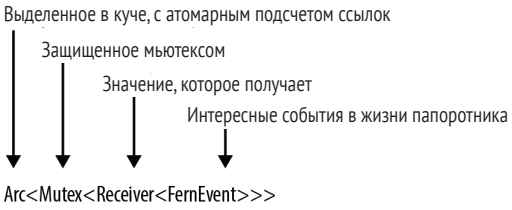
Выше мы отмечали, что каналы в Rust могут иметь несколько производителей и одного потребителя. Конкретнее, у канала может быть только один получатель `Receiver`. Невозможно организовать пул потоков, в котором несколько потоков работает с одним `mpsc`-каналом как с разделяемым списком работ.

Однако существует простой обходной путь, в котором не используется ничего, кроме компонентов из стандартной библиотеки. Мы можем обернуть `Receiver` мьютексом и сделать его разделяемым. Вот модуль, в котором реализована эта идея:

```
pub mod shared_channel {
    use std::sync::{Arc, Mutex};
```

```
    use std::sync::mpsc::{channel, Sender, Receiver};

    /// Потокобезопасная обертка вокруг `Receiver`.
    #[derive(Clone)]
    pub struct SharedReceiver<T>(Arc<Mutex<Receiver<T>>>);

    impl<T> Iterator for SharedReceiver<T> {
        type Item = T;

        /// Получает следующий объект от обернутого получателя.
        fn next(&mut self) -> Option<T> {
            let guard = self.0.lock().unwrap();
            guard.recv().ok()
        }
    }

    /// Создает новый канал, получатель которого может разделяться между потоками.
    /// Возвращает отправителя и получателя, как стандартная функция
    /// `channel()`, и иногда может быть подставлена вместо нее.
    pub fn shared_channel<T>() -> (Sender<T>, SharedReceiver<T>) {
        let (sender, receiver) = channel();
        (sender, SharedReceiver(Arc::new(Mutex::new(receiver))))
    }
}
```

Мы воспользовались типом `Arc<Mutex<Receiver<T>>>`. Универсальные типы громоздятся один на другой. Это случается в Rust чаще, чем в C++. Может показаться, что из-за этого можно запутаться, но зачастую, как в данном случае, простое зачитывание вслух имен раскрывает смысл всей конструкции:



Блокировки чтения-записи (RwLock)

Теперь перейдем от мьютексов к другим средствам синхронизации потоков, имеющимся в стандартном библиотечном модуле `std::sync`. Мы будем двигаться быстро, потому что полное обсуждение этих средств выходит за рамки книги.

В серверных программах часто бывают конфигурационные параметры, которые загружаются один раз и потом редко изменяются. Большинство потоков только запрашивает эту информацию, но поскольку конфигурационные параметры все же *могут* изменяться (например, сервер можно попросить перезагрузить их с диска), то они должны быть защищены какой-то блокировкой. В таких случаях мьютекс можно использовать, но он создает ненужное узкое место. Потокам ни к чему ждать своей очереди для опроса конфигурационных данных, если они не изменяются. Это тот случай, когда может пригодиться *блокировка чтения-записи*, или `RwLock`.

Если у мьютекса имеется всего один метод `lock`, то у блокировки чтения-записи два метода: `read` и `write`. Метод `RwLock::write` работает как `Mutex::lock`. Он ждет получения монопольного доступа к защищенным данным, чтобы затем можно было их изменить. Метод `RwLock::read` предоставляет неизменяющий доступ, а его преимущество состоит в том, что ждать, скорее всего, не придется, потому что несколько потоков может безопасно читать данные одновременно. В случае мьютекса в любой момент времени у защищенных данных может быть только один читатель или писатель (или ни одного). В случае блокировки чтения-записи может быть либо один писатель, либо несколько читателей – очень похоже на общий механизм работы ссылок в Rust.

В приложении `FernEmpireApp` могла бы быть структура для конфигурационных параметров, защищенная с помощью `RwLock`:

```
use std::sync::RwLock;

struct FernEmpireApp {
    ...
    config: RwLock<AppConfig>,
    ...
}
```

Методы, читающие конфигурацию, пользовались бы методом `RwLock::read()`:

```
/// True, если следует использовать экспериментальный код, учитывающий грибок.
fn mushrooms_enabled(&self) -> bool {
    let config_guard = self.config.read().unwrap();
    config_guard.mushrooms_enabled
}
```

Метод для перезагрузки конфигурации, вызывал бы `RwLock::write()`:

```
fn reload_config(&self) -> io::Result<()> {
    let new_config = AppConfig::load()?;
    let mut config_guard = self.config.write().unwrap();
    *config_guard = new_config;
    Ok(())
}
```

Конечно, Rust идеально подходит для гарантирования правил безопасной работы с `RwLock`. Концепция одного писателя или нескольких читателей лежит в основе системы заимствования в Rust. Метод `self.config.read()` возвращает стража, который дает изменяющий (разделяемый) доступ к `AppConfig`, а метод `self.config.write()` возвращает стража другого типа, который дает монопольный доступ, допускающий изменение.

Условные переменные (Condvar)

Часто поток должен ждать выполнения некоторого условия:

- в процессе остановки сервера главный поток должен дождаться завершения всех остальных потоков;
- когда рабочему потоку нечего делать, он должен ждать поступления данных для обработки;
- поток, реализующий протокол распределенного консенсуса, должен ждать получения ответа от кворума участников.

Иногда существует удобный блокирующий API, точно соответствующий ожидаемому условию, например в примере с остановкой сервера подошел бы метод `JoinHandle::join`. В других случаях готового блокирующего API нет. Для его построения программы могут воспользоваться *условными переменными*, которые в Rust реализованы типом `std::sync::Condvar`. В этом типе имеются методы `.wait()` и `.notify_all()`; `.wait()` блокирует поток, до тех пор пока какой-то другой поток не вызовет `.notify_all()`.

Но это еще не всё, поскольку условная переменная всегда ассоциирована с некоторым конкретным логическим условием, касающимся данных, которые защищены некоторым мьютексом. Поэтому мьютекс и условная переменная связаны между собой. Для полного объяснения у нас не хватит места, но программистам, которым раньше доводилось работать с условными переменными, помогут два небольших фрагмента кода ниже.

Как только интересующее условие станет равно `true`, мы вызываем `Condvar::notify_all` (или `notify_one`), чтобы разбудить ждущие его потоки:

```
self.has_data_condvar.notify_all();
```

Чтобы заснуть в ожидании условия, поток вызывает метод `Condvar::wait()`:

```
while !guard.has_data() {
    guard = self.has_data_condvar.wait(guard).unwrap();
}
```

Этот цикл `while` – стандартная идиома при работе с условными переменными. Однако сигнатура метода `Condvar::wait` необычна. Он принимает объект `MutexGuard` по значению, потребляет его и в случае успеха возвращает новый объект `MutexGuard`. Это соответствует интуитивному представлению о том, что метод `wait` освобождает мьютекс, а затем снова захватывает его перед возвратом. Передача `MutexGuard` по значению на самом деле выражает следующую мысль: «Дарую тебе, метод `.wait()`, мое исключительное право освобождать этот мьютекс».

Атомарные типы

Модуль `std::sync::atomic` содержит атомарные типы для безблокировочного конкурентного программирования. По существу, это те же типы, что в стандартном C++:

- `AtomicIsize` и `AtomicUsize` – разделяемые целые типы, соответствующие однопоточным типа `isize` и `usize`;
- `AtomicBool` – разделяемый аналог типа `bool`;
- `AtomicPtr<T>` – разделяемый аналог типа небезопасного указателя `*mut T`.

Описание корректной работы с атомарными данными выходит за рамки этой книги. Скажем лишь, что несколько потоков может одновременно читать и записывать атомарное значение, не опасаясь гонки за данные.

Вместо обычных арифметических и логических операторов в атомарных типах определены методы, выполняющие *атомарные операции*: загрузку, сохранение, обмен значениями и арифметические операции, которые производятся нераздельно и безопасно, даже когда другие потоки одновременно выполняют атомарные операции над теми же ячейками памяти. Инкремент значения `atom` типа `AtomicIsize` выглядит так:

```
use std::sync::atomic::Ordering;

atom.fetch_add(1, Ordering::SeqCst);
```

Эти методы могут компилироваться в специальные машинные команды. На машине с архитектурой x86-64 вызов `.fetch_add()` компилируется в команду `lock incq`, тогда как обычное выражение `n += 1` могло бы компилироваться в команду `incq` или ей подобную. Компилятор Rust вынужден также отказаться от некоторых оптимизаций в применении к атомарным операциям, потому что, в отличие от нормальной загрузки или сохранения значения в памяти, результаты этих операций должны быть сразу же видны другим потокам.

Аргумент `Ordering::SeqCst` задает *порядок доступа к памяти* – нечто похожее на уровень изоляции транзакций в базе данных. Порядок доступа говорит системе, что для вас важнее: философские понятия (причина должна предшествовать следствию, время не имеет циклов) или производительность. Порядок доступа к памяти критически важен для корректности программы, но сложен для понимания и рассуждений. По счастью, падение производительности из-за выбора последовательной согласованности, самого строгого порядка доступа к памяти, обычно очень мало – в отличие от того, что происходит, когда база данных SQL работает в режиме `SERIALIZABLE`. Так что, если сомневаетесь, выбирайте порядок `Ordering::SeqCst`. Rust наследует несколько других порядков доступа к памяти от атомарных типов стандартного C++ с различными ослабленными гарантиями относительно существования и времени. Обсуждать их мы не будем.

Одно из простых применений атомарных типов – прерывание потока. Пусть имеется поток, выполняющий некоторое длительное вычисление, например рендеринг видео, и мы хотели бы, чтобы его можно было асинхронно прервать. Проблема в том, как сообщить потоку, что он должен завершить работу. Это можно сделать с помощью разделяемой переменной типа `AtomicBool`.

```
use std::sync::atomic::{AtomicBool, Ordering};

let cancel_flag = Arc::new(AtomicBool::new(false));
let worker_cancel_flag = cancel_flag.clone();
```

Этот код создает два интеллектуальных указателя типа `Arc<AtomicBool>`, которые указывают на одно и то же значение типа `AtomicBool` в куче, первоначально равное `false`. Первый указатель, `cancel_flag`, остается в главном потоке, а второй, `worker_cancel_flag`, передается рабочему потоку.

Ниже приведен код рабочего потока:

```
let worker_handle = spawn(move || {
    for pixel in animation.pixels_mut() {
        render(pixel); // трассировка лучей - занимает несколько микросекунд
        if worker_cancel_flag.load(Ordering::SeqCst) {
            return None;
        }
    }
    Some(animation)
});
```

После отрисовки каждого пикселя поток проверяет значение флага, вызывая его метод `.load()`:

```
worker_cancel_flag.load(Ordering::SeqCst)
```

Если в главном потоке мы решим прервать рабочий поток, то сохраним значение `true` в `AtomicBool`, а затем дождемся завершения потока:

```
// Прервать рендеринг.
cancel_flag.store(true, Ordering::SeqCst);

// Отбросить результат, который, скорее всего, равен `None`.
worker_handle.join().unwrap();
```

Конечно, задачу можно решить и по-другому. Тип `AtomicBool` можно было бы заменить мьютексом `Mutex<bool>` или каналом. Основное отличие состоит в том, что у атомарных типов накладные расходы минимальны. Атомарные операции никогда не приводят к системному вызову. Загрузка и сохранение значений часто компилируются в одну машинную команду.

Атомарные типы – одна из форм внутренней изменяемости, как `Mutex` и `RwLock`, поэтому их методы принимают аргумент `self` по разделяемой ссылке.

Глобальные переменные

Предположим, что мы пишем сетевую программу. Мы хотели бы завести глобальную переменную – счетчик, который увеличивается после обслуживания каждого пакета.

```
/// Число успешно обработанных сервером пакетов.
static PACKETS_SERVED: usize = 0;
```

Этот код компилируется. Беда в том, что переменная `PACKETS_SERVED` неизменяемая, поэтому изменить ее не удастся.

Rust делает все, что в его силах, чтобы предотвратить появление глобального изменяемого состояния. Константы, объявленные с ключевым словом `const`, очевидно, неизменяемы. Статические переменные по умолчанию также неизменяемы, поэтому получить на них изменяемую ссылку невозможно. Статическую переменную можно объявить с ключевым словом `mut`, но тогда доступ к ней будет небезопасен. Приверженность Rust потокобезопасности – основная причина всех этих правил.

Глобальное изменяемое состояние имеет и неблагоприятные инженерные последствия: различные части программы становятся более тесно связаны, программу труднее тестировать и вносить в нее изменения. И все же в некоторых случаях просто не существует разумной альтернативы, так что хорошо бы найти безопасный способ объявления изменяемых статических переменных.

Самый простой способ поддержать инкремент переменной `PACKETS_SERVED`, не жертвуя потокобезопасностью, — сделать ее атомарным целым числом:

```
use std::sync::atomic::{AtomicUsize, ATOMIC_USIZE_INIT};

static PACKETS_SERVED: AtomicUsize = ATOMIC_USIZE_INIT;
```

Константа `ATOMIC_USIZE_INIT` имеет тип `AtomicUsize` и значение `0`. Мы используем ее вместо выражения `AtomicUsize::new(0)`, потому что начальное значение статической переменной должно быть константой, в версии Rust 1.17 вызовы методов для инициализации таких переменных запрещены. Аналогично константа

ATOMIC_ISIZE_INIT имеет тип AtomicIsize и равна нулю, а ATOMIC_BOOL_INIT имеет тип AtomicBool и равна false.

Теперь инкрементировать счетчик пакета очень просто:

```
PACKETS_SERVED.fetch_add(1, Ordering::SeqCst);
```

Атомарными глобальными переменными могут быть только простые целые и булевы величины. Создание глобальной переменной любого другого типа наталкивается на те же две проблемы, решить которые несложно.

1. Переменная должна быть потокобезопасной, поскольку иначе она не может быть глобальной: для безопасности статическая переменная должна реализовывать Sync и быть неизменяемой.
   К счастью, решение этой проблемы мы уже видели. В Rust есть типы для безопасного разделения значений, которые можно изменять: Mutex, RwLock и атомарные типы. Эти типы можно модифицировать с помощью разделяемой (без mut) ссылки. Для того они и предназначены (см. раздел «mut и Mutex» выше).
2. Как уже отмечалось, из статических инициализаторов нельзя вызывать функции. Это означает, что очевидный способ объявить статическую переменную типа Mutex работать не будет:

   ```
   static HOSTNAME: Mutex<String> =
       Mutex::new(String::new()); // ошибка: вызов функции в статическом инициализаторе
   ```

   Чтобы решить эту проблему, мы можем воспользоваться крейтом lazy_static.

С крейтом lazy_static мы познакомились в разделе «Ленивое построение значений типа Regex» главы 17. Определяя переменную с помощью макроса lazy_static!, мы получаем возможность инициализировать ее любым выражением; оно будет вычислено при первом разыменовании переменной, и значение сохранено для использования в будущем.

Мы можем объявить глобальный Mutex с помощью lazy_static следующим образом:

```
#[macro_use] extern crate lazy_static;

use std::sync::Mutex;

lazy_static! {
    static ref HOSTNAME: Mutex<String> = Mutex::new(String::new());
}
```

Такая же техника применима к переменным типа RwLock и AtomicPtr.

Использование lazy_static! влечет небольшие издержки при каждом доступе к статическим данным. В реализации используется тип std::sync::Once – низкоуровневый примитив синхронизации, предназначенный для одноразовой инициализации. За кулисами при каждом обращении к ленивой статической переменной программа выполняет команду атомарной загрузки, чтобы проверить, имела ли уже место инициализация. (Тип Once узко специализирован, поэтому мы не станем рассматривать его подробно. Обычно гораздо удобнее пользоваться макросом lazy_static!. Однако он удобен для инициализации библиотек, напи-

санных на других языках, см., например, раздел «Безопасный интерфейс к libgit2» главы 21.)

Как выглядит написание конкурентного кода на Rust

Мы продемонстрировали три способа работы с потоками в Rust: вилочный параллелизм, каналы и разделяемое изменяемое состояние, реализуемое с помощью блокировок. Нашей целью было дать введение в компоненты, предоставляемые Rust, с упором на то, как их можно сочетать в реальных программах.

Rust твердо настаивает на безопасности, поэтому, начиная с момента, когда вы решите написать многопоточную программу, следует акцентировать внимание на создании безопасного структурированного механизма коммуникации. Максимальная изоляция потоков – неплохой способ убедить Rust в безопасности того, что вы делаете. И по странному совпадению, изоляция – также неплохой способ убедиться в том, что сделанное вами корректно и удобно для сопровождения. Повторим: Rust поощряет написание хороших программ.

Но важнее то, что Rust позволяет комбинировать разные подходы и экспериментировать. Переходить от одной итерации к следующей можно быстро: споры с компилятором приводят к созданию корректно работающей программы гораздо быстрее, чем отладка гонок за данные.

Глава 20

Макросы

> Центон (от латинского *cento* – лоскутное покрывало) – стихотворение, целиком составленное из строк других стихотворений.
>
> *— Мэтт Мэдден*

> Здесь может быть ваша цитата.
>
> *— Бьярн Страуструп*

Rust поддерживает *макросы*, способ расширения языка такими способами, которых одними функциями достичь невозможно. Например, мы видели макрос `assert_eq!`, удобный для написания тестов:

```
assert_eq!(gcd(6, 10), 2);
```

Это можно было бы написать в виде универсальной функции, но макрос `assert_eq!` делает кое-что, недоступное функциям. Например, если утверждение не выполнено, то `assert_eq!` порождает сообщение об ошибке, содержащее имя файла и номер строки, где произошла ошибка. Функции никак не могут получить такую информацию. А макросы могут, потому что работают совершенно иначе.

Макрос – это своего рода стенограмма. В процессе компиляции еще до проверки типов и задолго до генерации машинного кода каждый вызов макроса *расширяется*, т. е. заменяется каким-то Rust-кодом. Приведенный выше вызов макроса расширяется в такой код:

```
match (&gcd(6, 10), &2) {
    (left_val, right_val) => {
        if !(*left_val == *right_val) {
            panic!("assertion failed: `(left == right)`, \
                    (left: `{:?}`, right: `{:?}`)", left_val, right_val);
        }
    }
}
```

`panic!` – также макрос, поэтому тоже расширяется в некий Rust-код. В этом коде используются два других макроса: `file!()` и `line!()`. После того как все вызовы макросов в крейте полностью расширены, Rust переходит к следующей фазе компиляции.

На этапе выполнения неправильное утверждение привело бы к такому результату (и означало бы, что в функции `gcd()` ошибка, потому что `2` – правильный ответ):

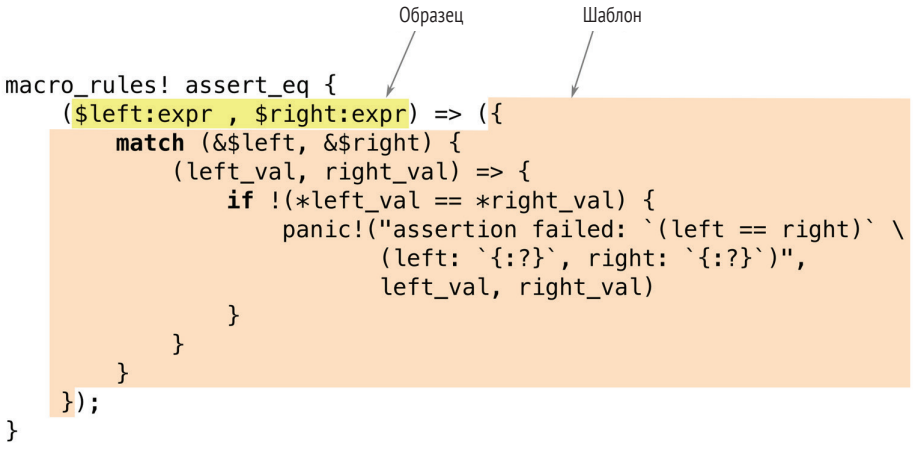
```
thread 'main' panicked at 'assertion failed: `(left == right)`, (left: `17`,
right: `2`)', gcd.rs:7
```

Если вы раньше работали с C++, то, возможно, о макросах у вас остались плохие воспоминания. В Rust к макросам принят другой подход, похожий на синтаксические правила (`syntax-rules`) в языке Scheme. По сравнению с C++, макросы Rust лучше интегрированы в язык и потому менее подвержены ошибкам. Вызов макроса всегда помечается восклицательным знаком, поэтому они легко отличимы от остального кода и случайно вызвать их вместо функции невозможно. Макросы Rust никогда не вставляют непарных круглых или квадратных скобок. А благодаря прилагающемуся механизму сравнения с образцом становится легко писать сопровождаемые и приятные в использовании макросы.

В этой главе мы продемонстрируем написание макросов на нескольких примерах. А затем разберемся с тем, как они в действительности работают, потому что, как и многое в Rust, этот инструмент тем легче использовать, чем глубже его понимаешь. И напоследок мы увидим, что можно сделать, когда простого сопоставления с образцом недостаточно.

Основы макросов

Вот часть исходного кода макроса `assert_eq!`:



Макрос `macro_rules!` – основной способ определения макросов в Rust. Обратите внимание, что в определении макроса `assert_eq` отсутствует `!` после имени – восклицательный знак добавляется только при вызове макроса.

Не все макросы определяются таким образом. Некоторые, например `file!`, `line!` и сам макрос `macro_rules!`, встроены в компилятор, а еще один подход – процедурные макросы – мы обсудим в конце главы. Но основное внимание мы все же уделим макросу `macro_rules!`, потому что это самый легкий (на данный момент) способ написать свой макрос.

Макрос, определенный с помощью `macro_rules!`, полностью описывается сравнением с образцами. Тело макроса представляет собой просто последовательность правил:

```
( образец1 ) => ( шаблон1 );
( образец2 ) => ( шаблон2 );
...
```

В той версии макроса `assert_eq!`, которая показана выше, есть только один образец и только один шаблон.

Кстати говоря, образец и шаблон можно заключать не только в круглые, но и в квадратные или фигурные скобки, для Rust это не играет роли. Аналогично следующие формы вызова макроса эквивалентны:

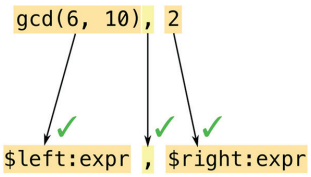
```
assert_eq!(gcd(6, 10), 2);
assert_eq![gcd(6, 10), 2];
assert_eq!{gcd(6, 10), 2}
```

Единственное отличие состоит в том, что после фигурной скобки точка с запятой необязательна. По соглашению принято использовать круглые скобки при вызове `assert_eq!`, квадратные – при вызове `vec!` и фигурные – для `macro_rules!`; но это всего лишь соглашение.

Основы макрорасширения

Rust расширяет макросы на очень ранней стадии компиляции. Компилятор читает код от начала до конца и по ходу дела определяет и расширяет макросы. Вызывать макрос до того, как он определен, не разрешается, потому что Rust расширяет каждый вызов макроса, даже не прочитав программу до конца. (Напротив, функции и другие артикулы не обязаны следовать в каком-то определенном порядке. Разрешается вызывать функцию, которая будет определена в крейте позже.)

Происходящее при расширении макроса `assert_eq!` очень напоминает вычисление выражения `match`. Rust сначала сопоставляет аргументы с образцом:



Образцы макросов записываются на специальном мини-языке внутри Rust. По сути дела, это регулярные выражения для сопоставления с кодом. Но если регулярные выражения работают с символами, то образцы – с лексемами: числами, именами, знаками препинания и прочими элементами Rust-программ. Поэтому в образцах макросов можно употреблять комментарии и символы пропуска, чтобы сделать их максимально простыми для восприятия. Комментарии и пропуски не являются лексемами, поэтому на сопоставление не влияют.

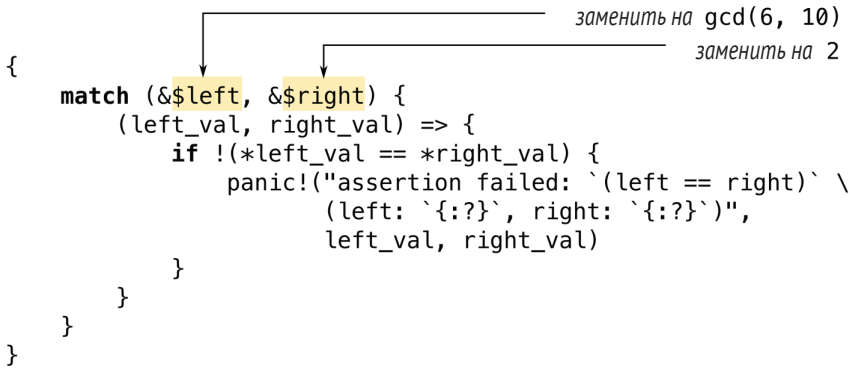
Еще одно важное различие между регулярными выражениями и образцами макросов заключается в том, что круглые, квадратные и фигурные скобки всегда должны быть парными. Это проверяется до расширения макроса, не только в образцах макросов, но и всюду в языке.

