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: