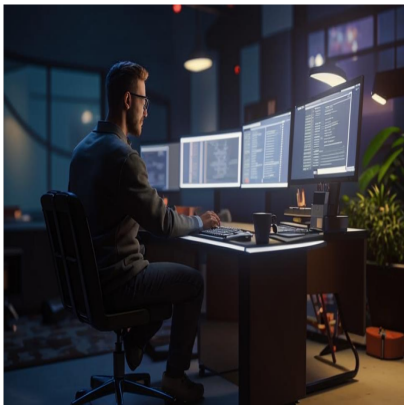


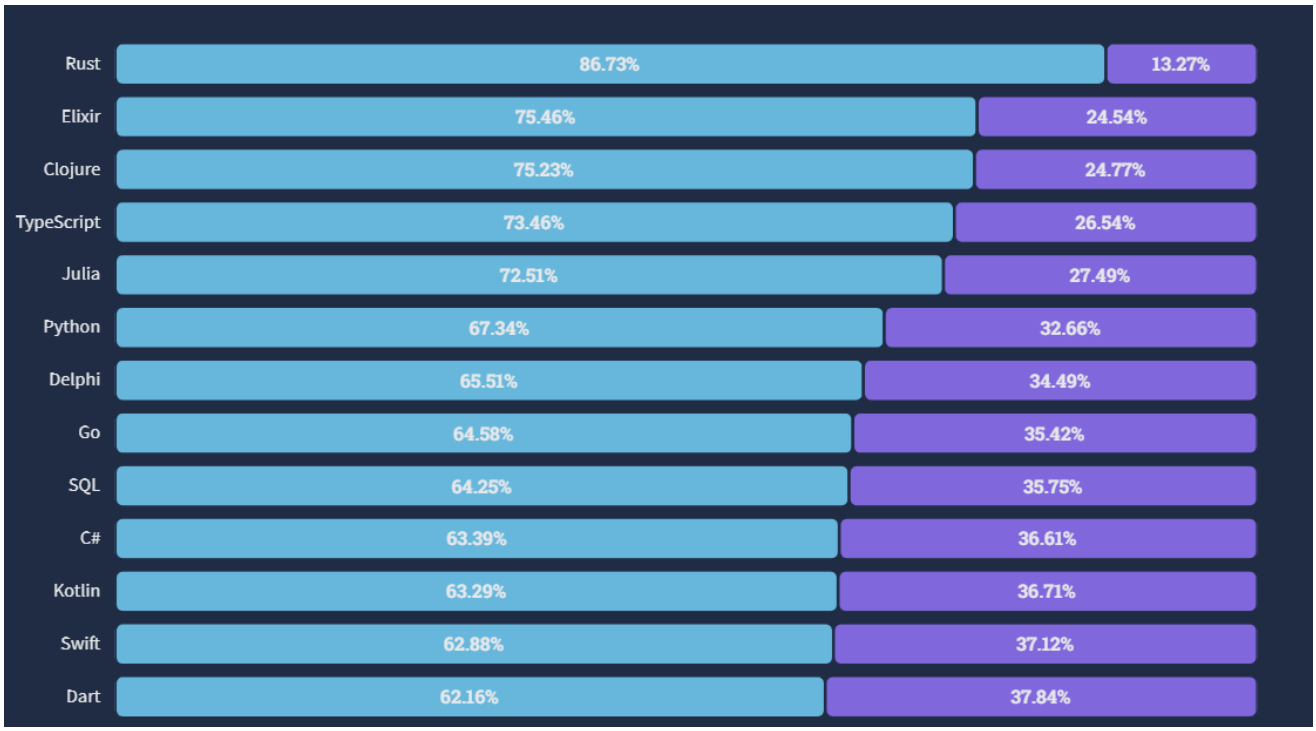
7 причин использовать Rust для вашего следующего проекта по разработке

Описание

Вы хотите изучить язык Rust? Чтобы помочь вам определиться с выбором, в этой статье мы расскажем о некоторых особенностях Rust, одного из самых популярных языков системного программирования. В этой статье мы рассмотрим язык программирования Rust и его особенности, такие как система типов, безопасность памяти и владение. Мы также рассмотрим список ресурсов, которые могут помочь вам в изучении Rust. Давайте начнем!

Что такое Rust?

Rust – это язык системного программирования. Он возник как личный проект разработчика Грейдона Хоара в 2006 году. Менее чем за десять лет он превратился в лучший выбор для системного программирования и смежных приложений. Средняя зарплата программиста на Rust составляет около 120 тысяч долларов. Так что если вы хотите перейти с C++ на Rust или хотите освоить новый язык, изучение Rust может стать отличным выбором! Согласно опросу разработчиков StackOverflow, Rust признан самым любимым языком программирования – уже семь лет подряд.



Rust предлагает **скорость** низкоуровневых языков системного программирования, таких как C и C++, и **безопасность** высокоуровневых языков программирования, таких как Python. Rust широко используется во всех областях разработки программного обеспечения- от таких известных проектов, как Dropbox и Firefox, до WebAssembly и встраиваемого программирования. Rust предлагает встроенную поддержку управления пакетами через Cargo.