В этом примере образец имеет вид `$left:expr`, для Rust это означает, что нужно сопоставить выражение (в данном случае `gcd(6, 10)`) и присвоить его имени `$left`.

Затем Rust сопоставляет запятую в образце с запятой в аргументах. Как и регулярные выражения, образцы могут содержать несколько специальных символов, порождающих нетривиальное поведение на этапе сопоставления; все прочие символы, в т. ч. и запятая, должны совпадать буквально, иначе считается, что сопоставить с образцом не удалось. Наконец, Rust сопоставляет выражение `2` и присваивает его имени `$right`.

Оба фрагмента кода в этом образце имеют тип `expr`: ожидается, что вместо них будут подставлены выражения. Фрагменты других типов мы рассмотрим в разделе «Типы фрагментов».

Поскольку этот образец удалось сопоставить со всеми аргументами, Rust расширяет соответствующий ему *шаблон*:



Rust просто заменяет `$left` и `$right` фрагментами кода, найденными в процессе сопоставления.

Типичная ошибка – включать тип фрагмента в выходной шаблон, т. е. писать `$left:expr`, а не просто `$left`. Rust не обнаруживает такую ошибку немедленно. Он рассматривает `$left` как замену, а `:expr` трактует как обычную лексему, которую следует включить в выход макроса. Поэтому ошибка не проявится до момента *вызова* макроса, и вот тогда будет сгенерирован некомпилируемый мусор. Увидев сообщение вида `expected type, found `:`` при использовании макроса, проверьте, не допустили ли вы такую ошибку (в разделе «Отладка макросов» ниже приведены дополнительные рекомендации на эту тему).

Шаблоны макросов мало чем отличаются от десятка языков шаблонов, используемых в веб-программировании. Единственное – и существенное – отличие состоит в том, что на выходе получается код на Rust.

Непредвиденные последствия

Подстановка фрагментов кода в шаблон имеет тонкие отличия от обычного кода, работающего со значениями. Эти отличия не всегда очевидны с первого взгляда. Рассматриваемый сейчас макрос `assert_eq!` содержит несколько странных кусков кода, которые многое могут сказать о программировании макросов. Рассмотрим два особенно интересных куска.

Во-первых, почему этот макрос создает переменные `left_val` и `right_val`? Нельзя ли упростить шаблон следующим образом:

```
if !($left == $right) {
    panic!("assertion failed: `(left == right)` \
            (left: `{:?}`, right: `{:?}`)", $left, $right)
}
```

Чтобы ответить на этот вопрос, попробуем мысленно расширить вызов макроса `assert_eq!(letters.pop(), Some('z'))`. Что получится на выходе? Естественно, Rust подставит сопоставленные выражения в несколько мест шаблона. Но идея вычислять выражения дважды при построении сообщения об ошибке никуда не годится – и не только потому, что это займет в два раза больше времени: поскольку вызов `letters.pop()` удаляет значение из вектора, при втором вызове значение будет другим! Именно поэтому реальные макросы вычисляют `$left` и `$right` один раз и сохраняют значения.

Перейдем ко второму вопросу: почему макрос заимствует ссылки на значения `$left` и `$right`? Почему бы не сохранить их прямо в переменных:

```
macro_rules! bad_assert_eq {
    ($left:expr, $right:expr) => ({
        match ($left, $right) {
            (left_val, right_val) => {
                if !(left_val == right_val) {
                    panic!("assertion failed" /* ... */);
                }
            }
        }
    });
}
```

Для конкретного рассматриваемого случая, когда аргументами макроса являются целые числа, это сработало бы. Но если вызывающая сторона передаст переменную типа `String` в качестве `$left` или `$right`, то показанный выше код отнимет у этой переменной владение значением!

```
fn main() {
    let s = "a rose".to_string();
    bad_assert_eq!(s, "a rose");
    println!("confirmed: {} is a rose", s);  // ошибка: значение "s" уже передано
}
```

Поскольку мы не хотим, чтобы утверждение меняло владельца значения, макрос заимствует ссылки. (Возможно, у вас возник вопрос, почему для определения переменных в этом макросе используется предложение `match`, а не `let`. Нам это тоже стало интересно. Как оказалось, никакой особой причины нет. Использование `let` дало бы такой же результат.)

Короче говоря, макросы способны преподнести сюрпризы. Если вокруг написанного вами макроса происходят странные вещи, можно держать пари, что макрос-то и виноват.

Но вот чего вы никогда *не встретите*, так это классической ошибки в макросе C++:

```
// ошибочный макрос C++ для прибавления 1 к числу
#define ADD_ONE(n)  n + 1
```

По причинам, знакомым большинству программистов на C++, в которые мы здесь вдаваться не будем, ничем не примечательный код вида `ADD_ONE(1) * 10` или `ADD_ONE(1 << 4)` дает удивительные результаты при таком определении макроса. Чтобы исправить ошибку, нужно включить дополнительные скобки в определение. В Rust это необязательно, потому что макросы лучше интегрированы с языком. Rust знает, когда обрабатывает выражения, поэтому сам добавляет скобки при подстановке одного выражения в другое.

Повторение

У стандартного макроса `vec!` есть две формы:

```
// Повторить значение N раз
let buffer = vec![0_u8; 1000];

// Список значений через запятую
let numbers = vec!["udon", "ramen", "soba"];
```

Вот как его можно реализовать:

```
macro_rules! vec {
    ($elem:expr ; $n:expr) => {
        ::std::vec::from_elem($elem, $n)
    };
    ( $( $x:expr ),* ) => {
        <[_]>::into_vec(Box::new([ $( $x ),* ]))
    };
    ( $( $x:expr ),+ ,) => {
        vec![ $( $x ),* ]
    };
}
```

Здесь мы видим три правила. Сначала объясним, как вообще работает макрос с несколькими правилами, а затем рассмотрим каждое правило в отдельности.

Расширяя вызов макроса вида `vec![1, 2, 3]`, Rust сначала сопоставляет аргументы `1, 2, 3` с образцом в первом правиле, в данном случае `$elem:expr ; $n:expr`. Это не проходит: `1` – выражение, но образец требует, чтобы после него была точка с запятой, а у нас ее нет. Поэтому Rust переходит ко второму правилу и т. д. Если не удалось сопоставить вызов ни с одним правилом, фиксируется ошибка.

Первое правило обрабатывает вызовы вида `vec![0u8; 1000]`. Существует стандартная функция `std::vec::from_elem`, которая делает в точности то, что здесь нужно, так что это правило простое.

Второе правило обрабатывает вызовы вида `vec!["udon", "ramen", "soba"]`. В образце `$( $x:expr ),*` используется возможность, которая нам раньше не встречалась: повторение. Он сопоставляется с 0 или более выражений, разделенных запятыми. Вообще, синтаксическая конструкция `$( PATTERN ),*` применяется для сопоставления со списком элементов через запятую, в котором каждый элемент сопоставляется с образцом `PATTERN`.

Символ `*` здесь имеет тот же смысл, что в регулярных выражениях (0 или более), хотя в регулярных выражениях нет специального повторителя `,*`. Можно также использовать символ `+`, означающий, что должно быть хотя бы одно соответствие.

Символ ? в этом месте не предусмотрен. В таблице ниже перечислены все варианты образцов с повторением.

Образец	Пояснение
$(...)*	повторение 0 или более раз без разделителя
$(...),*	повторение 0 или более с разделителем «запятая»
$(...);*	повторение 0 или более с разделителем «точка с запятой»
$(...)+	повторение 1 или более раз без разделителя
$(...),+	повторение 1 или более с разделителем «запятая»
$(...);+	повторение 1 или более с разделителем «точка с запятой»

Фрагмент кода $x – не одно выражение, а список выражений. В шаблоне для этого правила также используется синтаксис повторения:

```
<[_]>::into_vec(Box::new([ $( $x ),* ]))
```

И в этом случае для того, что нам необходимо, есть стандартные методы. В этом коде создается массив в боксе, а затем вызывается метод [T]::into_vec, который преобразует этот массив в вектор.

<[_]> – необычный способ записать тип «срезки чего-либо», ожидая, что Rust сам выведет тип элемента. Типы, имена которых являются простыми идентификаторами, можно использовать в выражениях без всяких церемоний, но такие типы, как fn(), &str или [_], следует заключать в угловые скобки.

Повторение встречается в конце шаблона: $($x),*. Конструкция $(...),* – то же самое, что мы уже видели. Она служит для обхода списка выражений, сопоставленных с $x, и вставки их всех в шаблон через запятую.

В данном случае повторенный выход выглядит в точности как вход. Но так бывает не всегда. Это правило можно было бы переписать в виде:

```
( $( $x:expr ),* ) => {
    {
        let mut v = Vec::new();
        $( v.push($x); )*
        v
    }
};
```

Здесь часть шаблона $(v.push($x);)* вставляет вызов v.push() для каждого выражения в $x.

В отличие от других частей Rust, образцы, в которых используется конструкция $(...),*, автоматически не поддерживают факультативную запятую в конце. Однако существует стандартный прием для поддержки конечной запятой с помощью дополнительного правила. Именно для этого и нужно третье правило в макросе vec!:

```
( $( $x:expr ),+ ,) => {  // если есть конечная запятая,
    vec![ $( $x ),* ]     // попробовать еще раз без нее
};
```

Часть $(...),+ , сопоставляется со списком, содержащим лишнюю запятую. А затем в шаблоне мы рекурсивно вызываем vec!, убирая эту запятую. Теперь сопоставится второе правило.

Встроенные макросы

В компилятор Rust встроено несколько макросов, полезных для определения собственных. Ни один из них нельзя было бы реализовать с помощью одного лишь макроса `macro_rules!`:

- макрос **`file!()`** расширяется в строковый литерал: имя текущего файла. Макросы **`line!()`** и **`column!()`** расширяются в литералы типа `u32`: номер текущей строки (нумерация начинается с 1) и столбца (нумерация начинается с 0).
  Если один макрос вызывает другой, а тот – третий, причем все вызовы находятся в разных файлах, и последний макрос вызывает `file!()`, `line!()` или `column!()`, то результатом расширения будет место вызова *первого* макроса;
- макрос **`stringify!(...tokens...)`** расширяется в строковый литерал, содержащий указанные лексемы. Макрос `assert!` пользуется этим макросом для генерации сообщения об ошибке, включающего код утверждения.
  Вызовы макросов в аргументе *не* расширяются: `stringify!(line!())` расширяется в строку `"line!()"`.
  Rust конструирует строку из лексем, поэтому в ней нет ни символов новой строки, ни комментариев;
- макрос **`concat!(str0, str1, ...)`** расширяется в строковый литерал, равный результату конкатенации всех аргументов.

Rust также определяет следующие макросы для опроса среды сборки:

- **`cfg!(...)`** расширяется в булеву константу, равную `true`, если текущая конфигурация сборки удовлетворяет указанному в скобках условию. Например, `cfg!(debug_assertions)` равно `true`, если компиляция производится с включенными отладочными утверждениями.
  Этот макрос поддерживает в точности такой же синтаксис, как атрибут `#[cfg(...)]`, описанный в разделе «Атрибуты» главы 8, но результатом является не условная компиляция, а ответ `true` или `false`;
- **`env!("VAR_NAME")`** расширяется в строку: значение указанной переменной окружения на этапе компиляции. Если переменная не существует, возникает ошибка компиляции.
  Этот макрос был бы практически бесполезен, если бы Cargo не устанавливала несколько интересных переменных окружения при компиляции крейта. Например, вот как можно получить текущую строку версии крейта:

  ```
  let version = env!("CARGO_PKG_VERSION");
  ```

  Полный перечень этих переменных окружения можно найти в документации по Cargo (http://doc.crates.io/environment-variables.html#environment-variables-cargo-sets-for-crates);
- **`option_env!("VAR_NAME")`** – то же, что `env!`, но возвращает значение типа `Option<&'static str>`, равное `None`, если указанная переменная не установлена.

Еще три встроенных макроса позволяют включить код или данные из другого файла:

- результатом расширения макроса **`include!("file.rs")`** является содержимое указанного файла, которое должно быть корректным Rust-кодом: выражением или последовательностью артикулов;
- **`include_str!("file.txt")`** расширяется в строку `&'static str`, содержащую текст указанного файла. Использовать его можно следующим образом:

```
const COMPOSITOR_SHADER: &str =
    include_str!("../resources/compositor.glsl");
```

Если файл не существует или его содержимое – не текст в кодировке UTF-8, компилятор выдает ошибку;

- `include_bytes!("file.dat")` – то же самое, только файл рассматривается как последовательность двоичных данных, а не текст в кодировке UTF-8. Результат имеет тип `&'static [u8]`.

Эти макросы, как и все прочие, обрабатываются на этапе компиляции. Если файл не существует или не может быть прочитан, возникает ошибка компиляции. Таким образом, на этапе выполнения они не могут завершиться с ошибкой. Во всех случаях, если задан относительный путь к файлу, он разрешается относительно каталога, содержащего текущий файл.

Отладка макросов

Отладка непослушного макроса может оказаться трудной задачей. Самая большая проблема – невозможность заглянуть внутрь процесса макрорасширения. Rust зачастую расширяет все макросы, находит какую-то ошибку и печатает сообщение, в котором не показан полностью расширенный код, содержащий эту ошибку!

Мы опишем три средства для отладки макросов. (Все они нестабильны, но поскольку предназначены для использования в процессе разработки, а не в коде, помещаемом в репозиторий системы управления версиями, на практике это не страшно.)

Первое и самое простое – попросить `rustc` показать, как выглядит код после расширения всех макросов. Выполните команду `cargo build -verbose`, чтобы увидеть, как Cargo вызывает `rustc`. Скопируйте командную строку `rustc` и добавьте в нее три параметра: `-Z unstable-options --pretty expanded`. На экран терминала будет выведен полностью расширенный код. К сожалению, это работает, только если в коде нет синтаксических ошибок.

Во-вторых, Rust предоставляет макрос `log_syntax!()`, который просто печатает свои аргументы на терминале во время компиляции. Его можно использовать для отладки в стиле `println!`. Этот макрос требует флага `#![feature(log_syntax)]`.

В-третьих, мы можем попросить компилятор протоколировать на терминале все вызовы макросов. Вставьте куда-нибудь строчку `trace_macros!(true);`. Начиная с этого места, при расширении любого макроса будут печататься его имя и аргументы. Например, следующая программа:

```
#![feature(trace_macros)]

fn main() {
    trace_macros!(true);
    let numbers = vec![1, 2, 3];
    trace_macros!(false);
    println!("total: {}", numbers.iter().sum::<u64>());
}
```

порождает такой результат:

```
$ rustup override set nightly
...
```

```
$ rustc trace_example.rs
note: trace_macro
 --> trace_example.rs:5:19
  |
5 |     let numbers = vec![1, 2, 3];
  |                   ^^^^^^^^^^^^
  |
  = note: expanding `vec! { 1 , 2 , 3 }`
  = note: to `< [ _ ] > :: into_vec ( box [ 1 , 2 , 3 ] )`
```

Компилятор показывает код каждого вызова макроса до и после расширения. Строчка `trace_macros!(false);` отключает трассировку, поэтому обращение к `println!()` не трассируется.

Макрос json!

Мы обсудили базовые возможности макроса `macro_rules!`. В этом разделе мы пошагам разработаем макрос для построения данных в формате JSON. На этом примере мы покажем, как выглядит процесс разработки макроса, познакомимся с ранее не описанными частями `macro_rules!` и дадим рекомендации о том, как убедиться, что ваши макросы работают, как задумано.

В главе 10 мы рассматривали такое перечисление для представления данных в формате JSON:

```
#[derive(Clone, PartialEq, Debug)]
    enumJson {
    Null,
    Boolean(bool),
    Number(f64),
    String(String),
    Array(Vec<Json>),
    Object(Box<HashMap<String, Json>>)
}
```

Но запись значений типа `Json` уж слишком громоздкая:

```
let students = Json::Array(vec![
    Json::Object(Box::new(vec![
        ("name".to_string(), Json::String("Jim Blandy".to_string())),
        ("class_of".to_string(), Json::Number(1926.0)),
        ("major".to_string(), Json::String("Tibetan throat singing".to_string()))
    ].into_iter().collect())),
    Json::Object(Box::new(vec![
        ("name".to_string(), Json::String("Jason Orendorff".to_string())),
        ("class_of".to_string(), Json::Number(1702.0)),
        ("major".to_string(), Json::String("Knots".to_string()))
    ].into_iter().collect()))
]);
```

А хотелось бы записывать их в более привычном виде:

```
let students = json!([
    {
        "name": "Jim Blandy",
```

```
        "class_of": 1926,
        "major": "Tibetan throat singing"
    },
    {
        "name": "Jason Orendorff",
        "class_of": 1702,
        "major": "Knots"
    }
]);
```

В общем, нам нужен макрос `json!`, который принимает значение в формате JSON в качестве аргумента и расширяет его в выражение Rust – такое, как в примере выше.

Типы фрагментов

Первая задача при написании любого сложного макроса – понять, как будет осуществляться сопоставление входных данных, или *разбор*.

Сразу видно, что у макроса будет несколько правил, поскольку в JSON-данных бывают разные структурные элементы: объекты, массивы, числа и т. д. Можно предположить, что должно быть по одному правилу для каждого типа JSON.

```
macro_rules! json {
    (null) => { Json::Null };
    ([ ... ]) => { Json::Array(...) };
    ({ ... }) => { Json::Object(...) };
    (???)     => { Json::Boolean(...) };
    (???)     => { Json::Number(...) };
    (???)     => { Json::String(...) };
}
```

Это не совсем правильно, потому что образцы макросов не дают никакой возможности различить последние три случая, но ниже мы увидим, как с этим справиться. Однако, по крайней мере, первые три случая начинаются четко различимыми лексемами, так что с них и начнем.

Первое правило уже работает.

```
macro_rules! json {
    (null) => {
        Json::Null
    }
}

#[test]
fn json_null() {
    assert_eq!(json!(null), Json::Null);  // проходит!
}
```

Чтобы добавить поддержку JSON-массивов, можно было бы попробовать сопоставлять элементы как выражения `expr`.

```
macro_rules! json {
    (null) => {
        Json::Null
```

```
    };
    ([ $( $element:expr ),* ]) => {
        Json::Array(vec![ $( $element ),* ])
    };
}
```

К сожалению, при этом сопоставляются не все JSON-массивы. Вот тест, иллюстрирующий, в чем проблема:

```
#[test]
fn json_array_with_json_element() {
    let macro_generated_value = json!(
        [
            // допустимый JSON, не сопоставляемый с `$element:expr`
            {
                "pitch": 440.0
            }
        ]
    );
    let hand_coded_value =
        Json::Array(vec![
            Json::Object(Box::new(vec![
                ("pitch".to_string(), Json::Number(440.0))
            ].collect()))
        ]);
    assert_eq!(macro_generated_value, hand_coded_value);
}
```

Образец `$( $element:expr ),*` означает «список выражений Rust, разделенных запятыми». Но многие значения JSON, в частности объекты, не являются допустимыми выражениями Rust. Они и не сопоставятся.

Поскольку не каждый кусок кода, который мы хотели бы сопоставлять, является выражением, Rust поддерживает еще несколько типов фрагментов, перечисленных в табл. 20.1.

Таблица 20.1. Типы фрагментов, поддерживаемые `macro_rules!`

Тип фрагмента	Сопоставляется (с примерами)	За ним может следовать
`expr`	Выражение: `2 + 2`, `"udon"`, `x.len()`	`=> , ;`
`stmt`	Выражение или объявление, не включающее завершающую точку с запятой (трудно использовать, попробуйте вместо этого `expr` или `block`)	`=> , ;`
`ty`	Тип: `String`, `Vec<u8>`, `(&str, bool)`	`=> , = | { [ : > as where`
`path`	Путь: `ferns`, `::std::sync::mpsc`	`=> , = | { [ : > as where`
`pat`	Образец: `_`, `Some(ref x)`	`=> , = | if in`
`item`	Артикул: `struct Point { x: f64, y: f64 }`, `mod ferns;`	все что угодно
`block`	Блок: `{ s += "ok\n"; true }`	все что угодно

Окончание табл. 20.1

Тип фрагмента	Сопоставляется (с примерами)	За ним может следовать
`meta`	Тело атрибута: `inline`, `derive(Copy, Clone)`, `doc="3D models."`	все что угодно
`ident`	Идентификатор: `std`, `Json`, `longish_variable_name`	все что угодно
`tt`	Дерево лексем (см. в тексте): `;`, `>=`, `{}`, `[0 1 (+ 0 1)]`	все что угодно

Большинство вариантов в этой таблице строго следует синтаксису Rust. Тип `expr` сопоставляется только с выражениями Rust (а не со значениями JSON), `ty` – только с типами Rust и т. д. Эти типы не расширяются: нельзя определить новые арифметические операторы или ключевые слова, которые распознавал бы тип фрагмента `expr`. Мы не сможем использовать такие типы для сопоставления с произвольными JSON-данными.

Но последние два типа, `ident` и `tt`, поддерживают сопоставление с аргументами макроса, не обязательно выглядящими как код на Rust. `ident` сопоставляется с любым идентификатором, а `tt` – с одним *деревом лексем*: либо с парой скобок `(...)` `[...]` или `{...}` и всем, что заключено между ними, включая вложенные деревья лексем, либо с одной лексемой, не являющейся скобкой, например `1926` или `"Knots"`.

Деревья лексем – именно то, что нужно для нашего макроса `json!`. Каждое значение JSON – это одно дерево лексем: числа, строки, булевы величины и `null` – одиночные лексемы, а объекты и массивы заключены в скобки. Следовательно, образцы можно записать так:

```
macro_rules! json {
    (null) => {
        Json::Null
    };
    ([ $( $element:tt ),* ]) => {
        Json::Array(...)
    };
    ({ $( $key:tt : $value:tt ),* }) => {
        Json::Object(...)
    };
    ($other:tt) => {
        ... // TODO: вернуть Number, String или Boolean
    };
}
```

Этот вариант макроса `json!` может сопоставляться с любыми JSON-данными. Остается только сгенерировать правильный Rust-код.

Чтобы можно было включать новые синтаксические конструкции в будущие версии языка, не нарушая работоспособность уже имеющихся макросов, Rust вводит ограничения на то, какие лексемы могут встречаться в образцах после фрагмента каждого типа. В столбце «За ним может следовать» показано, какие лексемы допустимы. Например, образец `$x:expr ~ $y:expr` ошибочный, потому что `~` не может находиться после `expr`. Образец `$val:expr : $t:ty` правильный, потому что за `$val:expr` следует двоеточие – одна из разрешенных после `expr` лексем, тогда как за `$t:ty` нет вообще ничего, а это разрешено всегда.

Рекурсия в макросах

Мы уже видели один тривиальный случай, когда макрос вызывает сам себя: в реализации `vec!` рекурсия используется для поддержки запятых после последнего элемента вектора. А сейчас мы покажем более интересный пример: макрос `json!` должен вызывать себя рекурсивно.

Можно было бы попробовать поддержать массивы JSON без рекурсии, как-то так:

```
([ $( $element:tt ),* ]) => {
    Json::Array(vec![ $( $element ),* ])
};
```

Но это не работает. Мы пытаемся вставить JSON-данные (деревья лексем `$element`) прямо в выражение Rust. Но это два разных языка.

Нужно преобразовать каждый элемент массива из формата JSON в Rust. К счастью, такой макрос существует, а именно тот, который мы разрабатываем!

```
([ $($element:tt),* ]) => {
    Json::Array(vec![ $( json!($element) ),* ])
};
```

Объекты можно поддержать точно так же:

```
({ $($key:tt : $value:tt),* }) => {
    Json::Object(Box::new(vec![
        $( ($key.to_string(), json!($value)) ),*
    ].into_iter().collect()))
};
```

По умолчанию компилятор ограничивает глубину рекурсии 64 вызовами. При обычном использовании `json!` этого более чем достаточно, но в сложных рекурсивных макросах иногда не хватает. Чтобы увеличить глубину, поместите следующий атрибут в начало крейта, в котором используется макрос:

```
#![recursion_limit = "256"]
```

Наш макрос `json!` почти готов. Осталось только поддержать булевы, числовые и строковые значения.

Использование характеристик совместно с макросами

При написании сложных макросов всегда возникают загадки. Важно помнить, что сами макросы – не единственный инструмент решения загадок, имеющийся в вашем распоряжении.

В данном случае нам нужно поддержать вызовы `json!(true)`, `json!(1.0)` и `json!("yes")`, преобразующие значения различных типов в соответствующее значение типа `Json`. Но макросы – неподходящее средство для различения типов. Можно было бы попробовать написать что-то в таком духе:

```
macro_rules! json {
    (true) => {
        Json::Boolean(true)
    };
    (false) => {
```

```
        Json::Boolean(false)
    };
    ...
}
```

Но несостоятельность этой идеи бросается в глаза. Булевых значений действительно всего два, но чисел, а тем более строк куда больше.

По счастью, существует стандартный способ преобразовать значения разных типов в один заданный тип: характеристика `From`. Нужно только реализовать ее для нескольких типов:

```
impl From<bool> for Json {
    fn from(b: bool) -> Json {
        Json::Boolean(b)
    }
}

impl From<i32> for Json {
    fn from(i: i32) -> Json {
        Json::Number(i as f64)
    }
}

impl From<String> for Json {
    fn from(s: String) -> Json {
        Json::String(s)
    }
}

impl<'a> From<&'a str> for Json {
    fn from(s: &'a str) -> Json {
        Json::String(s.to_string())
    }
}
...
```

На самом деле такие реализации необходимы для всех 12 числовых типов, поэтому имеет смысл написать макрос, чтобы избежать копирования и вставки:

```
macro_rules! impl_from_num_for_json {
    ( $( $t:ident )* ) => {
        $(
            impl From<$t> for Json {
                fn from(n: $t) -> Json {
                    Json::Number(n as f64)
                }
            }
        )*
    };
}

impl_from_num_for_json!(u8 i8 u16 i16 u32 i32 u64 i64 usize isize f32 f64);
```

Теперь можно использовать `Json::from(value)` для преобразования значения `value` любого поддерживаемого типа в тип `Json`. В нашем макросе это будет выглядеть так:

```
($other:tt) => {
    Json::from($other)  // обработка булевых, числовых и строковых значений
};
```

После добавления этого правила в макрос `json!` все написанные ранее тесты проходят. Соберем все вместе:

```
macro_rules! json {
    (null) => {
        Json::Null
    };
    ([ $($element:tt),* ]) => {
        Json::Array(vec![ $( json!($element) ),* ])
    };
    ({ $($key:tt : $value:tt),* }) => {
        Json::Object(Box::new(vec![
            $( ($key.to_string(), json!($value)) ),*
        ].into_iter().collect()))
    };
    ($other:tt) => {
        Json::from($other)  // обработка булевых, числовых и строковых значений
    };
}
```

Неожиданно выясняется, что этот макрос поддерживает использование переменных и даже произвольных выражений Rust внутри JSON-дата, – приятный бонус:

```
let width = 4.0;
let desc =
    json!({
        "width": width,
        "height": (width * 9.0 / 4.0)
    });
```

Поскольку выражение `(width * 9.0 / 4.0)` заключено в скобки, оно является одним деревом лексем, так что при разборе объекта макрос сопоставляет его с `$value:tt`.

Области видимости и гигиена

На удивление каверзным аспектом написания макросов является тот факт, что они склеивают вместе код из разных областей видимости. Ниже мы рассмотрим два разных способа обработки областей видимости в Rust: один – для локальных переменных и аргументов, другой – для всего остального.

Чтобы показать, почему это важно, перепишем правило разбора JSON-объектов (третье правило макроса `json!`), чтобы избавиться от временного вектора. Это можно сделать так:

```
({ $($key:tt : $value:tt),* }) => {
    {
        let mut fields = Box::new(HashMap::new());
        $( fields.insert($key.to_string(), json!($value)); )*
        Json::Object(fields)
    }
};
```

Теперь для заполнения `HashMap` мы не пользуемся методом `collect()`, а повторно вызываем метод `.insert()`. Это означает, что отображение нужно сохранить во временной переменной, которую мы назвали `fields`.

Но что произойдет, если в коде, вызывающем `json!`, случайно окажется переменная с таким же именем `fields`?

```
let fields = "Fields, W.C.";
let role = json!({
    "name": "Larson E. Whipsnade",
    "actor": fields
});
```

При расширении макроса склеиваются два разных фрагмента кода, в каждом из которых имя `fields` используется для разных целей!

```
let fields = "Fields, W.C.";
let role = {
    let mut fields = Box::new(HashMap::new());
    fields.insert("name".to_string(), Json::from("Larson E. Whipsnade"));
    fields.insert("actor".to_string(), Json::from(fields));
    Json::Object(fields)
};
```

Может показаться, что это неизбежная неприятность, сопряженная с использованием временных переменных в макросах, и, возможно, вы уже задумались о том, как это поправить. Ну, скажем, переименовать переменную, определенную в макросе `json!`, выбрав имя, которое вряд ли встретится в вызывающей программе: не `fields`, а `__json$fields`.

Но удивительно то, что *изначальный макрос тоже работает*. Rust позаботился о переименовании за вас! Эта идея, впервые реализованная в макросах языка Scheme, называется *гигиеной*, поэтому говорят, что макросы в Rust *гигиенические*.

Понять, как устроена гигиена макросов, проще всего, представив себе, что те части макрорасширения, источником которых является сам макрос, набраны другим цветом.

Тогда переменные разного цвета обрабатываются так, будто у них разные имена.

```
let fields = "Fields, W.C.";
let role = {
    let mut fields = Box::new(HashMap::new());
    fields.insert("name".to_string(), Json::from("Larson E. Whipsnade"));
    fields.insert("actor".to_string(), Json::from(fields));
    Json::Object(fields)
};
```

Обратите внимание, что части кода, переданные вызывающей стороной и вклеенные в результат расширения, например `"name"` и `"actor"`, сохраняют исходный цвет (черный). А другим цветом раскрашены лишь лексемы, взятые из шаблона макроса.

Теперь существуют переменная `fields` (объявленная в вызывающей программе) и отдельная от нее переменная `fields` (определенная внутри макроса). Поскольку имена разноцветные, никакой путаницы не возникает.

Если макросу действительно нужно обратиться к переменной в области видимости вызывающей программы, то вызывающая сторона должна передать имя этой переменной макросу.

(Метафора раскраски – не точное описание механизма работы гигиены. Истинный механизм даже несколько сложнее: два идентификатора считаются одинаковыми независимо от «цвета», если ссылаются на общую переменную, которая находится в области видимости макроса и вызывающей его программы. Но такие случаи в Rust редки. Если вы поняли приведенный выше пример, то знаете достаточно о гигиенических макросах.)

Вы, наверное, обратили внимание, что многие другие идентификаторы при расширении макроса раскрашиваются в один или несколько цветов, например: `Box`, `HashMap` и `Json`. Несмотря на цвет, Rust не испытывает затруднений с распознаванием этих имен типов. Дело в том, что гигиена в Rust ограничена локальными переменными и аргументами. В отношении констант, типов, методов, модулей и имен макросов Rust страдает «цветовой слепотой».

Это означает, что если наш макрос `json!` используется в модуле, где имена `Box`, `HashMap` или `Json` не находятся в области видимости, то работать он не будет. В следующем разделе мы покажем, как обойти эту проблему.

Но сначала рассмотрим случай, когда строгая гигиена мешает и с этим надо что-то делать. Предположим, что имеется много функций, содержащих такую строчку:

```
let req = ServerRequest::new(server_socket.session());
```

Каждый раз копировать и вставлять ее не хочется. Нельзя ли вместо этого использовать макрос?

```
macro_rules! setup_req {
    () => {
        let req = ServerRequest::new(server_socket.session());
    }
}

fn handle_http_request(server_socket: &ServerSocket) {
    setup_req!();  // заявляется `req`, используется `server_socket`
    ... // код, в котором используется `req`
}
```

В таком виде это не работает. Требуется, чтобы имя `server_socket` в макросе ссылалось на локальную переменную `server_socket`, объявленную в функции, а для переменной `req` – все наоборот. Но гигиена предотвращает совмещение имен переменных в макросе и в других областях видимости – даже тогда, когда оно желательно.

Решение – передать макросу все идентификаторы, которые мы планируем использовать как внутри, так и вне кода макроса:

```
macro_rules! setup_req {
    ($req:ident, $server_socket:ident) => {
        let $req = ServerRequest::new($server_socket.session());
    }
}
```

```
fn handle_http_request(server_socket: &ServerSocket) {
    setup_req!(req, server_socket);
    ... // код, в котором используется `req`
}
```

Так как `req` и `server_socket` теперь предоставляются функцией, они имеют правильный «цвет» в этой области видимости.

Из-за гигиены этот макрос немного потерял в лаконичности, но это не ошибка, а разумная мера: рассуждать о гигиенических макросах становится проще, если знать, что они не могут мухлевать с локальными переменными у вас за спиной. Поиск идентификатора `server_socket` в функции найдет все места, где он встречается, включая и вызовы макросов.

Импорт и экспорт макросов

Поскольку макросы расширяются на ранней стадии компиляции, еще до того как компилятору становится известна полная структура модулей проекта, их невозможно импортировать и экспортировать обычным способом:

- в пределах одного крейта макросы, видимые в одном модуле, автоматически видимы и в его дочерних модулях;
- чтобы экспортировать макрос из модуля «наверх» – в родительский модуль, пользуйтесь атрибутом `#[macro_use]`. Предположим, к примеру, что файл `lib.rs` выглядит следующим образом:

  ```
  #[macro_use] mod macros;
  mod client;
  mod server;
  ```

  Все макросы, определенные в модуле `macros`, импортируются в `lib.rs` и потому видимы во всех остальных частях крейта, включая модули `client` и `server`.

При работе с несколькими крейтами:

- чтобы импортировать макросы из другого крейта, снабдите атрибутом `#[macro_use]` объявление `extern crate`;
- чтобы экпортировать макросы из своего крейта, снабдите каждый открытый макрос атрибутом `#[macro_export]`.

Разумеется, любой из этих шагов означает, что ваш макрос можно вызывать из других модулей. Поэтому экспортированный макрос не должен полагаться на особенности области видимости – невозможно сказать, что встретится в той области видимости, где он будет использоваться. Даже средства из стандартной прелюдии можно замаскировать.

Вместо этого макрос должен работать с абсолютными путями ко всем именам. `macro_rules!` предоставляет для этой цели специальный тип фрагмента `$crate`. Он действует как абсолютный путь к корневому модулю крейта, в котором определен макрос. Вместо `Json` мы можем написать `$crate::Json`, и это будет работать, даже если `Json` не был импортирован. `HashMap` можно заменить на `::std::collections::HashMap` или на `$crate::macros::HashMap`. Во втором случае мы должны будем реэкспортировать `HashMap`, потому что `$crate` непригоден для доступа к закрытым возможностям крейта. На самом деле он просто расширяется во что-то типа `::jsonlib`, т. е. в обычный путь. Правила видимости остаются неизменными.

Ниже приведен окончательный вид макроса после вынесения в отдельный модуль `macros` и модификации с целью использования типа фрагмента `$crate`.

```
// macros.rs
pub use std::collections::HashMap;
pub use std::boxed::Box;
pub use std::string::ToString;

#[macro_export]
macro_rules! json {
    (null) => {
        $crate::Json::Null
    };
    ([ $( $element:tt ),* ]) => {
        $crate::Json::Array(vec![ $( json!($element) ),* ])
    };
    ({ $( $key:tt : $value:tt ),* }) => {
        {
            let mut fields = $crate::macros::Box::new(
                $crate::macros::HashMap::new());
            $( fields.insert($crate::ToString::to_string($key), json!($value)); )*
            $crate::Json::Object(fields)
        }
    };
    ($other:tt) => {
        $crate::Json::from($other)
    };
}
```

Поскольку метод `.to_string()` – часть стандартной характеристики `ToString`, то для ссылки на него мы тоже используем `$crate`, употребляя полностью квалифицированный вызов: `$crate::ToString::to_string($key)`. В нашем случае макрос будет работать и без этого, поскольку `ToString` входит в стандартную прелюдию. Но если вы вызываете методы характеристики, которая может отсутствовать в области видимости в том месте, где вызывается макрос, то лучше всего указывать полностью квалифицированное имя метода.

Предотвращение синтаксических ошибок при сопоставлении

Следующий макрос кажется вполне нормальным, но у Rust он вызывает сложности:

```
macro_rules! complain {
    ($msg:expr) => {
        println!("Complaint filed: {}", $msg);
    };
    (user : $userid:tt , $msg:expr) => {
        println!("Complaint from user {}: {}", $userid, $msg);
    };
}
```

Допустим, что он вызывается следующим образом:

```
complain!(user: "jimb", "the AI lab's chatbots keep picking on me");
```

На взгляд человека, должно быть сопоставление со вторым образцом. Но Rust сначала пробует первое правило, пытаясь сопоставить весь вход с `$msg:expr`. Тут-то и начинаются неприятности. Конечно, `user: "jimb"` не является выражением, поэтому мы получаем синтаксическую ошибку. Rust отказывается заметать ее под ковер – макросы и так-то нелегко отлаживать. Поэтому он сообщает об ошибке немедленно и прекращает компиляцию.

Если какая-нибудь другая лексема не сопоставится, Rust перейдет к следующему правилу. Фатальны лишь синтаксические ошибки, а они случаются только при попытке сопоставления с фрагментами.

Понять, в чем проблема, нетрудно: мы пытаемся сопоставить с фрагментом `$msg:expr` не в том правиле. Фрагмент и не сопоставится, потому что мы сюда и попасть-то не должны были. Вызывающей стороне нужно было другое правило. Есть два простых способа избежать этого.

Во-первых, избегайте трудно различимых правил. Мы могли бы, к примеру, изменить макрос, так чтобы все образцы начинались разными идентификаторами:

```
macro_rules! complain {
    (msg : $msg:expr) => {
        println!("Complaint filed: {}", $msg);
    };
    (user : $userid:tt , msg : $msg:expr) => {
        println!("Complaint from user {}: {}", $userid, $msg);
    };
}
```

Если первым аргументом макроса является `msg`, то мы попадаем на правило 1, если `user` – то на правило 2. В любом случае мы выберем нужное правило раньше попытки сопоставления с фрагментом.

В макросе `json!` принят другой подход: всеобъемлющее правило размещено последним.

За пределами macro_rules!

С помощью образцов макросов можно разбирать данные и посложнее, чем JSON, но, на наш взгляд, сложность быстро выходит из-под контроля.

Книга Daniel Keep et al «The Little Book of Rust Macros» (https://danielkeep.github.io/tlborm/book/README.html) – отличный справочник по программированию макросов с применением `macro_rules!`. Она написано ясно и толково, и все аспекты макрорасширения описываются подробнее, чем здесь. Представлено также несколько изобретательных приемов использования образцов `macro_rules!` как некоего эзотерического языка программирования для разбора сложных входных данных. Но на этот счет мы не испытываем особого энтузиазма. Применяйте с осторожностью.

В версии Rust 1.15 появился отдельный механизм *процедурных макросов*. Он поддерживает обобщение атрибута `#[derive]` на пользовательские характеристики, например:

```
#[derive(Copy, Clone, PartialEq, Eq, IntoJson)]
struct Money {
    dollars: u32,
    cents: u16,
}
```

Пользовательский вывод

Характеристики `IntoJson` не существует, но это и не важно: процедурный макрос может использовать это имя для вставки произвольного кода (в данном случае, вероятно, `impl From<Money> for Json { ... }`).

Слово «процедурный» в названии «процедурный макрос» означает, что макрос реализован в виде функции на Rust, а не набора декларативных правил. На момент написания этой книги процедурные макросы все еще считались новым средством, которое будет развиваться, поэтому отсылаем читателя к документации по адресу (https://doc.rust-lang.org/book/first-edition/procedural-macros.html).

Возможно, прочитав все это, вы решили, что макросы вам ненавистны. Что тогда? Альтернатива – генерация Rust-кода с помощью скрипта сборки. В документации по Cargo (http://doc.crates.io/build-script.html#case-study-code-generation) приведены пошаговые инструкции, как это сделать. Вам предстоит написать программу, которая генерирует нужный вам код на Rust, добавить строчку в файл `Cargo.toml`, чтобы эта программа запускалась как часть процесса сборки, и использовать макрос `include!`, который вставит сгенерированный код в ваш крейт.

Глава 21

Небезопасный код

Ни слабою, ни жалкою, наверно,
В устах людей я не останусь; нас
Не назовут и терпеливой; нрава
Иного я: на злобу я двумя,
А на любовь двойною отвечаю.
Все в мире дети славы таковы.

— Еврипид «Медея»[1]

Тайная отрада системного программирования заключается в том, что под каждым безопасным языком и под каждой тщательно спроектированной абстракцией бурлит водоворот диких небезопасных машинных команд и манипуляций на уровне бит. Это и к Rust относится.

Язык, представленный до сего момента, автоматически гарантирует, что в программе нет ошибок работы с памятью и гонок за данные. Для этого используются такие средства, как типы, время жизни, проверка выхода за границы диапазона и т. д. Но у такого рода автоматических рассуждений есть пределы, и существует немало ценных приемов, которые Rust не признает безопасными.

Небезопасный код позволяет сказать компилятору: «в этом случае просто поверь мне». Помечая блок или функцию как небезопасные, вы получаете возможность вызывать стандартные библиотечные функции, помеченные `unsafe`, разыменовывать небезопасные указатели, вызывать функции, написанные на других языках, например C и C++, и еще много чего. Все обычные проверки безопасности, встроенные в Rust, по-прежнему выполняются: проверка типов, времени жизни, индексов. В небезопасном коде просто разрешены некоторые дополнительные средства.

Благодаря возможности выйти за границы безопасности и реализованы самые фундаментальные средства самого языка, как обычно делается в системах, написанных на C и C++. Именно небезопасный код позволяет эффективно реализовать буфер вектора `Vec`, взаимодействовать с операционной системой из модуля `std::io`, предоставить примитивы конкурентности, находящиеся в модулях `std::thread` и `std::sync`.

В этой главе рассматриваются основы работы с небезопасными средствами:

- `unsafe`-блоки определяют границу между обыкновенным безопасным Rust-кодом и кодом, в котором используются небезопасные средства;

---

[1] Перевод И. Анненского.

- мы можем пометить функцию как `unsafe`, уведомляя тем самым вызывающую сторону о том, что она должна соблюдать дополнительные контракты во избежание неопределенного поведения;
- простые указатели и их методы дают неограниченный доступ к памяти и позволяют строить структуры данных, которые в противном случае были бы запрещены системой типов Rust;
- знакомство с определением неопределенного поведения поможет понять, почему его последствия могут быть куда серьезнее, чем просто получение неправильных результатов;
- интерфейс Rust с другими языками позволяет использовать библиотеки, написанные на других языках;
- небезопасные характеристики, как и `unsafe`-функции, определяют контракт, который должна соблюдать каждая реализация (а не каждая вызывающая сторона).

Небезопасность от чего?

В начале книги мы продемонстрировали написанную на C программу, которая «грохается» странным образом, потому что не следует одному из правил, прописанных в стандарте C. То же самое можно сделать и на Rust:

```
$ cat crash.rs
fn main() {
    let mut a: usize = 0;
    let ptr = &mut a as *mut usize;
    unsafe {
        *ptr.offset(3) = 0x7ffff72f484c;
    }
}

$ cargo build
   Compiling unsafe-samples v0.1.0
    Finished debug [unoptimized + debuginfo] target(s) in 0.44 secs
$ ../../target/debug/crash
crash: Error: .netrc file is readable by others.
crash: Remove password or make file unreadable by others.
Segmentation fault (core dumped)
$
```

Эта программа заимствует изменяемую ссылку на локальную переменную `a`, приводит ее к типу простого указателя `*mut usize`, а затем с помощью метода `offset` создает указатель на адрес в памяти, отстоящий на три слова дальше. По этому адресу находится адрес возврата функции `main`. Программа перезаписывает адрес возврата константой таким образом, что при возврате из `main` происходит нечто странное. Крах стал возможен из-за неправильного использования программой небезопасных средств – в данном случае разыменования простых указателей.

Всякое небезопасное средство вводит некий *контракт*: правила, соблюдение которых Rust не может проконтролировать автоматически, но которым тем не менее необходимо следовать, чтобы избежать *неопределенного поведения*.

Контракт выходит за рамки обычных проверок типов и времени жизни и содержит дополнительные правила, специфичные для конкретного небезопасного

средства. Обычно Rust вообще ничего не знает о контракте, все пояснения находятся только в документации. Так, в контракте типа простого указателя запрещается разыменовывать указатель, если он сдвинут дальше конца объекта исходного указателя. Выражение `*ptr.offset(3) = ...` в примере выше нарушает этот контракт. Но, как видно из протокола компиляции, Rust компилирует эту программу без ошибок: его механизмы безопасности не обнаружили нарушения. При использовании небезопасных средств ответственность за соблюдение программой контракта возлагается на программиста.

Для многих средств имеются правила, которым нужно следовать при правильном использовании, но такие правила не являются «контрактами» в приведенном выше смысле, если только в число возможных последствий не входит неопределенное поведение. Неопределенным считается поведение, которое, по твердому убеждению Rust, не может иметь места в правильной программе. Например, Rust предполагает, что вы никогда не будете перезаписывать адрес возврата функции. Код, прошедший обычный контроль безопасности Rust и соблюдающий контракты небезопасных средств, ничего подобного делать не может. Поскольку показанная выше программа нарушает контракт простого указателя, ее поведение не определено, и она съезжает с катушек.

Если ваш код демонстрирует неопределенное поведение, значит, вы нарушили свою часть сделки с Rust, и Rust отказывается нести ответственность за последствия. Печать не относящегося к делу сообщения из недр системной библиотеки и аварийное завершение – одно из возможных последствий, вручение контроля над своим компьютером злоумышленнику – другое. Результат может измениться при переходе на следующую версию Rust, причем без всяких предупреждений. Но иногда у неопределенного поведения нет видимых последствий. Например, если функция `main` в примере выше не возвращает управления (потому что вызывает `std::process::exit` для преждевременного завершения программы), то измененный адрес возврата не играет роли.

Использовать небезопасные средства можно только внутри `unsafe`-блока или `unsafe`-функции; ниже мы опишем то и другое. Благодаря этому использовать небезопасные средства невольно становится труднее: требуя помечать блок или функции словом `unsafe`, Rust заставляет подтвердить, что вы осознаете наличие дополнительных правил, которые ваш код обязан соблюдать.

Unsafe-блоки

`Unsafe`-блок выглядит как обыкновенный блок Rust, которому предшествует ключевое слово `unsafe`, но в этом блоке можно использовать небезопасные средства:

```
unsafe {
    String::from_utf8_unchecked(ascii)
}
```

Не будь ключевого слова `unsafe` в начале блока, Rust возражал бы против использования `unsafe`-функции `from_utf8_unchecked`. А так все возражения снимаются.

Как и в случае обыкновенного блока Rust, значением `unsafe`-блока является значение его последнего выражения или `()`, если такового не существует. В данном случае значением будет значение функции `String::from_utf8_unchecked`.

В `unsafe`-блоках в вашем распоряжении оказываются четыре дополнительных средства:

- можно вызывать `unsafe`-функции. Любая `unsafe`-функция должна специфицировать контракт, зависящий от ее назначения;
- можно разыменовывать простые указатели. В безопасном коде простые указатели можно передавать из одного места в другое, сравнивать и создавать путем преобразования из ссылок (или даже из целых чисел), но только в небезопасном коде с их помощью можно получать доступ к памяти. Мы подробно рассмотрим простые указатели и объясним, как пользоваться ими безопасно, в разделе «Простые указатели» ниже;
- можно обращаться к изменяемым статическим переменным. Как было сказано в разделе «Глобальные переменные» главы 19, Rust не знает точно, когда в потоке используются изменяемые статические переменные, поэтому их контракт требует, чтобы любой доступ к ним был синхронизирован;
- можно обращаться к функциям и переменным, объявленным посредством интерфейса с иноязычными функциями. Они считаются небезопасными, даже если неизменяемы, поскольку видны коду, написанному на другом языке, который может и не соблюдать принятых в Rust правил безопасности.

Разрешение использовать небезопасные средства только в `unsafe`-блоках на самом деле не мешает вам делать все что угодно. Вполне возможно просто воткнуть `unsafe`-блок в свой код и идти дальше. Смысл этого правила – в том, чтобы привлечь внимание человека к коду, безопасность которого Rust не может гарантировать:

- вы не сможете случайно воспользоваться небезопасным средством, а затем обнаружить, что должны были соблюдать контракт, о существовании которого даже не подозревали;
- `Unsafe`-блок вызывает пристальный интерес со стороны рецензентов. В некоторых проектах существуют даже соответствующие средства автоматизации, помечающие изменения кода, затрагивающие `unsafe`-блоки, специальным флагом для привлечения внимания;
- задумываясь о написании `unsafe`-блока, вы можете спросить себя, а так ли уж это необходимо в данной задаче. Если всё делается ради повышения производительности, то существуют ли измерения, доказывающие, что это действительно узкое место? Быть может, нужного результата можно достичь и с помощью безопасных средств Rust?

Пример: эффективный тип ASCII-строки

Ниже приведено определение строкового типа `Ascii`, гарантирующего, что в нем могут храниться только символы кода ASCII. В реализации используется небезопасное средство преобразования в `String` с нулевыми издержками:

```
mod my_ascii {
    use std::ascii::AsciiExt; // for u8::is_ascii

    /// Строка в кодировке ASCII.
    #[derive(Debug, Eq, PartialEq)]
    pub struct Ascii(
        // Должна содержать только ASCII-текст:
```

```
        // байты в диапазоне от `0` до `0x7f`.
        Vec<u8>
    );

    impl Ascii {
        /// Создает значение типа `Ascii` из ASCII-текста в `bytes`. Возвращает
        /// ошибку `NotAsciiError`, если `bytes` содержит хотя бы один символ
        /// вне диапазона ASCII.
        pub fn from_bytes(bytes: Vec<u8>) -> Result<Ascii, NotAsciiError> {
            if bytes.iter().any(|&byte| !byte.is_ascii()) {
                return Err(NotAsciiError(bytes));
            }
            Ok(Ascii(bytes))
        }
    }

    // Если преобразование завершается неудачно, возвращаем вектор,
    // который не смогли преобразовать. Должна реализовывать `std::error::Error`,
    // но для краткости это опущено.
    #[derive(Debug, Eq, PartialEq)]
    pub struct NotAsciiError(pub Vec<u8>);

    // Безопасное эффективное преобразование, реализованное небезопасным кодом.
    impl From<Ascii> for String {
        fn from(ascii: Ascii) -> String {
            // Если в этом модуле нет ошибок, то он безопасен, поскольку
            // допустимый ASCII-текст является также допустимым UTF-8.
            unsafe { String::from_utf8_unchecked(ascii.0) }
        }
    }
    ...
}
```

Самое интересное в этом модуле – определение типа `Ascii`. Тип помечен ключевым словом `pub`, чтобы он был виден вне модуля `my_ascii`. Но элемент `Vec<u8>` этого типа *не* является открытым, т. е. только сам модуль `my_ascii` может сконструировать значение типа `Ascii` или сослаться на его элемент. Таким образом, модуль полностью контролирует, что может происходить в этом элементе. Коль скоро открытые конструкторы и методы гарантируют, что значения типа `Ascii` корректны сразу после создания и остаются такими на протяжении всей своей жизни, то остальная программа не может нарушить этого правила. И действительно, открытый конструктор `Ascii::from_bytes` тщательно проверяет переданный ему вектор, прежде чем соглашается сконструировать из него значение типа `Ascii`. Для краткости мы опустили остальные методы, но можете представить себе набор методов работы с текстом, которые гарантируют, что значения типа `Ascii` всегда содержат допустимый ASCII-текст, точно так же, как методы `String` гарантируют, что строка всегда состоит из байтов, образующих правильную последовательность символов UTF-8.

Такая схема позволяет реализовать характеристику `From<Ascii>` для типа `String` очень эффективно. Небезопасная функция `String::from_utf8_unchecked` принимает байтовый вектор и строит из него строку `String`, не проверяя, является ли содержимое допустимым текстом в кодировке UTF-8; контракт функции возлагает ответственность за это на вызывающую сторону. По счастью, требования,

предъявляемые типом Ascii, – в точности то, что необходимо для удовлетворения контракта функции from_utf8_unchecked. Как было объяснено в разделе «UTF-8» главы 17, любой блок ASCII-текста является допустимым текстом в UTF-8, поэтому внутреннее поле Vec<u8>, хранящееся в типе Ascii, безо всяких изменений может использоваться в качестве буфера String.

Имея эти определения, мы можем написать:

```
use my_ascii::Ascii;

let bytes: Vec<u8> = b"ASCII and ye shall receive".to_vec();

// Этот вызов не влечет за собой ни выделение памяти, ни копирование текста,
// а только его просмотр.
let ascii: Ascii = Ascii::from_bytes(bytes)
    .unwrap(); // Мы знаем, что эти байты правильны.

// Стоимость этого вызова нулевая: ни выделения памяти, ни копирования, ни просмотра.
let string = String::from(ascii);

assert_eq!(string, "ASCII and ye shall receive");
```

Для использования типа Ascii не нужно никаких unsafe-блоков. Мы реализовали безопасный интерфейс с помощью небезопасных операций и организовали все так, что их контракты зависят только от кода самого модуля, а не от поведения пользователя.

Тип Ascii – не более чем обертка вокруг типа Vec<u8>, скрытого внутри модуля, который налагает дополнительные ограничения на содержимое вектора. Тип такого рода называется *неотипом* (newtype), этот паттерн часто встречается Rust. Встроенный в Rust тип String определен таким же образом, только его содержимое должно быть текстом в кодировке UTF-8, а не ASCII. Вот определение типа String, взятое из стандартной библиотеки:

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

Если отвлечься от того, что привносят в общую картину типы Rust, то на машинном уровне представления неотипа и его элемента в памяти идентичны, поэтому для конструирования неотипа машинные команды вообще не нужны. В методе Ascii::from_bytes выражение Ascii(bytes) просто постулирует, что представление Vec<u8> с этого момента содержит значение типа Ascii. Аналогично метод String::from_utf8_unchecked, вероятно, тоже не требует никаких машинных команд в случае встраивания: просто Vec<u8> теперь рассматривается как String.

Unsafe-функции

Определение unsafe-функции выглядит так же, как определение обыкновенной функции с предшествующим ключевым словом unsafe. Тело unsafe-функции автоматически считается unsafe-блоком.

Unsafe-функции разрешено вызывать только из unsafe-блоков. Пометка функции ключевым словом unsafe предупреждает пользователей о том, что они должны соблюдать ее контракт во избежание неопределенного поведения.

Вот, например, новый конструктор введенного выше типа `Ascii`, который строит `Ascii` из байтового вектора, не проверяя, что он содержит только ASCII-символы:

```
// Этот код должен находиться внутри модуля `my_ascii`.
impl Ascii {
    /// Конструирует значение типа `Ascii` из вектора `bytes`, не проверяя,
    /// что `bytes` содержит только ASCII-символы.
    ///
    /// Этот конструктор не может завершиться неудачно и возвращает значение
    /// типа `Ascii` непосредственно, а не `Result<Ascii, NotAsciiError>`,
    /// как конструктор `from_bytes`.
    ///
    /// # Безопасность
    ///
    /// Вызывающая сторона должна гарантировать, что `bytes` содержит только
    /// ASCII-символы: байты, не большие 0x7f. В противном случае результат
    /// не определен.
    pub unsafe fn from_bytes_unchecked(bytes: Vec<u8>) -> Ascii {
        Ascii(bytes)
    }
}
```

Предположительно код, вызывающий метод `Ascii::from_bytes_unchecked`, уже откуда-то знает, что вектор содержит только ASCII-символы, поэтому проверка, на которой настаивает `Ascii::from_bytes`, была бы лишней тратой времени, а вызывающей стороне пришлось бы писать код обработки ошибок, которые заведомо не могут произойти. Конструктор `Ascii::from_bytes_unchecked` позволяет в этом случае обойтись без проверок и обработки ошибок.

Но ведь в комментарии, предшествующем определению типа `Ascii`, говорится: «Никакой код в этом модуле не допускает появления не-ASCII-байтов в значении типа `Asci"`. А разве не это делает новый конструктор `from_bytes_unchecked`?

Не совсем: `from_bytes_unchecked` выполняет свои обязательства, делегируя их вызывающей стороне, которая должна соблюдать его контракт. Именно из-за наличия контракта эта функция помечена как `unsafe`: хотя сама функция не производит никаких небезопасных операций, вызывающий ее код во избежание неопределенного поведения должен соблюдать правила, которые Rust не может проконтролировать автоматически.

Действительно ли нарушение контракта `Ascii::from_bytes_unchecked` может привести к неопределенному поведению? Да, мы можем следующим образом сконструировать строку `String`, содержащую текст не в кодировке UTF-8:

```
// Представьте, что этот вектор получен в результате какого-то сложного процесса,
// который должен порождать ASCII-символы. Но что-то пошло наперекосяк!
let bytes = vec![0xf7, 0xbf, 0xbf, 0xbf];

let ascii = unsafe {
    // Контракт этой небезопасной функции нарушен, поскольку
    // `bytes` содержит байты, не принадлежащие ASCII.
    Ascii::from_bytes_unchecked(bytes)
};

let bogus: String = ascii.into();
```

```
// Теперь `bogus` содержит текст не в кодировке UTF-8. Уже самый первый символ
// не является допустимой кодовой позицией Юникода.
assert_eq!(bogus.chars().next().unwrap() as u32, 0x1fffff);
```

Этот пример иллюстрирует два важных факта, касающихся ошибок и небезопасного кода:

- **ошибки, имевшие место до `unsafe`-блока, могут нарушить контракты.** Приводит ли небезопасный блок к неопределенному поведению, может зависеть не только от кода в самом блоке, но и от кода, поставляющего ему входные данные. Все, от чего зависит соблюдение контрактов `unsafe`-кодом, имеет прямое отношение к безопасности. Преобразование из `Ascii` в `String`, основанное на методе `String::from_utf8_unchecked`, корректно определено, только если все прочие части модуля правильно поддерживают инварианты типа `Ascii`;
- **последствия нарушения контракта могут проявиться после выхода из `unsafe`-блока.** Неопределенное поведение, ставшее результатом нарушения контракта небезопасного средства, часто возникает не в самом `unsafe`-блоке. Появление некорректной строки, как в примере выше, может привести к проблемам гораздо позже в ходе выполнения программы.

По существу, встроенные в Rust механизмы контроля типов, заимствования и прочие статические проверки анализируют вашу программу и пытаются доказать, что в ней невозможно неопределенное поведение. Если компиляция программы завершилась успешно, значит, найти такое доказательство удалось. `Unsafe`-блок – это пробел в доказательстве, вы говорите Rust: «Этот код правилен, поверь мне». Истинность вашего заверения может зависеть от любой части программы, влияющей на то, что происходит внутри `unsafe`-блока, а последствия его ложности могут проявиться в любом месте, на которое влияет `unsafe`-блок. Наличие ключевого слова `unsafe` – напоминание о том, что вы не получаете всех преимуществ, которые дают встроенные в язык проверки безопасности.

Если есть выбор, то, безусловно, следует предпочесть создание безопасных интерфейсов без контрактов. Работать с ними гораздо проще, т. к. пользователи могут рассчитывать на то, что механизмы контроля безопасности Rust проверят отсутствие неопределенного поведения в их коде. Даже если в вашей реализации используются небезопасные средства, лучше, когда для соблюдения их контрактов используется только то, что вы можете гарантировать сами, применяя типы, времена жизни и систему модулей Rust и не перекладывая ответственность на вызывающую сторону.

К сожалению, на практике нередко встречаются небезопасные функции, в документации по которым контракты не описываются. Ожидается, что вы выведете правила сами, основываясь на своем опыте и знаниях о поведении кода. Если вам когда-нибудь доводилось недоумевать, правильно ли вы используете API, написанный на C или C++, то вы понимаете, о чем мы.

Unsafe-блок или unsafe-функция?

Может возникнуть вопрос, что лучше: воспользоваться `unsafe`-блоком или пометить всю функцию как небезопасную. Мы рекомендуем сначала решить насчет функции:

- если имеется возможность использовать функцию неправильно, так что она компилируется, но все равно может приводить к неопределенному поведению, то следует пометить ее как `unsafe`. Правила корректного использования функции прописаны в ее контракте; именно существование контракта и делает функцию небезопасной;
- в противном случае функция безопасна: никакой вызов не может привести к неопределенному поведению. Тогда ее не следует помечать как `unsafe`.

Используются ли в теле функции небезопасные средства, не имеет значения: важно лишь наличие контракта. Выше мы продемонстрировали небезопасную функцию, в которой не используются никакие небезопасные средства, и безопасную функцию, в которой используются небезопасные средства.

Не помечайте безопасную функцию ключевым словом `unsafe` только потому, что в ее теле используются небезопасные средства. Из-за этого функцию становится труднее использовать, и это будет смущать читателей, которые ожидают (и правильно делают) обнаружить где-то описание контракта. Вместо этого используйте `unsafe`-блок, даже если он совпадает со всем телом функции.

Неопределенное поведение

Выше мы сказали, что «неопределенным считается поведение, которое, по твердому убеждению Rust, не может иметь места в правильной программе». Это странный поворот мысли, особенно если учесть, что по опыту работы с другими языками мы знаем, что такое поведение все же иногда *имеет место* – по случайности. Почему же эта концепция полезна при определении обязательства небезопасного кода?

Компилятор – это переводчик с одного языка программирования на другой. Компилятор Rust принимает на входе программу, написанную на Rust, и транслирует ее в эквивалентную программу на машинном языке. Но что мы имеем в виду, говоря, что две программы на разных языках «эквивалентны»?

К счастью, этот вопрос для программистов проще, чем для лингвистов. Обычно мы говорим, что две программы эквивалентны, если они всегда демонстрируют одинаковое видимое поведение при выполнении: выполняют одни и те же системные вызовы, взаимодействуют с иноязычными библиотеками эквивалентными способами и т. д. Это нечто вроде теста Тьюринга для программ: если вы не можете сказать, с чем взаимодействуете – с оригинальной или с транслированной программой, – то они эквивалентны.

Теперь рассмотрим следующий код:

```
let i = 10;
very_trustworthy(&i);
println!("{}", i * 100);
```

Даже ничего не зная об определении функции `very_trustworthy`, мы видим, что она получает только разделяемую ссылку на `i`, поэтому вызов не может изменить значения `i`. Поскольку макросу `println!` всегда передается значение `1000`, Rust может транслировать этот код в машинный язык, как если бы было написано:

```
very_trustworthy(&10);
println!("{}", 1000);
```

Видимое поведение этой преобразованной версии такое же, как у исходной программы, но она чуточку быстрее.

Однако говорить о производительности этой версии имеет смысл, только если мы согласимся, что ее семантика не отличается от исходной. А что, если бы функция `very_trustworthy` была определена следующим образом:

```
fn very_trustworthy(shared: &i32) {
    unsafe {
        // Преобразовать разделяемую ссылку в изменяемый указатель.
        // Это неопределенное поведение.
        let mutable = shared as *const i32 as *mut i32;
        *mutable = 20;
    }
}
```

Этот код нарушает правила для разделяемых ссылок: он изменяет значение `i` на `20`, хотя одно должно быть заморожено, поскольку `i` заимствована для разделения. В результате преобразование, которому была подвергнута вызывающая сторона, становится как нельзя более видимым: если Rust преобразует код, то будет напечатано `1000`, а если оставит его как есть и будет использовать новое значение `i`, то будет напечатано `2000`. Нарушение правил работы с разделяемыми ссылками внутри `very_trustworthy` означает, что теперь разделяемые ссылки ведут себя не так, как ожидают пользователи этой функции.

Такого рода проблемы возникают чуть ли не для каждого преобразования, которое мог бы предпринять Rust. Даже простое встраивание функции в месте вызова предполагает, среди прочего, что когда вызываемая функция завершится, управление будет передано туда, откуда она вызвана. Но мы начали эту главу с примера функции, которая нарушает даже это предположение.

Ни Rust и никакой другой язык не смогут установить, что некое преобразование программы сохраняет ее семантику, если нельзя доверять, что фундаментальные средства языка ведут себя, как задумано. А так это или не так, зависит не только от непосредственно анализируемого кода, но и от других, возможно далеко отстоящих, частей программы. Чтобы сделать с кодом хоть что-то, Rust должен предполагать, что все прочие части программы ведут себя корректно.

Ниже перечислены правила Rust для корректных программ.

- Программа не должна читать из неинициализированной памяти.
- Программа не должна создавать недопустимых значений примитивных типов:
  - ссылки или боксы, равные `null`;
  - значения типа `bool`, отличные от 0 и 1;
  - значения `enum` с недопустимыми дискриминантными элементами;
  - значения типа `char`, не являющиеся допустимыми несуррогатными кодовыми позициями Юникода;
  - значения типа `str`, содержащие что-то, кроме текста в кодировке UTF-8.
- Должны соблюдаться правила работы со ссылками, описанные в главе 5. Ссылка не должна жить дольше объекта ссылки; разделяемый доступ допускает только чтение; изменяемый доступ является монопольным.
- Программа не должна разыменовывать нулевых, неправильно выровненных или висячих указателей.

- Программа не должна использовать указателя для доступа к памяти вне области, ассоциированной с этим указателем. Подробное разъяснение этого правила см. в разделе «Безопасное разыменование простых указателей» ниже.
- В программе не должно быть гонок за данные. Гонка за данные возникает, когда два потока обращаются к одной и той же ячейке памяти без синхронизации и хотя бы одна операция доступа – запись.
- Программа не должна раскручивать вызов, произведенный из функции на другом языке (см. раздел «Раскрутка стека» главы 7).
- Программа должна соблюдать контракты стандартных библиотечных функций.

Эти правила – всё, что Rust предполагает в процессе оптимизации программы и трансляции ее на машинный язык. Неопределенное поведение – это попросту нарушение какого-то из этих правил. Потому-то мы и говорим, что неопределенное поведение, по твердому убеждению Rust, не может иметь места в правильной программе: такое допущение необходимо, если мы надеемся заключить, что откомпилированная программа – достоверный перевод исходного кода.

Небезопасные характеристики

Небезопасной называется такая характеристика, что Rust не может проверить ее контракт сам и не может принудить реализующих его типов соблюдать этот контракт во избежание неопределенного поведения. Для реализации небезопасной характеристики саму реализацию следует пометить как небезопасную. Вы обязаны понимать контракт характеристики и гарантировать его выполнение в своем типе.

Функция, в которой на параметрические типы наложено ограничение, связанное с небезопасной характеристикой, обычно сама пользуется небезопасными средствами, и выполнение ей собственных контрактов зависит от контракта небезопасной характеристики. Некорректная реализация характеристики может стать причиной неопределенного поведения такой функции.

Классическими примерами небезопасных характеристик служат `std::marker::Send` и `std::marker::Sync`. Они не определяют никаких методов, поэтому тривиально реализуются для любого типа. Но у них есть контракты: `Send` требует, чтобы реализующее значение можно было безопасно передать во владение другому потоку, а `Sync` – чтобы такие значения можно было безопасно разделять между потоками с помощью разделяемых ссылок. Например, реализация `Send` для неподходящего типа могла бы сделать `std::sync::Mutex` не защищенным от гонки за данные.

В стандартной библиотеке имеется простой пример небезопасной характеристики, `core::nonzero::Zeroable`, для типов, которые можно безопасно инициализировать, присвоив всем байтам значение 0. Очевидно, что против обнуления `usize` нет никаких возражений, но обнуление `&T` дало бы нулевую ссылку, что привело бы к краху программы при разыменовании. Для обнуляемых типов возможны некоторые оптимизации, например массив значений такого типа можно быстро инициализировать, вызвав функцию `std::mem::write_bytes` (эквивалент `memset` в Rust) или использовав вызовы операционной системы, которые выделяют заполненные нулями страницы памяти. (В версии Rust 1.17 характеристика `Zeroable`

экспериментальная, поэтому в будущих версиях может быть изменена или удалена, но это хороший, простой и реальный пример.)

`Zeroable` – типичная маркерная характеристика, в ней нет ни методов, ни ассоциированных типов:

```
pub unsafe trait Zeroable {}
```

Ее реализации для подходящих типов тоже тривиальны:

```
unsafe impl Zeroable for u8 {}
unsafe impl Zeroable for i32 {}
unsafe impl Zeroable for usize {}
// и для всех остальных целых типов
```

Имея такие определения, мы можем написать функцию, которая быстро выделяет память для вектора заданной длины с элементами типа `Zeroable`:

```
#![feature(nonzero)]  // `Zeroable` разрешена

extern crate core;
use core::nonzero::Zeroable;

fn zeroed_vector<T>(len: usize) -> Vec<T>
    where T: Zeroable
{
    let mut vec = Vec::with_capacity(len);
    unsafe {
        std::ptr::write_bytes(vec.as_mut_ptr(), 0, len);
        vec.set_len(len);
    }
    vec
}
```

Сначала эта функция создает пустой вектор `Vec` требуемой емкости, а затем вызывает `write_bytes`, чтобы заполнить буфер нулями. (Функция `write_bytes` интерпретирует `len` как число элементов типа `T`, а не байтов, так что этот вызов заполняет весь буфер.) Метод `set_len` изменяет длину вектора, не производя никаких действий с буфером; это небезопасно, потому что мы должны быть уверены, что расширенный буфер содержит правильно инициализированные значения типа `T`. Но это как раз то, что гарантирует ограничение `T: Zeroable`: блок нулевых байтов является допустимым значением типа `T`. Поэтому наше использование `set_len` безопасно.

Применим на практике:

```
let v: Vec<usize> = zeroed_vector(100_000);
assert!(v.iter().all(|&u| u == 0));
```

Очевидно, что характеристика `Zeroable` должна быть небезопасной, потому что реализация, не соблюдающая ее контракта, может привести к неопределенному поведению:

```
struct HoldsRef<'a>(&'a mut i32);

unsafe impl<'a> Zeroable for HoldsRef<'a> { }

let mut v: Vec<HoldsRef> = zeroed_vector(1);
*v[0].0 = 1;   // авария: разыменование нулевого указателя
```

Rust компилирует этот код без ошибок, он не знает, в чем смысл `Zeroable`, и потому не может предупредить нас, что она реализуется для неподходящего типа. Как и для любого другого небезопасного средства, вы должны разобраться в контракте и придерживаться его.

Отметим, что небезопасный код не должен зависеть от того, правильно ли реализованы обыкновенные безопасные характеристики. Предположим, к примеру, что реализация характеристики `std::hash::Hasher` просто возвращает случайный хеш-код, никак не связанный с хешируемым значением. Эта характеристика требует, чтобы повторное хеширование одной и той же битовой комбинации давало один и тот же хеш-код, но данная реализация этому требованию не удовлетворяет, она откровенно неправильная. Но поскольку характеристика `Hasher` не является небезопасной, никакой небезопасный код не должен приводить к неопределенному поведению при использовании такого алгоритма хеширования. Тип `std::collections::HashMap` старательно написан с учетом контрактов небезопасных средств, которые в нем используются, вне зависимости от того, как ведет себя алгоритм хеширования. Конечно, таблица будет работать неправильно: поиск не будет находить записей, записи могут случайным образом появляться и исчезать. Но неопределенного поведения не будет.

Простые указатели

Простым указателем в Rust называется указатель без ограничений. Простые указатели можно использовать для построения различных структур, которые с помощью одних лишь контролируемых указателей не создать, например двусвязных списков и произвольных графов объектов. Но из-за такой гибкости простых указателей Rust не может сказать, безопасно они используются или нет, поэтому разыменовывать их можно только в `unsafe`-блоках.

Простые указатели, по существу, эквивалентны указателям в C и C++, поэтому полезны также для взаимодействия с кодом, написанным на этих языках.

Существуют два вида простых указателей:

- `*mut T` – простой указатель на `T`, позволяющий модифицировать объект, на который указывает;
- `*const T` – простой указатель на `T`, позволяющий только читать объект, на который указывает.

(Не существует типа `*T` без модификатора, нужно указать либо `const`, либо `mut`.)

Простой указатель можно создать преобразованием ссылки и разыменовать с помощью оператора `*`:

```
let mut x = 10;
let ptr_x = &mut x as *mut i32;

let y = Box::new(20);
let ptr_y = &*y as *const i32;

unsafe {
    *ptr_x += *ptr_y;
}
assert_eq!(x, 30);
```

В отличие от боксов и ссылок, простые указатели могут быть нулевыми, как NULL в C или nullptr в C++:

```
fn option_to_raw<T>(opt: Option<&T>) -> *const T {
    match opt {
        None => std::ptr::null(),
        Some(r) => r as *const T
    }
}

assert!(!option_to_raw(Some(&("pea", "pod"))).is_null());
assert_eq!(option_to_raw::<i32>(None), std::ptr::null());
```

В этом примере нет unsafe-блоков: создание простых указателей, их передача функциям и сравнение между собой – безопасные операции. Небезопасно только разыменование простого указателя.

Простой указатель на безразмерный тип является толстым указателем, как и соответствующая ссылка или значение типа Box. Указатель *const [u8] включает наряду с адресом и длину, а объект характеристики указательного типа *mut std::io::Write несет с собой таблицу виртуальных функций.

Хотя Rust неявно разыменовывает безопасные указательные типы в различных ситуациях, разыменование простого указателя должно быть явным:

- оператор . не будет неявно разыменовывать простой указатель, вы должны написать (*raw).field или (*raw).method(...);
- простые указатели не реализуют характеристику Deref, поэтому к ним неприменимы Deref-преобразования;
- операторы ==, < и им подобные сравнивают простые указатели, как адреса: два простых указателя равны, если они указывают на один и тот же адрес в памяти. Аналогично хеширование простого указателя означает хеширование адреса, на который тот указывает, а не значения, хранящегося по этому адресу;
- форматные характеристики, в т. ч. std::fmt::Display, автоматически следуют по ссылкам, но простых указателей вообще не обрабатывают. Исключениями являются характеристики std::fmt::Debug и std::fmt::Pointer, которые показывают простые указатели в виде шестнадцатеричных адресов, не разыменовывая.

В отличие от C и C++, в Rust оператор + неприменим к простым указателям, но для арифметических операций с указателями можно использовать их методы offset и wrapping_offset. Не существует стандартного способа найти расстояние между двумя указателями, как оператор - в C и C++, но можно реализовать его самостоятельно:

```
fn distance<T>(left: *const T, right: *const T) -> isize {
    (left as isize - right as isize) / std::mem::size_of::<T>() as isize
}

let trucks = vec!["garbage truck", "dump truck", "moonstruck"];
let first = &trucks[0];
let last = &trucks[2];
assert_eq!(distance(last, first), 2);
assert_eq!(distance(first, last), -2);
```

Хотя параметрами функции `distance` являются простые указатели, мы можем передавать ей ссылки – Rust неявно преобразует ссылки в простые указатели (но, конечно, не наоборот).

Оператор `as` допускает почти любое разумное преобразование из ссылки в простой указатель или между двумя типами простых указателей. Однако иногда сложное преобразование приходится разбивать на несколько шагов, например:

```
&vec![42_u8] as *const String  // ошибка: неверное преобразование
&vec![42_u8] as *const Vec<u8> as *const String;  // так можно
```

Отметим, что `as` не преобразует простых указателей в ссылки. Такие преобразования были бы небезопасны, а оператор `as` должен оставаться безопасной операцией. Вместо этого необходимо разыменовать простой указатель (в `unsafe`-блоке), а затем заимствовать получившееся значение.

Делая это, будьте очень осторожны: созданная таким способом ссылка имеет неограниченное время жизни, т. е. будет существовать неограниченно долго, поскольку простой указатель не дает Rust никакой информации для более точного решения. В разделе «Безопасный интерфейс к libgit2» ниже в этой главе мы на нескольких примерах покажем, как правильно ограничить время жизни.

У многих типов имеются методы `as_ptr` и `as_mut_ptr`, которые возвращают простой указатель на значение. Например, в случае срезок массивов и строк возвращается указатель на первый элемент, а некоторые итераторы возвращают указатель на следующий порождаемый объект. Владеющие указательные типы, например `Box`, `Rc` и `Arc`, обладают методами `into_raw` и `from_raw`, которые выполняют преобразование в простой указатель и обратно. В контрактах некоторых из этих методов содержатся неожиданные требования, поэтому внимательно читайте документацию.

Простой указатель можно построить также преобразованием целого числа, хотя доверять можно только тем целым числам, которые можно получить из указателя. В примере `RefWithFlag` ниже простые указатели используются подобным образом.

В отличие от ссылок, простые указатели не реализуют ни `Send`, ни `Sync`. А следовательно, этих характеристик не реализует никакой тип, содержащий простые указатели. В разделении или передаче простых указателей между потоками нет ничего принципиально небезопасного, в конце концов, куда бы они ни попали, для их разыменования нужен `unsafe`-блок. Но, учитывая, какую роль обычно играют простые указатели, проектировщики языка решили, что такое поведение по умолчанию будет полезнее. Как реализовать характеристики `Send` и `Sync` самостоятельно, мы обсуждали в разделе «Небезопасные характеристики» выше.

Безопасное разыменование простых указателей

Приведем несколько основанных на здравом смысле рекомендаций по безопасному использованию простых указателей:

- разыменование нулевых или висячих указателей является неопределенным поведением, как и обращение к неинициализированной памяти или к значениям, покинувшим область видимости;
- разыменование указателей, не выровненных в соответствии с типом объекта, на который указывают, – неопределенное поведение;
- заимствовать ссылки на разыменованный простой указатель можно только при соблюдении правил безопасной работы со ссылками, описанных в гла-

ве 5: ссылка не должна жить дольше объекта ссылки; разделяемый доступ допускает только чтение; изменяемый доступ является монопольным (это правило легко случайно нарушить, т. к. простые указатели часто применяются для создания структур данных с нестандартным разделением или владением);

- использовать объект указателя можно, только если это корректное значение своего типа. Например, необходимо гарантировать, что разыменование `*const char` дает допустимую несуррогатную кодовую позицию Юникода;
- методы `offset` и `wrapping_offset` простых указателей можно использовать только для указания на байты в пределах переменной или выделенной в куче области памяти, на которую указывает исходный указатель, либо на первый байт вне этой области.

  При использовании арифметики указателей путем преобразования указателя в целое число, выполнения арифметической операции с этим числом и преобразования результата обратно в указатель результат должен быть указателем, который можно было бы получить, применяя правила, действующие для метода `offset`;
- если производится присваивание объекту, на который указывает простой указатель, то не должны нарушаться инварианты типа, частью которого является этот объект. Например, если `*mut u8` указывает на байт строки `String`, то сохранять в этом байте можно только значения, оставляющие содержимое строки допустимым текстом в кодировке UTF-8.

Если оставить в стороне правила заимствования, то это, по существу, те же правила, которые надо соблюдать при работе с указателями в C и C++.

Причина, по которой не следует нарушать инварианты типа, должна быть понятна. Многие стандартные типы Rust реализованы с помощью небезопасного кода, но тем не менее предоставляют безопасные интерфейсы в предположении, что действуют все предусмотренные в Rust проверки безопасности, система модулей и правила видимости. Использование простых указателей для обхода этих защитных мер может привести к неопределенному поведению.

Полный и точный контракт для простых указателей сформулировать нелегко, и, возможно, он еще не раз изменится в процессе эволюции языка. Но соблюдение изложенных выше принципов позволит оставаться на безопасной территории.

Пример: RefWithFlag

Ниже приведен пример классического[1] трюка, основанного на манипуляциях с битами и возможного благодаря простым указателям. Но трюк этот обернут полностью безопасным типом Rust. В следующем модуле определен тип `RefWithFlag<'a, T>`, в котором хранятся значения типа `&'a T` и `bool`, упакованные в кортеж `(&'a T, bool)`, и тем не менее он занимает всего одно машинное слово, а не два. Такая техника часто используется в сборщиках мусора и виртуальных машинах, где некоторые типы – например, тип, представляющий объект, – встречаются в таком изобилии, что добавление даже одного слова к каждому значению сильно увеличивает потребление памяти.

[1] Классического в языке, из которого мы пришли.

```
mod ref_with_flag {
    use std::marker::PhantomData;
    use std::mem::align_of;

    /// `&T` и `bool`, упакованные в одно слово.
    /// Тип `T` должен быть выровнен хотя бы на границу двух байтов.
    ///
    /// Если вы относитесь к числу программистов, которые и хотели бы
    /// заимствовать младший бит указателя, да не знали как, то теперь можете
    /// сделать это безопасно!
    /// ("Но так совсем не интересно...")
    pub struct RefWithFlag<'a, T: 'a> {
        ptr_and_bit: usize,
        behaves_like: PhantomData<&'a T> // не занимает места
    }

    impl<'a, T: 'a> RefWithFlag<'a, T> {
        pub fn new(ptr: &'a T, flag: bool) -> RefWithFlag<T> {
            assert!(align_of::<T>() % 2 == 0);
            RefWithFlag {
                ptr_and_bit: ptr as *const T as usize | flag as usize,
                behaves_like: PhantomData
            }
        }

        pub fn get_ref(&self) -> &'a T {
            unsafe {
                let ptr = (self.ptr_and_bit & !1) as *const T;
                &*ptr
            }
        }

        pub fn get_flag(&self) -> bool {
            self.ptr_and_bit & 1 != 0
        }
    }
}
```

В этом коде используется тот факт, что многие типы необходимо размещать по четному адресу в памяти: поскольку младший бит четного адреса всегда равен нулю, мы можем сохранить в нем все что угодно, а затем надежно реконструировать исходный адрес, просто замаскировав младший бит. Не все типы подходят для такого трюка; например, типы `u8` и `(bool, [i8; 2])` можно разместить по любому адресу. Но мы можем проверить выравнивание типа при конструировании и отвергнуть неподходящие типы.

Тип `RefWithFlag` используется следующим образом:

```
use ref_with_flag::RefWithFlag;

let vec = vec![10, 20, 30];
let flagged = RefWithFlag::new(&vec, true);
assert_eq!(flagged.get_ref()[1], 20);
assert_eq!(flagged.get_flag(), true);
```

Конструктор `RefWithFlag::new` принимает ссылку и булево значение, проверяет, что ссылка имеет подходящий тип, а затем преобразует ее в простой указатель и далее в `usize`. Тип `usize` по определению достаточно велик для хранения указателя на любом процессоре, для которого мы компилируем код, так что преобразование простого указателя в `usize` и обратно определено корректно. Мы точно знаем, что получившееся значение типа `usize` четно, поэтому можем с помощью оператора `|` поразрядно сложить его с булевым значением, предварительно преобразованным в 0 или 1.

Метод `get_flag` извлекает из `RefWithFlag` булеву компоненту. Это просто: нужно лишь наложить маску на младший бит и сравнить результат с нулем.

Метод `get_ref` извлекает из `RefWithFlag` ссылку. Сначала он маскирует младший байт `usize` и преобразует результат в простой указатель. Оператор `as` не умеет преобразовывать простые указатели в ссылки, но мы можем разыменовать простой указатель (естественно, в `unsafe`-блоке) и заимствовать результат. Заимствование объекта, на который указывает простой указатель, дает ссылку с неограниченным временем жизни, поскольку Rust согласует время жизни ссылки со временем жизни окружающего кода, если таковой имеется. Обычно, однако, существует более специфическое и потому более точное время жизни, позволяющее обнаружить больше ошибок. В данном случае значение, возвращаемое методом `get_ref`, имеет тип `&'a T`, поэтому Rust делает вывод, что время жизни ссылки должно совпадать со временем жизни аргумента `RefWithFlag`, а это именно то, что нам нужно: время жизни ссылки, с которой мы начали.

В памяти значение типа `RefWithFlag` выглядит в точности как `usize`: поскольку `PhantomData` – тип нулевого размера, поле `behaves_like` не занимает места в структуре. Но `PhantomData` необходим, чтобы Rust знал, как обрабатывать времена жизни в коде, где используется `RefWithFlag`. Взгляните, как выглядел бы тип без поля `behaves_like`:

```
// Этот код не компилируется.
pub struct RefWithFlag<'a, T: 'a> {
    ptr_and_bit: usize
}
```

В главе 5 мы отмечали, что никакая структура, содержащая ссылки, не должна жить дольше значений, от которых они заимствованы, иначе ссылки станут висячими указателями. Структура должна соблюдать ограничения, действующие для ее членов. Это, конечно, относится и к `RefWithFlag`: в приведенном выше фрагменте кода переменная `flagged` не должна жить дольше вектора `vec`, поскольку метод `flagged.get_ref()` возвращает ссылку на него. Но наш упрощенный тип `RefWithFlag` не содержит вообще никаких ссылок, и в нем нигде не используется параметр времени жизни `'a`. Это просто `usize`. Тогда как же Rust узнает, что на время жизни `pab` налагаются какие-то ограничения? Включение поля `PhantomData<&'a T>` сообщает Rust, что с типом `RefWithFlag<'a, T>` нужно обращаться, *как будто* он содержит поле типа `&'a T`, хотя на реальное представление структуры в памяти это поле не влияет.

Хотя Rust толком не знает, что происходит (что именно делает тип `RefWithFlag` небезопасным), он делает все, что в его силах, чтобы помочь нам. Если мы опустим поле `_marker`, то Rust будет ругаться, что параметры `'a` и `T` не используются, и предложит воспользоваться типом `PhantomData`.

В коде RefWithFlag применена та же тактика, что в представленном выше типе Ascii, с целью избежать неопределенного поведения в unsafe-блоке. Сам тип открытый (объявлен с модификатором pub), но его члены закрыты, а значит, только код внутри самого модуля pointer_and_bool может создать значение типа RefWithFlag и получить доступ к его внутренним полям. Нет нужды анализировать незнамо сколько кода, чтобы увериться в корректности конструирования поля ptr_and_bit.

Нулевые указатели

Нулевой простой указатель в Rust – это нулевой адрес, как в C и C++. Для любого типа T функция std::ptr::null<T> возвращает нулевой указатель типа *const T, а функция std::ptr::null_mut<T> – нулевой указатель типа *mut T.

Существует несколько способов проверить, является ли простой указатель нулевым. Простейший из них – метод is_null, но метод as_ref иногда удобнее: он принимает указатель типа *const T и возвращает Option<&'a T>, преобразующее нулевой указатель в None. Аналогично метод as_mut преобразует указатели типа *mut T в значения типа Option<&'a mut T>.

Размеры и выравнивание типов

Значение любого типа, реализующего характеристику Sized, занимает постоянное число байтов в памяти и должно размещаться по адресу, кратному некоторой «границе выравнивания», которая определяется машинной архитектурой. Например, кортеж (i32, i32) занимает восемь байтов, и для большинства процессоров предпочтительно, чтобы он размещался по адресу, кратному 4.

Вызов std::mem::size_of::<T>() возвращает размер значения типа T в байтах, а std::mem::align_of::<T>() – требуемое выравнивание. Например:

```
assert_eq!(std::mem::size_of::<i64>(), 8);
assert_eq!(std::mem::align_of::<(i32, i32)>(), 4);
```

Выравнивание любого типа является степенью двойки.

Размер типа всегда округляется до величины, кратной его выравниванию, даже если технически он мог бы поместиться в области меньшего размера. Например, хотя для кортежа типа (f32, u8) необходимо всего пять байтов, size_of::<(f32, u8)>() равно 8, потому что align_of::<(f32, u8)>() равно 4. Тем самым гарантируется, что в массиве размер типа элемента всегда учитывает промежуток между соседними элементами.

Для безразмерных типов размер и выравнивание зависят от конкретного значения. Если имеется ссылка на безразмерное значение, то функции std::mem::size_of_val и std::mem::align_of_val возвращают его размер и выравнивание. Эти функции применимы к ссылкам на значения размерных и безразмерных типов.

```
// Толстые указатели на срезки хранят в себе длину объекта ссылки..
let slice: &[i32] = &[1, 3, 9, 27, 81];
assert_eq!(std::mem::size_of_val(slice), 20);

let text: &str = "alligator";
assert_eq!(std::mem::size_of_val(text), 9);

use std::fmt::Display;
let unremarkable: &Display = &193_u8;
```

```
let remarkable: &Display = &0.0072973525664;

// Эти функции возвращают размер и выравнивание значения, на которое
// указывает объект характеристики, а не самого объекта характеристики.
// Эта информация берется из vtable, на которую ссылается объект характеристики.
assert_eq!(std::mem::size_of_val(unremarkable), 1);
assert_eq!(std::mem::align_of_val(remarkable), 8);
```

Арифметика указателей

Rust размещает элементы массива, срезки или векторы в непрерывной области памяти. Промежутки между элементами постоянны, т. е. если элемент занимает `size` байтов, то `i`-й элемент начинается в позиции `i * size` (в байтах).

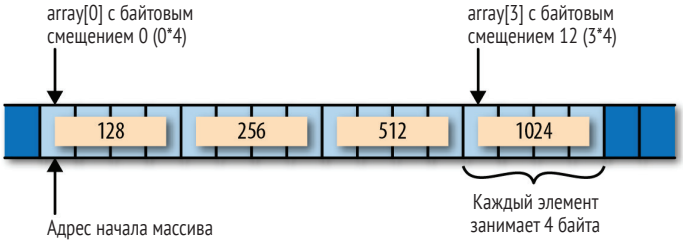


Рис. 21.1 ❖ Массив в памяти

Полезное следствие этого факта состоит в том, что если имеются два простых указателя на элементы массива, то сравнение указателей даст такой же результат, как сравнение индексов элементов: если `i < j`, то простой указатель на `i`-й элемент меньше простого указателя на `j`-ый элемент. Поэтому простые указатели можно использовать как границы при обходе массива. На самом деле простой итератор по срезке определен в стандартной библиотеке следующим образом:

```
struct Iter<'a, T: 'a> {
    ptr: *const T,
    end: *const T,
    ...
}
```

Поле `ptr` указывает на следующий элемент, который должен породить итератор, а поле `end` играет роль ограничителя: когда `ptr == end`, итерирование завершается.

Еще одно полезное следствие размещения массива в памяти: если `element_ptr` – простой указатель типа `*const T` или `*mut T` на `i`-й элемент некоторого массива, то `element_ptr.offset(o)` – простой указатель на `(i + o)`-ый элемент. Определение этой функции эквивалентно такому:

```
fn offset(self: *const T, count: isize) -> *const T
    where T: Sized
{
    let bytes_per_element = std::mem::size_of::<T>() as isize;
    let byte_offset = count * bytes_per_element;
    (self as isize).checked_add(byte_offset).unwrap() as *const T
}
```

Функция `std::mem::size_of::<T>` возвращает размер типа `T` в байтах. Поскольку тип `isize`, по определению, достаточно широк для хранения адреса, мы можем преобразовать базовый указатель в `isize`, произвести над этим значением арифметические действия и затем преобразовать результат обратно в указатель.

Разрешается порождать указатель на первый байт за концом массива. Разыменовать такой указатель нельзя, но он полезен для представления верхней границы цикла или для проверки выхода за границу массива.

Однако использование указателя на более отдаленные адреса или на адрес до начала массива считается неопределенным поведением, даже если такой указатель никогда не разыменовывается. Для оптимизации Rust хотел бы предполагать, что `ptr.offset(i) > ptr`, когда `i` положительно, и что `ptr.offset(i) < ptr`, когда `i` отрицательно. Это предположение выглядит безопасным, но может нарушаться, если арифметическая операция в `offset` приводит к переполнению типа `isize`. Если на `i` наложено ограничение – оставаться в том же массиве, что и `ptr`, – то переполнение невозможно: ведь сам массив не переполняет границ адресного пространства. (Чтобы указатель на первый байт за концом массива был безопасен, Rust никогда не размещает ничего в последнем байте адресного пространства.)

Если все-таки необходимо сдвигать указатели за пределы массива, с которым они ассоциированы, то можно использовать метод `wrapping_offset`. Он эквивалентен `offset`, но Rust не делает никаких предположений об относительном порядке `ptr.wrapping_offset(i)` и самого `ptr`. Разумеется, такие указатели по-прежнему нельзя разыменовывать, если они не указывают внутрь массива.

Передача в память и из памяти

Если вы реализуете тип, который сам управляет своей памятью, то должны следить за тем, в каких частях памяти находятся используемые значения, а какие не инициализированы, – точно так же, как Rust поступает в отношении локальных переменных. Рассмотрим такой код:

```
let pot = "pasta".to_string();
let plate;

plate = pot;
```

После его выполнения ситуация выглядит следующим образом:

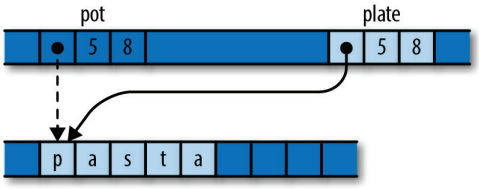


Рис. 21.2 ❖ Передача строки
от одной локальной переменной другой

После присваивания переменная `pot` оказывается неинициализированной, а `plate` владеет строкой.

На машинном уровне не определяется, как передача владения отражается на источнике, но на практике обычно не производится никаких действий. После присваивания `pot`, скорее всего, по-прежнему хранит указатель на строку, ее емкость и длину. Но, конечно, было бы безумием считать эти значения актуальными, и Rust этого не позволяет.

Те же соображения применимы и к структурам данных, управляющим собственной памятью. Рассмотрим такой код:

```
let mut noodles = vec!["udon".to_string()];
let soba = "soba".to_string();
let last;
```

В памяти мы имеем следующее состояние:

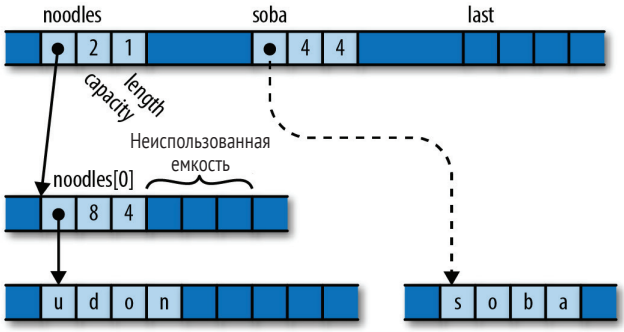


Рис. 21.3 ❖ Вектор, в котором имеется неинициализированная область

В этом векторе не использована емкость для хранения еще одного элемента, но эта область содержит мусор, возможно, оставшийся от предыдущего использования. Предположим, что мы затем выполнили такой код:

```
noodles.push(soba);
```

В результате добавления строки в вектор неинициализированная память становится новым элементом:

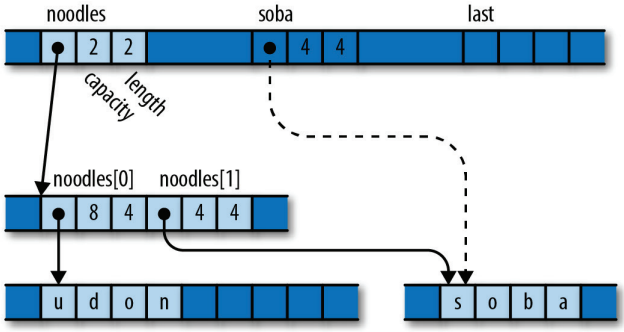


Рис. 21.4 ❖ После добавления в вектор значения `soba`

Вектор инициализировал свое ранее пустое пространство, так чтобы стать владельцем строки, и увеличил свою длину, чтобы отразить появление нового элемента. Теперь вектор владеет строкой, мы можем сослаться на его второй элемент, и при уничтожении вектора будет освобождена память обеих строк. А переменная `soba` деинициализирована.

Наконец, рассмотрим, что произойдет, когда мы удаляем значение из вектора:

```
last = noodles.pop().unwrap();
```

В памяти складывается такая ситуация:

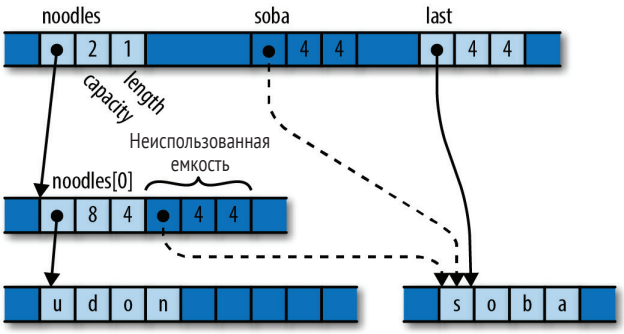


Рис. 21.5 ❖ После удаления элемента из вектора в переменную `last`

Переменная `last` приняла владение строкой. Вектор уменьшил свою длину, показывая, что область, в которой эта строка хранилась, теперь не инициализирована.

Как и в случае с переменными `pot` и `pasta` выше, переменные `soba`, `last`, а также освобожденная область вектора, скорее всего, содержат одинаковые комбинации битов. Но только `last` считается владельцем значения. Обращение с остальными двумя областями, как с вместилищами актуальных данных, было бы ошибкой.

Инициализированное значение правильно определять как *рассматриваемое как актуальное*. Запись в байты значения обычно является необходимой частью инициализации, но лишь потому, что тем самым мы подготавливаем его к тому, чтобы рассматривать как актуальное.

Rust следит за локальными переменными на этапе компиляции. Такие типы, как `Vec`, `HashMap`, `Box` и т. д., динамически следят за своими буферами. Если вы соберетесь реализовать тип, управляющий своей памятью, то должны будете поступить так же.

Rust предоставляет две основные операции для реализации таких типов:

- **`std::ptr::read(src)`** передает владение значением в области, на которую указывает `src`, вызывающей стороне. После вызова `read` память `*src` следует считать неинициализированной. Аргумент `src` должен быть простым указателем типа `*const T`, где `T` – размерный тип.

  Эта операция стоит за методом `Vec::pop`. При удалении значения из вектора вызывается `read`, чтобы изъять его из буфера, а затем длина вектора уменьшается на единицу, чтобы пометить освободившуюся область как неиспользованную емкость;

- `std::ptr::write(dest, value)` перемещает значение `value` в область, на которую указывает `dest`. До вызова эта область должна быть неинициализированной. Значением теперь владеет объект, на который направлен указатель. Аргумент `dest` должен быть простым указателем типа `*mut T`, а `value` – иметь тип `T`, где `T` – размерный тип.

  Эта операция стоит за `Vec::push`. При записи значения вызывается `write`, чтобы передать его в следующую свободную позицию вектора, а затем длина вектора увеличивается на 1, чтобы показать, что там теперь находится актуальный элемент.

Обе функции свободные, а не методы простого указателя.

Отметим, что с типами безопасных указателей Rust такие штуки не проходят. Все они требуют, чтобы соответствующие объекты в любой момент были инициализированы, поэтому преобразование неинициализированной памяти в значение или наоборот превышает их возможности. А с простыми указателями – пожалуйста.

Стандартная библиотека также предлагает функции для перемещения массивов значений из одной области памяти в другую:

- `std::ptr::copy(src, dst, count)` перемещает массив `count` значений из области с начальным адресом `src` в область с начальным адресом `dst`, как если бы это делалось в цикле, где `read` и `write` попеременно вызываются для перемещения одного значения. Конечная область памяти перед вызовом должна быть неинициализированной, а после вызова неинициализированной становится исходная область. Аргументы `src` и `dest` должны быть простыми указателями типа `*const T` и `*mut T` соответственно, `count` должен иметь тип `usize`;
- `std::ptr::copy_nonoverlapping(src, dst, count)` похожа на соответствующий вызов `copy`, но ее контракт дополнительно требует, чтобы исходная и конечная области памяти не пересекались. Операция может оказаться немного быстрее, чем `copy`.

Существуют еще два семейства функций `read` и `write`, также в модуле `std::ptr`:

- функции `read_unaligned` и `write_unaligned` похожи на `read` и `write`, но указатель необязательно должен быть выровнен, как обычно требуется для типов объектов, на которые ведет указатель. Эти функции могут работать медленнее, чем обычные функции `read` и `write`;
- функции `read_volatile` и `write_volatile` – эквиваленты волатильного чтения и записи в C и C++.

Пример: GapBuffer

В этом разделе приведен пример использования описанных выше функций для работы с простыми указателями.

Предположим, что мы пишем текстовый редактор и ищем тип для представления текста. Можно было бы взять `String` и использовать методы `insert` и `remove` для вставки и удаления символов в процессе ввода текста пользователем. Но если мы редактируем текст, находящийся в начале большого файла, то эти методы обойдутся дорого: вставка нового символа означает сдвиг всей последующей части строки вправо, а удаление – сдвиг влево. Хотелось бы, чтобы такие частые операции были подешевле.

В редакторе Emacs используется простая структура данных, называемая «буферным окном» (gap buffer), позволяющая эффективно вставлять и удалять символы за постоянное время. Если в `String` все свободное место находится в конце текста, что повышает эффективность методов `push` и `pop`, то в буферном окне это место находится в середине текста – в той точке, где производится редактирование. Это свободное место называется «окном». Вставка и удаление элементов в окне не занимают много времени – нужно просто уменьшить или увеличить окно. Окно можно сдвинуть в любое место, для этого требуется только переместить текст с одной стороны окна по другую его сторону. Если окно оказывается пусто, то мы переходим на буфер большего размера.

Хотя вставка и удаление в буферное окно производятся быстро, смена точки редактирования влечет за собой перемещение окна в новую позицию. Сдвиг элементов занимает время, пропорциональное расстоянию от старого места до нового. К счастью, в типичном сеансе редактирования изменения происходят пачками в одной окрестности буфера, а затем пользователь переходит в другое место.

В этом разделе мы реализуем буферное окно в Rust. Чтобы не отвлекаться на возню с UTF-8, будем хранить в буфере непосредственно значения типа `char`, но принцип работы остается таким же при хранении текста в иной форме.

Сначала покажем буферное окно в действии. Ниже создается `GapBuffer`, в него вставляется текст, а затем точка вставки перемещается в место, предшествующее последнему слову.

```
use gap::GapBuffer;

let mut buf = GapBuffer::new();
buf.insert_iter("Lord of the Rings".chars());
buf.set_position(12);
```

После выполнения этого кода буфер выглядит, как показано на рис. 21.6.

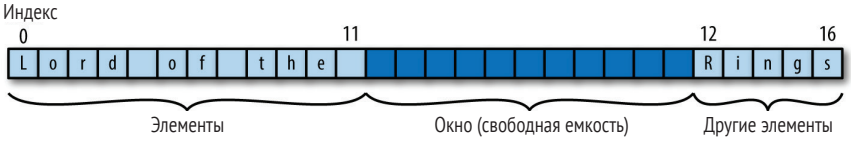


Рис. 21.6 ❖ Буферное окно, содержащее текст

Вставка сводится к заполнению окна новым текстом. В следующей строчке мы добавляем слово, не оставляя от фильма камня на камне[1]:

```
buf.insert_iter("Onion ".chars());
```

Это приводит к состоянию, показанному на рис. 21.7.

Вот определение нашего типа `GapBuffer`.

```
mod gap {
    use std;
    use std::ops::Range;
```

[1] Lord of the Rings – Властелин колец. Lord of the Onion Rings – Властелин луковых колец. – *Прим. перев.*

```
    pub struct GapBuffer<T> {
        // Место для хранения элементов. Его емкость такая, как нам нужно, но длина
        // всегда равна нулю. GapBuffer помещает элементы и окно в "неиспользованную"
        // емкость этого вектора.
        storage: Vec<T>,

        // Диапазон неинициализированных элементов в середине `storage`.
        // Элементы до и после этого диапазона всегда инициализированы.
        gap: Range<usize>
    }

    ...
}
```

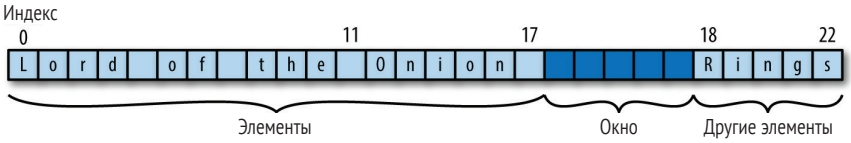


Рис. 21.7 ❖ Буферное окно, содержащее дополнительный текст

`GapBuffer` использует свое поле `storage` необычным способом[1]. Он никогда не помещает никаких элементов в сам вектор – ну или почти так. Он просто вызывает метод `Vec::with_capacity(n)`, чтобы получить блок памяти, достаточный для хранения `n` значений, получает простые указатели на эту память от методов вектора `as_ptr` и `as_mut_ptr`, а затем использует буфер вектора для собственных целей. Длина вектора всегда остается равной нулю. Когда вектор уничтожается, он не пытается освободить свои элементы, потому что не знает об их существовании, однако отведенный ему блок памяти освобождает. Именно это и нужно `GapBuffer`; у него имеется собственная реализация характеристики `Drop`, которая знает, где находятся элементы, и корректно уничтожает их.

Простейшие методы `GapBuffer` совпадают с нашими ожиданиями:

```
impl<T> GapBuffer<T> {
    pub fn new() -> GapBuffer<T> {
        GapBuffer { storage: Vec::new(), gap: 0..0 }
    }

    /// Возвращает число элементов, помещающихся в этот GapBuffer
    /// без перераспределения памяти.
    pub fn capacity(&self) -> usize {
        self.storage.capacity()
    }

    /// Возвращает число элементов, хранящихся в GapBuffer в данный момент.
    pub fn len(&self) -> usize { self.capacity() -
        self.gap.len()
    }

    /// Возвращает текущую точку вставки.
```

[1] Существуют более подходящие способы на основе типа `RawVec` из крейта `alloc`, но этот крейт еще нестабилен.

```
    pub fn position(&self) -> usize {
        self.gap.start
    }

    ...
}
```

Многие из последующих функций становятся чище, если завести вспомогательный метод, который возвращает простой указатель на элемент буфера с заданным индексом. Поскольку это Rust, нам понадобится один метод для `mut`-указателей и другой для `const`-указателей. В отличие от предыдущих, эти методы закрытые. Продолжаем начатый выше блок `impl`:

```
/// Возвращает указатель на элемент внутреннего хранилища с индексом `index`,
/// не зависящий от положения окна.
///
/// Безопасность: `index` должен быть корректным индексом элемента внутри
/// `self.storage`.
unsafe fn space(&self, index: usize) -> *const T {
    self.storage.as_ptr().offset(index as isize)
}

/// Возвращает изменяемый указатель на элемент внутреннего хранилища с индексом
/// `index`, не зависящий от положения окна.
///
/// Безопасность: `index` должен быть корректным индексом элемента внутри
/// `self.storage`.
unsafe fn space_mut(&mut self, index: usize) -> *mut T {
    self.storage.as_mut_ptr().offset(index as isize)
}
```

При поиске элемента с заданным индексом нужно принять во внимание, относится ли индекс к позиции до или после окна, и внести соответствующую поправку.

```
/// Вернуть смещение элемента с индексом `index` относительно начала буфера, учтя
/// наличие окна. Метод не проверяет, попадает ли индекс в диапазон, но никогда
/// не возвращает индекс в окне.
fn index_to_raw(&self, index: usize) -> usize {
    if index < self.gap.start {
        index
    } else {
        index + self.gap.len()
    }
}

/// Вернуть ссылку на элемент с индексом `index`
/// или `None`, если `index` выходит за границу.
pub fn get(&self, index: usize) -> Option<&T> {
    let raw = self.index_to_raw(index);
    if raw < self.capacity() {
        unsafe {
            // Мы только что сравнили `raw` с self.capacity(),
            // и index_to_raw пропускает окно, так что это безопасно.
            Some(&*self.space(raw))
```

```
        }
    } else {
        None
    }
}
```

Начиная делать вставки и удаления в другой части буфера, мы должны сдвинуть окно в новое положение. Это означает сдвиг элементов влево или наоборот – как пузырек в спиртовом уровне движется в направлении, противоположном движению жидкости.

```
/// Установить текущую позицию вставки равной `pos`.
/// Если `pos` выходит за границы, паниковать.
pub fn set_position(&mut self, pos: usize) {
    if pos > self.len() {
        panic!("index {} out of range for GapBuffer", pos);
    }

    unsafe {
        let gap = self.gap.clone();
        if pos > gap.start {
            // `pos` находится дальше окна. Переместить окно вправо,
            // сдвинув следующие за ним элементы влево.
            let distance = pos - gap.start;
            std::ptr::copy(self.space(gap.end),
                           self.space_mut(gap.start),
                           distance);
        } else if pos < gap.start {
            // `pos` находится раньше окна. Переместить окно влево,
            // сдвинув следующие за ним элементы вправо.
            let distance = gap.start - pos;
            std::ptr::copy(self.space(pos),
                           self.space_mut(gap.end - distance),
                           distance);
        }

        self.gap = pos .. pos + gap.len();
    }
}
```

Эта функция пользуется методом `std::ptr::copy` для сдвига элементов; `copy` требует, чтобы конечная область была неинициализированной, и оставляет неинициализированной исходную область. Исходная и конечная области могут перекрываться, но `copy` обрабатывает этот случай корректно. Поскольку до вызова окно является неинициализированной областью памяти, а функция корректирует положение окна, так что оно покрывает область, освобожденную при копировании, то контракт функции `copy` соблюден.

Операции вставки и удаления элементов сравнительно просты. Вставка занимает под новый элемент одну позицию окна, а удаление выдвигает одно значение и увеличивает окно, распространяя его на занятую ранее позицию.

```
/// Вставляет `elt` в текущую позицию вставки
/// и располагает позицию вставки после него.
pub fn insert(&mut self, elt: T) {
```

```
    if self.gap.len() == 0 {
        self.enlarge_gap();
    }

    unsafe {
        let index = self.gap.start;
        std::ptr::write(self.space_mut(index), elt);
    }
    self.gap.start += 1;
}

/// Вставляет элементы, порожденные `iter`, в текущую позицию вставки
/// и располагает позицию вставки после них.
pub fn insert_iter<I>(&mut self, iterable: I)
    where I: IntoIterator<Item=T>
{
    for item in iterable {
        self.insert(item)
    }
}

/// Удаляет элемент, расположенный после позиции вставки, и
/// возвращает его. Если позиция вставки находится в конце
/// GapBuffer, то возвращает `None`.
pub fn remove(&mut self) -> Option<T> {
    if self.gap.end == self.capacity() {
        return None;
    }

    let element = unsafe {
        std::ptr::read(self.space(self.gap.end))
    };
    self.gap.end += 1;
    Some(element)
}
```

Аналогично тому, как в `Vec` используются функции `std::ptr::write` для `push` и `std::ptr::read` для `pop`, в `GapBuffer` используется `write` для `insert` и `read` для `remove`. И точно так же, как `Vec` должен корректировать свою длину, чтобы провести границу между инициализированными элементами и свободной емкостью, так и `GapBuffer` корректирует свое окно.

Когда окно заполняется, метод `insert` должен увеличить буфер. Это делает метод `enlarge_gap` (последний в блоке `impl`).

```
/// Удваивает емкость `self.storage`.
fn enlarge_gap(&mut self) {
    let mut new_capacity = self.capacity() * 2;
    if new_capacity == 0 {
        // Существующий вектор пуст.
        // Выбрать разумную начальную емкость.
        new_capacity = 4;
    }

    // Мы понятия не имеем, что происходит с "неиспользованной" емкостью
    // при изменении размера Vec. Поэтому просто создадим новый вектор
```

```
    // и переместим в него элементы.
    let mut new = Vec::with_capacity(new_capacity);
    let after_gap = self.capacity() - self.gap.end;
    let new_gap = self.gap.start .. new.capacity() - after_gap;

    unsafe {
        // Переместить элементы, оказавшиеся левее окна.
        std::ptr::copy_nonoverlapping(self.space(0),
                                      new.as_mut_ptr(),
                                      self.gap.start);

        // Переместить элементы, оказавшиеся правее окна.
        let new_gap_end = new.as_mut_ptr().offset(new_gap.end as isize);
        std::ptr::copy_nonoverlapping(self.space(self.gap.end),
                                      new_gap_end,
                                      after_gap);
    }

    // Здесь старый Vec освобождается, но не уничтожает никаких элементов,
    // потому что его длина равна нулю.
    self.storage = new;
    self.gap = new_gap;
}
```

Если `set_position` должна использовать `copy` для перемещения элементов туда-сюда в окне, то `enlarge_gap` может обратиться к `copy_nonoverlapping`, поскольку перемещает элементы в совершенно новый буфер.

При перемещении нового вектора в `self.storage` старый вектор уничтожается. Поскольку его длина равна нулю, никаких подлежащих уничтожению элементов в нем нет, так что всё сводится к освобождению занятой буфером памяти. Очень удобно, что `copy_nonoverlapping` оставляет исходную область неинициализированной, так что старый вектор прав в своих ожиданиях: всеми элементами теперь владеет новый вектор.

Наконец, мы должны гарантировать, что уничтожение `GapBuffer` приводит к уничтожению всех его элементов.

```
impl<T> Drop for GapBuffer<T> {
    fn drop(&mut self) {
        unsafe {
            for i in 0 .. self.gap.start {
                std::ptr::drop_in_place(self.space_mut(i));
            }
            for i in self.gap.end .. self.capacity() {
                std::ptr::drop_in_place(self.space_mut(i));
            }
        }
    }
}
```

Элементы находятся до и после окна, поэтому мы просто обходим оба участка и вызываем функцию `std::ptr::drop_in_place` для уничтожения каждого элемента. `drop_in_place` – это служебная функция, которая работает, как `drop(std::ptr::read(ptr))`, но не утруждает себя передачей значения вызывающей стороне

(а потому применима и к безразмерным типам). И, как и в случае `enlarge_gap`, к тому моменту, как вектор `self.storage` будет уничтожен, его буфер действительно деинициализирован.

Как и другие типы, продемонстрированные в этой главе, `GapBuffer` гарантирует, что его собственных инвариантов достаточно для соблюдения контрактов всех используемых в нем небезопасных средств, поэтому ни один из его открытых методов можно не помечать как небезопасный. `GapBuffer` реализует безопасный интерфейс к средству, которое невозможно эффективно написать с помощью безопасного кода.

Безопасность паники в небезопасном коде

В Rust паника обычно не приводит к неопределенному поведению; макрос `panic!` не является небезопасным средством. Но когда вы начинаете работать с небезопасным кодом, обеспечение безопасности паники становится вашей задачей.

Рассмотрим метод `GapBuffer::remove` из предыдущего раздела:

```
pub fn remove(&mut self) -> Option<T> {
    if self.gap.end == self.capacity() {
        return None;
    }

    let element = unsafe {
        std::ptr::read(self.space(self.gap.end))
    };
    self.gap.end += 1;
    Some(element)
}
```

Вызов `read` удаляет элемент, непосредственно следующий за окном, из буфера, оставляя вместо него неинициализированную память. К счастью, следующее же предложение расширяет окно на эту позицию, так что к моменту возврата из метода все оказывается в порядке: все элементы вне окна инициализированы, а все элементы внутри окна не инициализированы.

Но подумаем, что произойдет, если после обращения к `read`, но до изменения `self.gap.end` мы попытались бы воспользоваться средством, способным вызвать панику, – скажем, доступом к срезке по индексу. Незапланированный выход из метода между этими двумя действиями оставил бы `GapBuffer` с неинициализированным элементом вне окна. При следующем обращении к `remove` мы могли бы попытаться снова прочитать его с помощью `read`, и даже простое уничтожение `GapBuffer` привело бы к попытке удалить этот элемент. То и другое – неопределенное поведение, поскольку производится доступ к неинициализированной памяти.

Невозможно избежать краткосрочного нарушения инвариантов типа его методами, когда они делают свою работу. Но еще до возврата управления методы приводят всё в корректное состояние. Однако если в середине метода возникает паника, то до процесса восстановления дело может и не дойти, и тогда тип останется в противоречивом состоянии.

Если в реализации типа используется только безопасный код, то такая несогласованность может стать причиной некорректного поведения типа, но не приведет к неопределенному поведению. Однако код, в котором используются небезопасные средства, обычно рассчитывает на свои инварианты, считая, что та-

ким образом соблюдает контракты этих средств. Нарушенные инварианты ведут к нарушенным контрактам, а значит, к неопределенному поведению.

При работе с небезопасными средствами следует уделять особое внимание выявлению таких участков и следить за тем, чтобы в них не могла возникнуть паника.

Иноязычные функции: вызов функций на C и C++ из Rust

Интерфейс с иноязычными функциями позволяет вызывать из Rust-кода функции, написанные на C или C++.

В этом разделе мы напишем программу, которая компонуется с библиотекой `libgit2`, написанной на C и предназначенной для работы с системой управления версиями Git. Сначала покажем, как оформляется вызов C-функций из Rust. А затем продемонстрируем построение безопасного интерфейса к `libgit2`, взяв за образец крейт с открытым исходным кодом `git2-rs`, который как раз это и делает.

Предполагается, что вы знакомы с языком C и механизмами компиляции и компоновки программ на C. Работа с C++ устроена аналогично. Предполагается также, что вы хоть немного знакомы с системой Git.

Поиск общего представления данных

Общим знаменателем Rust и C является машинный язык, поэтому чтобы понять, как значения Rust выглядят со стороны C-кода и наоборот, нужно рассмотреть их машинные представления. В этой книге мы не раз показывали размещение значений в памяти, так что вы, наверное, обратили внимание, что у данных в C и Rust много общего: к примеру, тип Rust `usize` и тип C `size_t` идентичны, а идея структур одинакова в обоих языках. Чтобы установить соответствие между типами Rust и C, мы начнем с примитивов, а затем постепенно перейдем к более сложным типам.

Для языка системного программирования C на удивление небрежно относится к представлению типов: длина `int` обычно равна 32 бита, но может быть как больше, так и меньше (16 бит): тип `char` может быть знаковым и беззнаковым и т. д. Чтобы как-то справиться с таким непостоянством, в модуле Rust `std::os::raw` определен набор типов, которые гарантированно имеют такое же представление, как некоторые типы C. В него входят примитивные целые и символьные типы.

Тип C	Соответствующий тип в std::os::raw
`short`	`c_short`
`int`	`c_int`
`long`	`c_long`
`long long`	`c_longlong`
`unsigned short`	`c_ushort`
`unsigned, unsigned int`	`c_uint`
`unsigned long`	`c_ulong`
`unsigned long long`	`c_ulonglong`
`char`	`c_char`
`signed char`	`c_schar`
`unsigned char`	`c_uchar`
`float`	`c_float`
`double`	`c_double`
`void *, const void *`	`*mut c_void, *const c_void`

Сделаем несколько замечаний об этой таблице:

- за исключением `c_void`, все представленные в ней типы Rust – псевдонимы некоторого примитивного типа Rust, например: `c_char` – это либо `i8`, либо `u8`;
- официально не существует типа Rust, соответствующего типу C `bool`. В настоящий момент тип `bool` Rust занимает один байт, равный 0 или 1, и такое же представление используется во всех основных реализациях C и C++. Однако разработчики Rust еще не решили, сохранять ли такое представление в будущем, потому что оно закрывает некоторые возможности оптимизации;
- 32-разрядный тип `char` в Rust не является аналогом типа `wchar_t`, ширина и кодировка которого могут зависеть от реализации. Ближе к нему тип C `char32_t`, но до сих пор не гарантируется, что его кодировка – Юникод;
- примитивные типы Rust `usize` и `isize` имеют такое же представление, как типы C `size_t` и `ptrdiff_t`;
- указатели C и C++ и ссылки C++ соответствуют типам простых указателей Rust, `*mut T` и `*const T`;
- технически стандарт C разрешает реализации использовать представления, для которых в Rust нет подходящего типа: 36-разрядные целые, представления со знаком и порядком для чисел со знаком и т. д. Практически же на всех платформах, куда перенесен Rust, каждому привычному целому типу C, за исключением `bool`, есть соответствие в Rust.

Для определения структурных типов Rust, совместимых со структурами C, можно использовать атрибут `#[repr(C)]`. Его наличие перед определением структуры означает, что Rust должен разместить поля структуры в памяти точно так же, как это сделал бы компилятор C с аналогичной структурой. Например, в заголовочном файле `git2/errors.h` библиотеки `libgit2` определена C-структура, содержащая сведения об ошибке:

```
typedef struct {
    char *message;
    int klass;
} git_error;
```

Определение типа Rust с идентичным представлением выглядит так:

```
#[repr(C)]
pub struct git_error {
    pub message: *const c_char,
    pub klass: c_int
}
```

Атрибут `#[repr(C)]` влияет только на размещение в памяти самой структуры, а не отдельных полей, поэтому соответствие с C-структурой будет достигнуто, только если каждое поле имеет тип, совместимый с C: `*const c_char` для `char *`, `c_int` для `int` и т. д.

В данном конкретном случае атрибут `#[repr(C)]`, скорее всего, не изменит размещения структуры `git_error`. Не так уж много интересных способов разместить указатель и целое. Но если C и C++ гарантируют, что поля структуры размещены в памяти точно в порядке объявления и каждое – по отдельному адресу, то Rust изменяет порядок членов, чтобы минимизировать общий размер структуры,

а типы нулевого размера вообще не занимают места. Атрибут `#[repr(C)]` просит Rust соблюдать правила C для данного типа.

Существует также атрибут `#[repr(C)]`, контролирующий представление перечислений в стиле C:

```
#[repr(C)]
enumgit_error_code {
    GIT_OK         =  0,
    GIT_ERROR      = -1,
    GIT_ENOTFOUND  = -3,
    GIT_EEXISTS    = -4,
    ...
}
```

Обычно Rust применяет различные приемы для выбора представления перечисления. Например, мы уже упоминали трюк, благодаря которому Rust хранит `Option<&T>` в одном слове (если тип `T` размерный). Без атрибута `#[repr(C)]` Rust использовал бы один байт для представления перечисления `git_error_code`, а при наличии этого атрибута используется столько байтов, сколько занимает тип C `int`.

Можно попросить Rust использовать для перечисления такое же представление, как для конкретного целого типа. Если предпослать определению атрибут `#[repr(i16)]`, то мы получим 16-разрядный тип с таким же представлением, как у следующего перечисления в C++:

```
#include <stdint.h>

enumgit_error_code: int16_t {
    GIT_OK         =  0,
    GIT_ERROR      = -1,
    GIT_ENOTFOUND  = -3,
    GIT_EEXISTS    = -4,
    ...
};
```

Передать строки между Rust и C несколько сложнее. В C строка представляется указателем на массив символов, завершающийся нулем. А в Rust длина строки хранится явно – либо в поле `String`, либо во втором слове толстой ссылки `&str`. Строки в Rust не завершаются нулем, они вполне могут содержать внутри нулевые символы, которые в этом смысле ничем не отличаются от прочих.

Это означает, что нельзя заимствовать строку Rust, как C-строку: если передать в C-код указатель на строку Rust, то он может по ошибке принять внутренний нулевой символ за конец строки или отправиться неизвестно куда в поисках завершающего нуля, которого нет. Но заимствовать C-строку как ссылку Rust `&str`, в принципе, можно при условии, что она содержит корректный текст в кодировке UTF-8.

Из-за всего этого Rust вынужден рассматривать C-строки как типы, полностью отличающиеся от `String` и `&str`. В модуле `std::ffi` типы `CString` и `CStr` представляют принадлежащие и заимствованные массивы байтов, завершающиеся нулем. По сравнению с `String` и `str`, набор методов `CString` и `CStr` крайне ограничен: допустимы только конструирование и преобразование в другие типы. В следующем разделе мы продемонстрируем эти типы в действии.

Объявление иноязычных функций и переменных

В блоке `extern` объявляются функции и переменные, определенные в какой-то другой библиотеке, с которой будет компоноваться окончательная исполняемая Rust-программа. Например, любая Rust-программа компонуется со стандартной библиотекой C, поэтому можно сообщить Rust о библиотечной C-функции `strlen` следующим образом:

```
use std::os::raw::c_char;

extern {
    fn strlen(s: *const c_char) -> usize;
}
```

Тем самым мы сообщаем Rust имя и тип функции, а компоновку с ее определением оставляем на потом.

Rust предполагает, что для функций, объявленных в блоках `extern`, действуют принятые в C соглашения о передаче аргументов и возврате значений. Они определяются как `unsafe`-функции. Для `strlen` то и другое правильно: это и в самом деле C-функция, и ее спецификация требует передачи действительного указателя на строку, завершающуюся нулем, – контракт, который Rust проконтролировать не может. (Почти любая функция, принимающая простой указатель, должна быть небезопасной: в безопасном Rust можно конструировать простые указатели из произвольных целых чисел, но разыменование такого указателя считается неопределенным поведением.)

Имея такой блок `extern`, мы можем вызывать `strlen`, как любую другую функцию Rust, хотя тип выдает в ней туриста:

```
use std::ffi::CString;

let rust_str = "I'll be back";
let null_terminated = CString::new(rust_str).unwrap();
unsafe {
    assert_eq!(strlen(null_terminated.as_ptr()), 12);
}
```

Функция `CString::new` создает завершающуюся нулем C-строку. Сначала она проверяет, что в аргументе нет внутренних нулей, поскольку такую строку представить в C было бы невозможно, и возвращает ошибку, если найдет таковые (отсюда необходимость в `unwrap`). В противном случае она добавляет в конец нулевой байт и возвращает значение типа `CString`, владеющее получившимися символами.

Стоимость `CString::new` зависит от переданного ей типа. Она согласна на любой тип, реализующий характеристику `Into<Vec<u8>>`. Передача `&str` влечет за собой выделение памяти и копирование, поскольку преобразование в `Vec<u8>` создает выделенную в куче копию строки, которой мог бы владеть вектор. Но передача `String` по значению просто потребляет строку и перенимает владение ее буфером, поэтому если добавление завершающего нуля не приводит к изменению размера буфера, то преобразование не сопровождается ни выделением памяти, ни копированием.

Результатом разыменования `CString` является значение типа `CStr`, метод `as_ptr` которого возвращает `*const c_char`, указывающий на начало строки. Это как раз тот

тип, которого ожидает `strlen`. В нашем примере `strlen` просматривает строку, находит нулевой байт, помещенный туда функцией `CString::new`, и возвращает длину строки в виде числа байтов.

В блоках `extern` можно также объявлять глобальные переменные. В POSIX-системах имеется глобальная переменная `environ`, содержащая значения всех переменных среды процесса. В C она объявлена следующим образом:

```
extern char **environ;
```

В Rust нужно было бы написать:

```
use std::ffi::CStr;
use std::os::raw::c_char;

extern {
    static environ: *mut *mut c_char;
}
```

Чтобы напечатать первую переменную среды, следует написать:

```
unsafe {
    if !environ.is_null() && !(*environ).is_null() {
        let var = CStr::from_ptr(*environ);
        println!("первая переменная среды: {}",
                 var.to_string_lossy())
    }
}
```

Убедившись, что в `environ` есть первый элемент, программа вызывает `CStr::from_ptr`, чтобы построить заимствующее его значение типа `CStr`. Метод `to_string_lossy` возвращает `Cow<str>`: если C-строка содержит допустимый текст в кодировке UTF-8, то `Cow` возвращает ее содержимое в виде `&str` без завершающего нулевого байта. В противном случае `to_string_lossy` создает копию текста в куче, заменяет все недопустимые в UTF-8 последовательности официальным символом замены Юникода, '`&#xFFFD;`', и строит для нее владеющее значение `Cow`. В любом случае результат реализует характеристику `Display`, так что его можно напечатать посредством параметра формата `{}`.

Использование библиотечных функций

Для использования функций из некоторой библиотеки можно поместить над блоком `extern` атрибут `#[link]` с именем библиотеки, с которой Rust должен скомпоновать исполняемую программу. Вот, например, программа, которая вызывает функции инициализации и очистки библиотеки `libgit2`, но больше ничего не делает:

```
use std::os::raw::c_int;

#[link(name = "git2")]
extern {
    pub fn git_libgit2_init() -> c_int;
    pub fn git_libgit2_shutdown() -> c_int;
}

fn main() {
```

```
    unsafe {
        git_libgit2_init();
        git_libgit2_shutdown();
    }
}
```

В блоке `extern` объявлены внешние функции, как и раньше. Атрибут `#[link(name = "git2")]` оставляет в крейте примечание, означающее, что когда Rust будет создавать исполняемую программу или разделяемую библиотеку, он должен скомпоновать ее с библиотекой `git2`. Для построения исполняемых файлов Rust пользуется системным компоновщиком; в Unix ему передается аргумент командной строки `-lgit2`, в Windows – `git2.LIB`.

Атрибут `#[link]` работает и для библиотечных крейтов. При построении программы, зависящей от других крейтов, Cargo собирает примечания компоновки со всего графа зависимостей и передает необходимые библиотеки компоновщику.

Если вы захотите проделать все описанное в примере на своей машине, то должны будете самостоятельно собрать библиотеку `libgit2`. Мы работали с `libgit2` версии 0.25.1, доступной на сайте https://libgit2.github.com. Для компиляции `libgit2` необходимо установить систему сборки CMake и язык Python; мы пользовались версиями CMake 3.8.0 и Python 2.7.13, скачав их с сайтов https://cmake.org и https://www.python.org.

Полные инструкции по сборке `libgit2` имеются на сайте, но это достаточно просто, так что мы приведем лишь основные команды. Для Linux будем предполагать, что исходный код библиотеки уже распакован в каталог `/home/jimb/libgit2-0.25.1`:

```
$ cd /home/jimb/libgit2-0.25.1
$ mkdir build
$ cd build
$ cmake ..
$ cmake --build .
```

В Linux результатом этого процесса являются разделяемая библиотека `/home/jimb/libgit2-0.25.1/build/libgit2.so.0.25.1` и стандартный набор указывающих на нее символических ссылок, включая `libgit2.so`. В macOS результат похожий, но библиотека называется `libgit2.dylib`.

В Windows тоже всё просто. Допустим, что исходный код распакован в каталог `C:\Users\JimB\libgit2-0.25.1`. В окне команд Visual Studio введите:

```
> cd C:\Users\JimB\libgit2-0.25.1
> mkdir build
> cd build
> cmake -A x64 ..
> cmake --build .
```

Это те же команды, что в Linux, только при первом запуске CMake нужно запросить 64-разрядную сборку, соответствующую компилятору Rust. (Если вы установили 32-разрядный комплект инструментов Rust, то опустите флаг `-A x64` в первой команде `cmake`.) В результате получаются импортируемая библиотека `git2.LIB` и динамически загружаемая библиотека `git2.DLL`, обе в каталоге `C:\Users\JimB\libgit2-0.25.1\build\Debug`. (Далее мы приводим инструкции только для Unix, за исключением случаев, когда в Windows все совсем по-другому.)

Создайте Rust-программу в отдельном каталоге:

```
$ cd /home/jimb
$ cargo new --bin git-toy
```

и скопируйте показанный выше код в файл `src/main.rs`. Естественно, если вы попробуете его сейчас собрать, то Rust не будет знать, где взять только что собранную библиотеку `libgit2`:

```
$ cd git-toy
$ cargo run
      Compiling git-toy v0.1.0 (file:///home/jimb/git-toy)
error: linking with `cc` failed: exit code: 1
  |
 = note: "cc" ... "-l" "git2" ...
 = note: /usr/bin/ld: cannot find -lgit2
         collect2: error: ld returned 1 exit status

error: aborting due to previous error

error: Could not compile `git-toy`.

To learn more, run the command again with --verbose.
$
```

Чтобы сообщить Rust, где искать библиотеки, мы можем написать *сборочный скрипт* – код на Rust, который Cargo компилирует и исполняет во время сборки. Сборочные скрипты могут делать самые разные вещи: динамически генерировать код, компилировать C-код, включаемый в крейт, и т. д. В данном случае нужно всего лишь добавить путь к библиотеке в команду компоновки исполняемого файла. Выполняя сборочный скрипт, Cargo разбирает его вывод в поисках необходимой информации, так что скрипт должен выводить в стандартный поток данные в точно определенном формате.

Для создания сборочного скрипта добавьте файл `build.rs` в тот же каталог, что `Cargo.toml`, и поместите в него такой код:

```
fn main() {
    println!(r"cargo:rustc-link-search=native=/home/jimb/libgit2-0.25.1/build");
}
```

Это правильный путь для Linux; в Windows после `native=` нужно указать путь `C:\Users\JimB\libgit2-0.25.1\build\Debug`. (Мы для простоты срезаем некоторые углы; в реальном приложении следует избегать абсолютных путей в сборочном скрипте. В конце раздела мы дадим ссылку на документацию, в которой описывается, как делать правильно.)

Далее скажите Cargo, что это сборочный скрипт, добавив строку `build = "build.rs"` в секцию `[package]` файла `Cargo.toml`. Теперь файл должен выглядеть так:

```
[package]
name = "git-toy"
version = "0.1.0"
authors = ["Вы <you@example.com>"]
build = "build.rs"
[dependencies]
```

Почти все готово к запуску программы. В macOS она уже может работать, в Linux вы, скорее всего, увидите что-то в этом роде:

```
$ cargo run
   Compiling git-toy v0.1.0 (file:///home/jimb/git-toy)
    Finished dev [unoptimized + debuginfo] target(s) in 0.64 secs
     Running `target/debug/git-toy`
target/debug/git-toy: error while loading shared libraries:
libgit2.so.25: cannot open shared object file: No such file or directory
$
```

Это означает, что хотя Cargo успешно скомпоновала исполняемый файл с библиотекой, она не знает, где искать библиотеку на этапе выполнения. В Windows аналогичное сообщение появляется в диалоговом окне. В Linux необходимо задать переменную среды `LD_LIBRARY_PATH`:

```
$ export LD_LIBRARY_PATH=/home/jimb/libgit2-0.25.1/build:$LD_LIBRARY_PATH
$ cargo run
   Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running `target/debug/git-toy`
$
```

В macOS соответствующая переменная называется `DYLD_LIBRARY_PATH`.

А в Windows следует задать переменную среды `PATH`:

```
> set PATH=C:\Users\JimB\libgit2-0.25.1\build\Debug;%PATH%
> cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/git-toy`
>
```

Понятно, что в развернутом приложении хотелось бы избежать задания переменных среды с одной лишь целью – найти библиотеку. Одна из альтернатив – статически прикомпоновать к крейту C-библиотеку. Тогда объектные файлы библиотеки копируются в `rlib`-файл крейта вместе с объектными файлами и метаданными Rust-кода. И весь этот набор участвует в окончательной компоновке.

В Cargo принято соглашение, согласно которому крейт, предоставляющий доступ к C-библиотеке, должен называться `LIB-sys`, где `LIB` – имя C-библиотеки. Такой `-sys`-крейт не должен содержать ничего, кроме статически скомпонованной библиотеки и модулей Rust с блоками `extern` и определениями типов. Тогда интерфейсы более высокого уровня будут находиться в крейтах, зависящих от `-sys`-крейта. В результате от одного и того же `-sys`-крейта может зависеть несколько крейтов верхнего уровня, при условии что существует единая версия `-sys`-крейта, отвечающая потребностям всех.

Полную информацию о поддержке сборочных скриптов и компоновке с системными библиотеками см. в документации по Cargo на странице http://doc.crates.io/guide.html. Там описано, как избежать абсолютных путей в сборочных скриптах, какие имеются флаги компилятора, как пользоваться инструментами типа `pkg-config` и т. д. Крейт `git2-rs` также дает хорошие примеры для подражания, в его сборочных скриптах обрабатываются кое-какие сложные ситуации.

Низкоуровневый интерфейс с libgit2

Вопрос о том, как правильно использовать библиотеку use `libgit2`, распадается на два вопроса:

- что значит использовать функции из `libgit2` в Rust?
- как обернуть их безопасным интерфейсом на Rust?

Будем рассматривать эти вопросы поочередно. В этом разделе мы напишем программу, представляющую собой, по сути дела, один гигантский `unsafe`-блок, наполненный неидиоматическим Rust-кодом, в котором нашло отражение несоответствие систем типов и соглашений, неизбежное при программировании на смеси языков. Мы называем это «низкоуровневым» интерфейсом. Код получился запутанным, но зато на нем видны все шаги, которые нужно предпринять для использования `libgit2` из программы на Rust.

А в следующем разделе мы построим безопасный интерфейс к `libgit2`, в котором типы Rust используются для принудительного соблюдения требований, предъявляемых библиотекой `libgit2` к пользователям. По счастью, библиотека `libgit2` исключительно хорошо спроектирована, поэтому вопросы, которые необходимо задать с точки зрения требований безопасности Rust, имеют достаточно хорошие ответы, и вполне можно построить идиоматичный интерфейс с Rust без `unsafe`-функций.

Программа, которую мы собираемся написать, тоже очень проста: она принимает в командной строке путь, открывает репозиторий Git, расположенный по этому пути, и печатает головную фиксацию. Но этого достаточно, чтобы проиллюстрировать основные стратегии построения безопасного и идиоматичного интерфейса с Rust.

Для построения низкоуровневого интерфейса нам понадобится несколько больший набор функций и типов из `libgit2`, чем мы использовали раньше, поэтому имеет смысл перенести блок `extern` в отдельный модуль. Создадим в каталоге `gittoy/src` файл `raw.rs` с таким содержимым:

```
#![allow(non_camel_case_types)]
use std::os::raw::{c_int, c_char, c_uchar};

#[link(name = "git2")]
extern {
    pub fn git_libgit2_init() -> c_int;
    pub fn git_libgit2_shutdown() -> c_int;
    pub fn giterr_last() -> *const git_error;

    pub fn git_repository_open(out: *mut *mut git_repository,
                               path: *const c_char) -> c_int;
    pub fn git_repository_free(repo: *mut git_repository);

    pub fn git_reference_name_to_id(out: *mut git_oid,
                                    repo: *mut git_repository,
                                    reference: *const c_char) -> c_int;

    pub fn git_commit_lookup(out: *mut *mut git_commit,
                             repo: *mut git_repository,
                             id: *const git_oid) -> c_int;

    pub fn git_commit_author(commit: *const git_commit) -> *const git_signature;
```

```
    pub fn git_commit_message(commit: *const git_commit) -> *const c_char;
    pub fn git_commit_free(commit: *mut git_commit);
}

pub enum git_repository {}
pub enum git_commit {}

#[repr(C)]
pub struct git_error {
    pub message: *const c_char,
    pub klass: c_int
}

#[repr(C)]
pub struct git_oid {
    pub id: [c_uchar; 20]
}

pub type git_time_t = i64;

#[repr(C)]
pub struct git_time {
    pub time: git_time_t,
    pub offset: c_int
}

#[repr(C)]
pub struct git_signature {
    pub name: *const c_char,
    pub email: *const c_char,
    pub when: git_time
}
```

Здесь каждый элемент основан на объявлении из собственных заголовочных файлов `libgit2`. Например, `libgit2-0.25.1/include/git2/repository.h` включает объявления:

```
extern int git_repository_open(git_repository **out, const char *path);
```

Эта функция пытается открыть репозиторий Git с путем `path`. Если все получается, то она создает объект `git_repository` и сохраняет указатель на него по адресу, на который указывает аргумент `out`. Эквивалентное объявление на Rust имеет вид:

```
pub fn git_repository_open(out: *mut *mut git_repository,
                           path: *const c_char) -> c_int;
```

В открытых заголовочных файлах `libgit2` тип `git_repository` определен как typedef для неполного структурного типа:

```
typedef struct git_repository git_repository;
```

Поскольку детали этого типа скрыты в библиотеке, в открытых заголовочных файлах тип `struct git_repository` не определен, а значит, пользователь библиотеки не сможет построить экземпляр этого типа самостоятельно. Один из возможных аналогов неполного структурного типа в Rust выглядит так:

```
pub enum git_repository {}
```

В этом типе перечисления нет вариантов. Rust никогда не сможет создать значение такого типа. Это немного странно, но является точным отражением типа C, который должна конструировать только сама библиотека `libgit2` и которым можно манипулировать только с помощью простых указателей.

Написание больших блоков `extern` вручную может оказаться неприятной работой. Создавая интерфейс к сложной C-библиотеке, можете попробовать использовать крейт `bindgen`, в нем имеются функции для автоматического разбора заголовочных файлов C и генерации соответствующих объявлений на Rust, которые вы сможете вставить в свой сборочный скрипт. У нас нет места для демонстрации `bindgen`, но на его странице на сайте `crates.io` приведены ссылки на документацию.

Далее мы полностью перепишем файл `main.rs`. Прежде всего нужно объявить модуль `raw`:

```
mod raw;
```

По соглашению, принятому в `libgit2`, функции, которые могут завершиться неудачно, возвращают целочисленный код, равный положительному числу или нулю в случае успеха и отрицательному числу в случае ошибки. Если произошла ошибка, то функция `giterr_last` возвращает указатель на структуру `git_error`, содержащую подробную информацию о случившемся. Этой структурой владеет `libgit2`, поэтому мы не должны освобождать ее самостоятельно, но при следующем вызове библиотеки она может быть перезаписана. В правильном интерфейсе с Rust следовало бы использовать тип `Result`, но в низкоуровневой версии мы хотим использовать функции `libgit2` в их исходном виде, поэтому придется изобрести свою функцию для обработки ошибок:

```
use std::ffi::CStr;
use std::os::raw::c_int;

fn check(activity: &'static str, status: c_int) -> c_int {
    if status < 0 {
        unsafe {
            let error = &*raw::giterr_last();
            println!("error while {}: {} ({})",
                     activity,
                     CStr::from_ptr(error.message).to_string_lossy(),
                     error.klass);
            std::process::exit(1);
        }
    }

    status
}
```

Этой функцией мы будем пользоваться для проверки результата обращения к `libgit2`:

```
check("initializing library", raw::git_libgit2_init());
```

Здесь применяются методы `CStr`, с которыми мы уже встречались: `from_ptr` для конструирования `CStr` из C-строки и `to_string_lossy` для преобразования ее в нечто такое, что Rust может напечатать.

Далее нам нужна функция для печати фиксации:

```
unsafe fn show_commit(commit: *const raw::git_commit) {
    let author = raw::git_commit_author(commit);

    let name = CStr::from_ptr((*author).name).to_string_lossy();
    let email = CStr::from_ptr((*author).email).to_string_lossy();
    println!("{} <{}>\n", name, email);

    let message = raw::git_commit_message(commit);
    println!("{}", CStr::from_ptr(message).to_string_lossy());
}
```

Получив указатель на `git_commit`, функция `show_commit` вызывает `git_commit_author` и `git_commit_message`, чтобы получить необходимую ей информацию. Обе эти функции следуют соглашению, которое в документации по `libgit2` объясняется следующим образом:

> *Если функция возвращает объект в качестве значения, то она является аксессором чтения, и время жизни этого объекта привязано к родительскому объекту.*

В терминах Rust `author` и `message` заимствованы у `commit`: `show_commit` не обязана освобождать их самостоятельно, но не должна хранить их после освобождения `commit`. Поскольку в этом API используются простые указатели, Rust не проверяет за нас их время жизни: если мы случайно создадим висячие указатели, то, возможно, узнаем об этом, только когда программа «грохнется».

В приведенном выше коде предполагается, что поля содержат текст в кодировке UTF-8, что на самом деле не так. Git допускает и другие кодировки. Для правильной интерпретации этих строк, вероятно, потребуется крейт `encoding`. Но для краткости мы не будем останавливаться на этом моменте.

Функция `main` нашей программы выглядит так:

```
use std::ffi::CString;
use std::mem;
use std::ptr;
use std::os::raw::c_char;

fn main() {
    let path = std::env::args().skip(1).next()
        .expect("usage: git-toy PATH");
    let path = CString::new(path)
        .expect("path contains null characters");

    unsafe {
        check("initializing library", raw::git_libgit2_init());

        let mut repo = ptr::null_mut();
        check("opening repository",
            raw::git_repository_open(&mut repo, path.as_ptr()));

        let c_name = b"HEAD\0".as_ptr() as *const c_char;
        let mut oid = mem::uninitialized();
        check("looking up HEAD",
            raw::git_reference_name_to_id(&mut oid, repo, c_name));
```

```
        let mut commit = ptr::null_mut();
        check("looking up commit",
            raw::git_commit_lookup(&mut commit, repo, &oid));

        show_commit(commit);

        raw::git_commit_free(commit);

        raw::git_repository_free(repo);

        check("shutting down library", raw::git_libgit2_shutdown());
    }
}
```

Она начинается с кода для обработки пути в командной строке и инициализации библиотеки, все это мы уже видели. А вот и новый код:

```
let mut repo = ptr::null_mut();
check("opening repository",
    raw::git_repository_open(&mut repo, path.as_ptr()));
```

Функция `git_repository_open` пытается открыть репозиторий Git с указанным путем. Если это получается, то она выделяет память для нового объекта `git_repository` и присваивает его адрес указателю `repo`. Rust неявно преобразует ссылки в простые указатели, поэтому переданный аргумент `&mut repo` преобразуется в `*mut *mut git_repository`, чего и ожидает функция.

Это иллюстрация еще одного соглашения `libgit2`. Снова процитируем документацию:

> *Объекты, возвращаемые с помощью первого аргумента типа «указатель на указатель», принадлежат вызывающей стороне, которая несет ответственность за их освобождение.*

В терминах Rust функции типа `git_repository_open` передают владение новым значением вызывающей стороне.

Далее рассмотрим код, который ищет объектный хеш текущей головной фиксации в репозитории:

```
let mut oid = mem::uninitialized();
check("looking up HEAD",
    raw::git_reference_name_to_id(&mut oid, repo, c_name));
```

В типе `git_oid` хранится идентификатор объекта – 160-разрядный хеш-код, который внутри Git (и в его достойном восхищения пользовательском интерфейсе) используется для идентификации фиксаций, версий отдельных файлов и много чего еще. Это обращение к `git_reference_name_to_id` ищет идентификатор объекта текущей фиксации `"HEAD"`.

В C вполне допустимо инициализировать переменную путем передачи указателя на нее некоторой функции, которая заполняет значение; ожидается, что именно так `git_reference_name_to_id` интерпретирует свой первый аргумент. Но Rust не позволяет заимствовать ссылку на неинициализированную переменную. Можно было бы инициализировать `oid` нулями, но это лишняя трата времени: любое хранящееся в ней значение будет перезаписано.

Инициализация `oid` значением `uninitialized()` решает проблему. Функция `std::mem::uninitialized` возвращает значение любого нужного нам типа, только

вот это значение содержит случайные биты, для порождения которых никакой машинный код не выполнялся. Тем не менее Rust считает, что переменной `oid` присвоено какое-то значение, и позволяет заимствовать ссылку на нее. Понятно, что в общем случае это совершенно небезопасная практика. Чтение неинициализированного значения – неопределенное поведение, а если какая-то часть этого значения реализует характеристику `Drop`, то даже его уничтожение является неопределенным поведением. Есть лишь несколько небезопасных действий с такими значениями:

- его можно перезаписать функцией `std::ptr::write`, которая требует, чтобы конечное значение было неинициализированным;
- его можно передать функции `std::mem::forget`, которая принимает владение своим аргументом и заставляет его исчезнуть без уничтожения (применение этой функции к инициализированному значению может привести к утечке ресурсов);
- или его можно передать иноязычной функции, которая его инициализирует, как то делает `git_reference_name_to_id`.

Если вызов завершается успешно, то `oid` становится инициализированной, и всё хорошо. Если же происходит ошибка, то функция не использует `oid` и уничтожать ее не нужно, так что код безопасен и в этом случае.

Мы могли бы использовать функцию `uninitialized()` также для переменных `repo` и `commit`, но поскольку они занимают всего по одному слову, а использование `uninitialized()` обставлено таким количеством оговорок, то лучше просто инициализировать их нулем.

```
let mut commit = ptr::null_mut();
check("looking up commit",
    raw::git_commit_lookup(&mut commit, repo, &oid));
```

Функция принимает идентификатор объекта фиксации и ищет соответствующую фиксацию, сохраняя указатель на `git_commit` в аргументе `commit` в случае успеха.

Остальная часть функции `main` не требует пояснений. Вызывается определенная выше функция `show_commit`, затем объекты фиксации и репозитория освобождаются и производится очистка.

Теперь можно попробовать нашу программу на любом репозитории Git, имеющемся под рукой:

```
$ cargo run /home/jimb/rbattle
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/git-toy /home/jimb/rbattle`
Jim Blandy <jimb@red-bean.com>

Animate goop a bit.

$
```

Безопасный интерфейс к libgit2

Низкоуровневый интерфейс к `libgit2` – прекрасный пример небезопасного средства: нет сомнений, что его можно использовать правильно (как мы и поступили выше, по крайней мере, надеемся на это), но Rust не может проконтро-

лировать соблюдение всех обязательных правил. Проектирование безопасного API для работы с библиотекой заключается в выявлении всех таких правил и нахождении способов превратить любое их нарушение в ошибку контроля типов или заимствования.

Перечислим правила `libgit2`, относящиеся к средствам, используемым в нашей программе:

- до использования любой библиотечной функции необходимо вызвать `git_libgit2_init`. После вызова `git_libgit2_shutdown` нельзя использовать никакие библиотечные функции;
- все значения, передаваемые функциям `libgit2`, должны быть инициализированы, за исключением выходных параметров;
- если вызов завершается неудачно, то выходные параметры, в которые должны быть помещены результаты вызова, остаются неизменными, и использовать их значения запрещается;
- объект `git_commit` ссылается на объект `git_repository`, из которого он получен, поэтому первый не должен жить дольше второго (это явно не оговорено в документации по `libgit2`, мы вывели это правило из присутствия определенных функций в интерфейсе, а затем подтвердили путем изучения исходного кода);
- аналогично `git_signature` всегда заимствуется у заданного `git_commit`, поэтому первый объект не должен жить дольше второго (в документации этот случай рассмотрен);
- сообщение, ассоциированное с фиксацией, а также имя и почтовый адрес автора заимствуются у фиксации и не должны использоваться после освобождения последней;
- однажды освобожденный объект `libgit2` не должен использоваться снова.

Как выясняется, можно построить интерфейс из Rust к `libgit2`, который будет контролировать соблюдение всех этих правил – либо с помощью системы типов, либо обрабатывая детали внутри себя.

Но прежде чем приступать к работе, немного изменим структуру проекта. Мы хотим, чтобы модуль `git` экспортировал безопасный интерфейс, а низкоуровневый интерфейс предыдущей программы был его закрытым подмодулем.

Дерево проекта будет выглядеть следующим образом:

```
git-toy/
├── Cargo.toml
├── build.rs
└── src/
    ├── main.rs
    └── git/
        ├── mod.rs
        └── raw.rs
```

Согласно правилам, сформулированным в разделе «Модули в отдельных файлах» главы 8, исходный код модуля `git` находится в файле `git/mod.rs`, а исходный код его подмодуля `git::raw` – в файле `git/raw.rs`.

Мы снова полностью перепишем `main.rs`. В начале кода должно находиться объявление модуля `git`:

```
mod git;
```

Затем создадим подкаталог `git` и переместим в него файл `raw.rs`:

```
$ cd /home/jimb/git-toy
$ mkdir src/git
$ mv src/raw.rs src/git/raw.rs
```

Модуль `git` должен объявить свой подмодуль `raw`. В файле `src/git/mod.rs` должна быть строчка:

```
mod raw;
```

Поскольку этот подмодуль не открытый, то остальным частям программы он не виден.

Скоро нам понадобится воспользоваться некоторыми функциями из крейта `libc`, поэтому добавим зависимость в файл `Cargo.toml`. Теперь файл выглядит так:

```
[package]
name = "git-toy"
version = "0.1.0"
authors = ["Jim Blandy <jimb@red-bean.com>"]
build = "build.rs"
[dependencies]
libc = "0.2.23"
```

В файл `src/main.rs` следует добавить соответствующий артикул `extern crate`:

```
extern crate libc;
```

Переработав структуру модулей, поговорим об обработке ошибок. Даже функция инициализации `libgit2` может возвращать код ошибки, поэтому мы должны решить этот вопрос, прежде чем приступать к реализации. В идиоматическом интерфейсе Rust должен быть специальный тип `Error`, в котором запоминается код ошибки `libgit2`, а также сообщение об ошибке и ее класс, полученные от `giterr_last`. Правильно написанный тип ошибки должен реализовывать обычные характеристики `Error`, `Debug` и `Display`. Кроме того, необходим соответствующий ему тип `Result`. Ниже приведены необходимые определения из файла `src/git/mod.rs`:

```
use std::error;
use std::fmt;
use std::result;

#[derive(Debug)]
pub struct Error {
    code: i32,
    message: String,
    class: i32
}

impl fmt::Display for Error {
    fn fmt(&self, f: &mut fmt::Formatter) -> result::Result<(), fmt::Error> {
        // Отображение значения типа `Error` сводится к отображению
        // сообщения, полученного от libgit2.
        self.message.fmt(f)
    }
}
```

```
impl error::Error for Error {
    fn description(&self) -> &str { &self.message }
}

pub type Result<T> = result::Result<T, Error>;
```

Для проверки результата вызова библиотечных функций модулю необходима функция, которая преобразует код возврата `libgit2` в `Result`:

```
use std::os::raw::c_int;
use std::ffi::CStr;

fn check(code: c_int) -> Result<c_int> {
    if code >= 0 {
        return Ok(code);
    }

    unsafe {
        let error = raw::giterr_last();

        // libgit2 гарантирует, что (*error).message всегда не null и завершается
        // нулем, так что этот вызов безопасен.
        let message = CStr::from_ptr((*error).message)
            .to_string_lossy()
            .into_owned();

        Err(Error {
            code: code as i32,
            message,
            class: (*error).klass as i32
        })
    }
}
```

Основное отличие от функции `check` из низкоуровневой версии состоит в том, что эта функция конструирует значение типа `Error`, вместо того чтобы напечатать сообщение об ошибке и немедленно завершить программу.

Вот теперь можно заняться инициализацией `libgit2`. Безопасный интерфейс будет предоставлять тип `Repository`, представляющий открытый репозиторий Git, с методами для разрешения ссылок, поиска фиксаций и т. д. Вот определение `Repository`, также находящееся в файле `git/mod.rs`:

```
/// Репозиторий Git.
pub struct Repository {
    // Это всегда должен быть указатель на активную структуру `git_repository`.
    // Никакой другой `Repository` не может указывать на нее.
    raw: *mut raw::git_repository
}
```

Поле `raw` структуры `Repository` не открыто. Поскольку обращаться к указателю `raw::git_repository` может только код самого модуля, то если модуль будет написан правильно, то и указатель всегда будет использоваться корректно.

Если создать `Repository` можно только в результате успешного открытия репозитория Git, то каждое значение типа `Repository` указывает на отличный от других объект `git_repository`:

```
use std::path::Path;

impl Repository {
    pub fn open<P: AsRef<Path>>(path: P) -> Result<Repository> {
    ensure_initialized();

        let path = path_to_cstring(path.as_ref())?;
        let mut repo = null_mut();
        unsafe {
            check(raw::git_repository_open(&mut repo, path.as_ptr()))?;
        }
        Ok(Repository { raw: repo })
    }
}
```

Поскольку единственный способ сделать что-то в безопасном интерфейсе – это начать со значения `Repository`, а метод `Repository::open` первым делом вызывает `ensure_initialized`, мы можем быть уверены, что `ensure_initialized` будет вызвана раньше, чем какие-либо функции из `libgit2`. Вот ее определение:

```
use std;
use libc;

fn ensure_initialized() {
    static ONCE: std::sync::Once = std::sync::ONCE_INIT;
    ONCE.call_once(|| {
        unsafe {
            check(raw::git_libgit2_init())
                .expect("initializing libgit2 failed");
            assert_eq!(libc::atexit(shutdown), 0);
        }
    });
}

use std::io::Write;

extern fn shutdown() {
    unsafe {
        if let Err(e) = check(raw::git_libgit2_shutdown()) {
            let _ = writeln!(std::io::stderr(),
                             "shutting down libgit2 failed: {}",
                             e);
            std::process::abort();
        }
    }
}
```

Тип `std::sync::Once` позволяет выполнить инициализацию потокобезопасным способом. Лишь самый первый поток, обратившийся к `ONCE.call_once`, выполнит указанное замыкание. Все последующие вызовы, из того же или другого потока, блокируются, пока первый вызов не завершится, а затем немедленно возвращают управление, не выполняя замыкания еще раз. После того как замыкание выполнится, последующие вызовы `ONCE.call_once` обходятся дешево, т. к. сводятся всего лишь к атомарной загрузке флага, хранящегося в `ONCE`.

В приведенном выше коде инициализирующее замыкание вызывает функцию `git_libgit2_init` и проверяет результат. Оно просто использует `expect`, чтобы проверить успешность инициализации, не пытаясь передать ошибку вызывающей стороне.

Чтобы программа наверняка вызвала функции `git_libgit2_shutdown`, инициализирующее замыкание пользуется библиотечной функцией C `atexit`, принимающей указатель на функцию, которую следует вызвать перед завершением процесса. Замыкания Rust не могут служить в роли указателей на функции в смысле C: замыкание – это значение некоторого анонимного типа, в котором хранятся значения захваченных переменных или ссылки на них, а указатель на C-функцию – это просто указатель. Однако Rust'овские типы `fn` прекрасно работают, нужно лишь объявить их `extern`, чтобы Rust понимал, что нужно использовать соглашения о вызове, принятые в C. Локальная функция `shutdown` удовлетворяет всем условиям и гарантирует надлежащую очистку `libgit2`.

В разделе «Раскрутка стека» главы 7 мы отмечали, что паника через границы языков – неопределенное поведение. Вызов `shutdown` из `atexit` – пример такой границы, поэтому важно, чтобы `shutdown` не паниковала. Именно поэтому `shutdown` не может просто использовать `.expect` для обработки ошибок, полученных от `raw::git_libgit2_shutdown`. Вместо этого она должна сообщить об ошибке и самостоятельно завершить процесс. Спецификация POSIX запрещает вызывать `exit` из обработчика `atexit`, поэтому `shutdown` вызывает `std::process::abort`, чтобы завершить программу безо всяких околичностей.

В принципе, можно было бы организовать обращение к `git_libgit2_shutdown` раньше – скажем, после уничтожения последнего значения `Repository`. Но как бы ни была устроена программа, вызов `git_libgit2_shutdown` – непременная обязанность безопасного API. После ее вызова использование всех оставшихся объектов `libgit2` становится небезопасным, поэтому безопасный API не должен давать прямого доступа к этой функции.

Простой указатель внутри `Repository` всегда должен указывать на активный объект `git_repository`. Отсюда следует, что единственный способ закрыть репозиторий – уничтожить владеющее им значение `Repository`:

```
impl Drop for Repository {
    fn drop(&mut self) {
        unsafe {
            raw::git_repository_free(self.raw);
        }
    }
}
```

Поскольку `git_repository_free` вызывается только непосредственно перед уничтожением единственного указателя на `raw::git_repository`, тип `Repository` заодно гарантирует, что указатель невозможно будет использовать после освобождения.

Метод `Repository::open` пользуется закрытой функцией `path_to_cstring`, имеющей два определения – для Unix и для Windows:

```
use std::ffi::CString;

#[cfg(unix)]
fn path_to_cstring(path: &Path) -> Result<CString> {
```

```
    // Метод `as_bytes` существует только в Unix-системах.
    use std::os::unix::ffi::OsStrExt;

    Ok(CString::new(path.as_os_str().as_bytes())?)
}

#[cfg(windows)]
fn path_to_cstring(path: &Path) -> Result<CString> {
    // Пытаемся преобразовать в UTF-8. Если не получается, то и libgit2
    // не сможет обработать такой путь.
    match path.to_str() {
        Some(s) => Ok(CString::new(s)?),
        None => {
            let message = format!("Couldn't convert path '{}' to UTF-8",
                                  path.display());
            Err(message.into())
        }
    }
}
```

Интерфейс `libgit2` несколько усложняет этот код. На любой платформе `libgit2` принимает пути в виде C-строк, завершающихся нулем. В Windows `libgit2` предполагает, что такая C-строка содержит корректную последовательность символов UTF-8 и внутри себя преобразует ее в 16-разрядные пути, которых требует Windows. Обычно эта схема работает, но она не идеальна. Windows допускает имена файлов, не являющиеся правильными с точки зрения Юникода и, стало быть, не представимые в кодировке UTF-8. Имя такого файла передать `libgit2` невозможно.

В Rust для представления пути в файловой системе служит тип `std::path::Path`, спроектированный так, чтобы поддержать любой путь, удовлетворяющий спецификациям Windows или POSIX. Это означает, что в Windows существуют такие значения `Path`, которые невозможно передать `libgit2`, поскольку они не являются корректными с точки зрения UTF-8. Таким образом, хотя поведение `path_to_cstring` не идеально, это лучшее, что можно сделать с имеющимся интерфейсом `libgit2`.

Оба определения `path_to_cstring` опираются на преобразования нашего типа `Error`: оператор `?` пытается произвести такое преобразование, а версия для Windows явно вызывает метод `.into()`. Сами преобразования ничем не примечательны:

```
impl From<String> for Error {
    fn from(message: String) -> Error {
        Error { code: -1, message, class: 0 }
    }
}

// NulError - это то, что возвращает `CString::new`, когда в строке
// встречаются нулевые байты.
impl From<std::ffi::NulError> for Error {
    fn from(e: std::ffi::NulError) -> Error {
        Error { code: -1, message: e.to_string(), class: 0 }
    }
}
```

Далее разберемся, как разрешить ссылку Git на идентификатор объекта. Поскольку идентификатор объекта – это просто 20-байтовый хеш-код, ничто не мешает сделать его видимым в безопасном API:

```
/// Идентификатор какого-то объекта, хранящегося в базе данных Git:
/// фиксации, дерева, блоба, тега и т. д. Это результат хеширования
/// содержимого объекта.
pub struct Oid {
    pub raw: raw::git_oid
}
```

Добавим в тип `Repository` метод для выполнения поиска:

```
use std::mem::uninitialized;
use std::os::raw::c_char;

impl Repository {
    pub fn reference_name_to_id(&self, name: &str) -> Result<Oid> {
        let name = CString::new(name)?;
        unsafe {
            let mut oid = uninitialized();
            check(raw::git_reference_name_to_id(&mut oid, self.raw,
                                                name.as_ptr() as *const c_char))?;
            Ok(Oid { raw: oid })
        }
    }
}
```

Хотя `oid` остается неинициализированным, если поиск неудачен, эта функция гарантирует, что вызывающая сторона никогда не увидит неинициализированное значение. Для этого она просто придерживается идиомы `Result`: вызывающая сторона получает либо `Ok` с правильно инициализированным значением `Oid`, либо `Err`.

Далее модулю необходимо как-то извлекать фиксации из репозитория. Определим тип `Commit`:

```
use std::marker::PhantomData;

pub struct Commit<'repo> {
    // Это должен быть указатель на пригодную к использованию структуру `git_commit`.
    raw: *mut raw::git_commit,
    _marker: PhantomData<&'repo Repository>
}
```

Как уже отмечалось, объект `git_commit` не должен жить дольше объекта `git_repository`, от которого он получен. Время жизни Rust позволяет точно отразить это правило.

В примере `RefWithFlag` выше в этой главе использовался член типа `PhantomData`, который говорит Rust, что тип нужно рассматривать так, будто он содержит ссылку с указанным временем жизни, хотя на самом деле такая ссылка в типе явно не присутствует. В типе `Commit` нужно сделать нечто подобное. В данном случае поле `_marker` имеет тип `PhantomData<&'repo Repository>`, что служит для Rust указанием рассматривать тип `Commit<'repo>` так, будто в нем хранится ссылка со временем жизни `'repo` на какое-то значение `Repository`.

Вот код метода поиска фиксации:

```
use std::ptr::null_mut;

impl Repository {
    pub fn find_commit(&self, oid: &Oid) -> Result<Commit> {
        let mut commit = null_mut();
        unsafe {
            check(raw::git_commit_lookup(&mut commit, self.raw, &oid.raw))?;
        }
        Ok(Commit { raw: commit, _marker: PhantomData })
    }
}
```

Как это позволяет соотнести время жизни `Commit` со временем жизни `Repository`? В сигнатуре `find_commit` опущены времена жизни ссылок в полном соответствии с правилами, изложенными в разделе «Опускание параметрического времени жизни» главы 5. Если указать времена жизни явно, то сигнатура будет выглядеть так:

```
fn find_commit<'repo, 'id>(&'repo self, oid: &'id Oid)
    -> Result<Commit<'repo>>
```

Это именно то, что нам нужно: Rust рассматривает возвращенное значение `Commit`, как если бы оно заимствовало что-то у `self`, а это и есть `Repository`.

При уничтожении значения `Commit` оно должно освободить свой объект `raw::git_commit`:

```
impl<'repo> Drop for Commit<'repo> {
    fn drop(&mut self) {
        unsafe {
            raw::git_commit_free(self.raw);
        }
    }
}
```

У `Commit` можно заимствовать `Signature` (имя и почтовый адрес автора) и текст сообщения, сопровождающего фиксацию:

```
impl<'repo> Commit<'repo> {
    pub fn author(&self) -> Signature {
        unsafe {
            Signature {
                raw: raw::git_commit_author(self.raw),
                _marker: PhantomData
            }
        }
    }

    pub fn message(&self) -> Option<&str> {
        unsafe {
            let message = raw::git_commit_message(self.raw);
            char_ptr_to_str(self, message)
        }
    }
}
```

Вот как выглядит тип `Signature`:

```
pub struct Signature<'text> {
    raw: *const raw::git_signature,
    _marker: PhantomData<&'text str>
}
```

Объект `git_signature` всегда заимствует свой текст откуда-то еще; в частности, сигнатуры, возвращаемые функцией `git_commit_author`, заимствуют текст у `git_commit`. Поэтому наш безопасный тип `Signature` включает поле типа `PhantomData<&'text str>`, которое говорит Rust, что нужно вести себя так, будто в нем имеется поле типа `&str` со временем жизни `'text`. Как и раньше, метод `Commit::author` правильно связывает это время жизни `'text` возвращаемого им значения `Signature` со временем жизни `Commit`, нам даже писать ничего не нужно. Метод `Commit::message` делает то же самое с полем типа `Option<&str>`, содержащим сообщение, ассоциированное с фиксацией.

Тип `Signature` располагает методами для получения имени и почтового адреса автора:

```
impl<'text> Signature<'text> {
    /// Возвращает имя автора в виде `&str`
    /// или `None`, если это не корректный текст в кодировке UTF-8.
    pub fn name(&self) -> Option<&str> {
        unsafe {
            char_ptr_to_str(self, (*self.raw).name)
        }
    }

    /// Возвращает почтовый адрес автора в виде `&str`
    /// или `None`, если это не корректный текст в кодировке UTF-8.
    pub fn email(&self) -> Option<&str> {
        unsafe {
            char_ptr_to_str(self, (*self.raw).email)
        }
    }
}
```

Описанные выше методы пользуются закрытой вспомогательной функцией `char_ptr_to_str`:

```
/// Пытается заимствовать `&str` у `ptr`, учитывая, что `ptr` может быть нулевым
/// или ссылаться на некорректную последовательноть UTF-8. Придает результату
/// время жизни, как если бы он был заимствован у `_owner`.
///
/// Безопасность: если `ptr` не равен null, то он должен указывать на завершающуюся
/// нулем C-строку, к которой можно безопасно обращаться.
unsafe fn char_ptr_to_str<T>(_owner: &T, ptr: *const c_char) -> Option<&str> {
    if ptr.is_null() {
        return None;
    } else {
        CStr::from_ptr(ptr).to_str().ok()
    }
}
```

Само значение параметра `_owner` нигде не используется, но используется его время жизни. Явно указав времена жизни, получим такую сигнатуру функции:

```
fn char_ptr_to_str<'o, T: 'o>(_owner: &'o T, ptr: *const c_char)
    -> Option<&'o str>
```

Функция `CStr::from_ptr` возвращает значение типа `&CStr` с ничем не ограниченным временем жизни, поскольку оно было заимствовано у разыменованного простого указателя. Неограниченное время жизни почти всегда неточно, поэтому лучше бы ограничить его как можно раньше. Наличие параметра `_owner` заставляет Rust назначить его время жизни типу возвращаемого значения, так что вызывающая сторона может получить более точно ограниченную ссылку.

Из документации по библиотеке `libgit2` не ясно, могут ли указатели `email` и `author` в `git_signature` быть нулевыми, хотя в целом документация очень хороша. Авторы немного покопались в исходном коде, но не смогли прийти к какому-то определенному выводу, а потому решили, что на всякий случай лучше, если `char_ptr_to_str` будет готова к нулевым указателям. В Rust на такие вопросы сразу же отвечает сам тип: если это `&str`, значит, можно рассчитывать на наличие строки, а если `Option<&str>` – то строки может и не быть.

Вот мы и предоставили безопасные интерфейсы ко всей необходимой функциональности. Новая функция `main` в файле `src/main.rs` значительно «похудела» и теперь выглядит как настоящий Rust-код:

```
fn main() {
    let path = std::env::args_os().skip(1).next()
        .expect("usage: git-toy PATH");

    let repo = git::Repository::open(&path)
        .expect("opening repository");

    let commit_oid = repo.reference_name_to_id("HEAD")
        .expect("looking up 'HEAD' reference");

    let commit = repo.find_commit(&commit_oid)
        .expect("looking up commit");

    let author = commit.author();
    println!("{} <{}>\n",
             author.name().unwrap_or("(none)"),
             author.email().unwrap_or("none"));

    println!("{}", commit.message().unwrap_or("(none)"));
}
```

В этом разделе мы разработали безопасный API поверх небезопасного, сделав так, что любое нарушение контракта последнего оказывается ошибкой типизации Rust. В результате получился интерфейс, правильность использования которого Rust может проконтролировать. По большей части правила, которые мы поручили контролировать Rust, – это именно те правила, которые программисты на C и C++ все равно соблюдают добровольно. И большая строгость Rust, по сравнению с C и C++, состоит не в том, что он навязывает чужеродные правила, а в том, что контроль их соблюдения механический и исчерпывающий.

Заключение

Rust – не простой язык. Его цель – объять два очень разных мира. Это современный язык программирования, безопасный по построению, с такими удобствами, как замыкания и итераторы. И в то же время он позволяет программисту контролировать низкоуровневые средства компьютера, на котором исполняется, с минимальными накладными расходами.

Контуры языка определяются этими целями. Rust ухитряется навести почти безупречный мост к безопасному коду. Механизм контроля заимствования и абстракции с нулевой стоимостью дают возможность максимально приблизиться к «железу», не рискуя получить неопределенное поведение. Если этого недостаточно или если мы хотим воспользоваться существующим кодом на C, то к нашим услугам небезопасный код. Но и в этом случае язык не просто вручает небезопасные средства и желает удачи. Цель всегда состоит в том, чтобы на основе небезопасных средств построить безопасный API. Именно так мы и поступили с библиотекой `libgit2`. Разработчики Rust проделали то же самое с типами `Box`, `Vec`, другими коллекциями, каналами и многим другим – стандартная библиотека полна абстракций, которые под капотом реализованы с помощью небезопасного кода.

Языку с такими амбициями, как у Rust, пожалуй, не суждено стать простейшим из имеющихся инструментов. Но Rust безопасный, быстрый, конкурентный – и при этом эффективный. Применяйте его для создания крупных, быстрых, защищенных, надежных систем, в полной мере задействующих всю мощь оборудования, на котором работают. Используйте его, чтобы сделать программное обеспечение лучше.

Предметный указатель

П

P

Об авторах

Джим Блэнди занимается программированием с 1981 года, а свободное программное обеспечение пишет с 1990 года. Он сопровождал GNU Emacs и GNU Guile, а также GDB – отладчик GNU. Является одним из проектировщиков системы управления версиями Subversion. В настоящее время работает в проекте Mozilla над средствами разработчика для браузера Firefox.

Джейсон Орендорф пишет на C++ для проекта Mozilla, в котором является владельцем модуля движка JavaScript в Firefox. Активный член сообщества разработчиков в Нэшвилле, время от времени организует местные встречи технических специалистов. Интересуется грамматикой, выпечкой, путешествиями во времени, помогает другим осваивать сложные темы.

Колофон

На обложке книги изображен краб Монтэгю (Xantho hydrophilus).

У этого мускулистого, коренастого на вид краба широкий панцирь шириной примерно 70 мм. Край панциря изрезан бороздками, цвет варьируется от желтоватого до красно-коричневого. У краба 10 ног (пять пар): передние ноги (клешненосные) одинаковы по размеру и заканчиваются клешнями с кончиками черного цвета. За ними находятся три пары ходильных ног, мощных и сравнительно коротких. Последняя пара ног предназначена для плавания. Краб ходит и плавает боком.

Краб Монтэгю встречается в северо-восточной части Атлантического океана и в Средиземном море. Обитает под камнями и валунами. Если поднять камень, под которым находится краб, он агрессивно поднимает и широко раскрывает клешни, чтобы казаться больше.

Питается водорослями, улитками и крабами других видов. Активен преимущественно ночью. Самки откладывают яйца с марта по июль, личинки встречаются в планктоне большую часть лета.

Многие животные, изображенные на обложках книг O'Reilly, находятся под угрозой вымирания; все они важны для нашего мира. Если хотите узнать, чем вы можете помочь, зайдите на сайт *animals.oreilly.com*.

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: books@alians-kniga.ru.

Джим Блэнди, Джейсон Орендорф

Программирование на языке Rust

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать офсетная.
Усл. печ. л. 51,5625. Тираж 400 экз.

Веб-сайт издательства: **www.дмк.рф**