Cargo: менеджер пакетов для Rust

Cargo – это менеджер пакетов для Rust. Вы можете использовать cargo для установки пакетов из crates, реестра пакетов Rust. Помимо менеджера пакетов, позволяющего искать, устанавливать и управлять пакетами, cargo также служит в качестве прогонщика тестов, генератора документации и системы сборки. Теперь, когда вы получили общее представление о Rust, давайте подробнее рассмотрим некоторые его особенности, которые выделяют его как язык системного программирования, получивший широкое распространение.

Полезные сообщения об ошибках

Как начинающий программист, вы будете сталкиваться с ошибками и тратить значительное количество времени на отладку своего кода. Вы будете использовать сообщения об ошибках и предупреждения, выдаваемые компилятором, для устранения этих проблем. А полезные сообщения помогут вам ускорить отладку.

Пример сообщения об ошибке

Когда ваш код не компилируется успешно, Rust выдает полезные сообщения об ошибках, в которых объясняется, что и где нужно исправить в вашем коде. В этом примере переменная `num2` определена внутри функции `inner()`. Поэтому она ограничена областью видимости функции. Если вы попытаетесь получить к ней доступ вне функции, компилятор выдаст ошибку:

```
fn main() {
    let num1 = 10;
    fn inner(){
        let num2 = 9;
    }
    println!("The value of num2 is: {}", num2);
}
```

А сообщение об ошибке дает информацию о том, что нужно исправить.

```
error[E0425]: cannot find value `num2` in this scope
 --> src/main.rs:6:42
  |
6 |     println!("The value of num2 is: {}", num2);
  |                                          ^^^^ help: a local variable with
```

Предупреждения во время компиляции

Компилятор также выдает полезные предупреждения о проблемах в вашем коде. Если вы определяете переменные, но никогда не используете их в остальной части программы, Rust выдает предупреждение, как показано на рисунке.

```
fn main() {
    let num1 = 10;
    let num2 = 9;
    println!("The value of num1 is: {}", num1);
}
```

Здесь переменная num2 объявлена, но никогда не используется.

```
warning: unused variable: `num2`
 --> src/main.rs:3:9
  |
3 |     let num2 = 9;
  |         ^^^^ help: if this is intentional, prefix it with an underscore: `
  |
```

Сильно типизированный язык

Еще одна причина, по которой вам стоит использовать Rust в своих проектах, – это его **система типов**. Rust – **сильно типизированный язык**, что означает, что он не поддерживает согласование типов. Принуждение типов – это когда язык может неявно преобразовать значение одного типа данных в другой. Например, код Python в следующей ячейке кода будет выполняться без ошибок. Это происходит потому, что в Python ненулевое число имеет истинностное значение True, и поэтому оператор if выполняется без ошибок, несмотря на то, что число 10 является целым

числом, а не булевым.

```
num1 = 10
if num1:
    num2 = 9
print(f"num2 is {num2}")

# Output: num2 is 9
```

С другой стороны, Rust не коэрцитирует типы. Поэтому следующий код приведет к ошибке:

```
fn main() {
    let num1 = 10;
    if num1{
        let num2 = 9;
    }
}
```

Ошибка сообщает о несоответствии типов, где ожидалось булево, а оказалось целое число.

```
error[E0308]: mismatched types
 --> src/main.rs:3:8
  |
3 |     if num1{
  |        ^^^^ expected `bool`, found integer
```

Копировать

Безопасность памяти

Безопасность памяти – еще одна особенность Rust, которая делает его привлекательным для программистов. Мы постараемся дать беглый обзор того, как это работает.

Переменные должны быть инициализированы до их использования

В Rust все переменные *должны* быть инициализированы, прежде чем их можно будет использовать. В таких языках, как C, следующий код, в котором `num` не инициализирована, скомпилируется и выполнится без ошибок. Значением неинициализированной переменной будет какое-то мусорное значение.

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

```
int main(void) {
    int num;
 printf("The value of num is %d", num);
 return 0;
}
// Output: The value of num is 0
```

Если вы попытаетесь сделать нечто подобное в Rust, то столкнетесь с ошибкой компиляции. Поэтому в Rust нет понятия сборки мусора.

```
fn main() {
    let num:i32;
    println!("The value of num is: {}",num);
}
```

```
error[E0381]: used binding `num` isn't initialized
 --> src/main.rs:3:40
  |
2 |     let num:i32;
  |         --- binding declared here but left uninitialized
3 |     println!("The value of num is: {}",num);
  |                                        ^^^ `num` used here but it isn't in
  |
```

Безопасность памяти на этапе компиляции

Rust обеспечивает безопасность памяти во время компиляции. Давайте рассмотрим простой пример. Здесь, несмотря на то что условный оператор if имеет булево значение `true`, что означает, что значение `num` всегда будет равно 100, мы получаем ошибку при попытке вывести значение `num`.

```
fn main() {
    let num:i32;
    if true{
        num = 100;
    }
    println!("The value of num is: {}", num);
}
```

Это связано с тем, что условная оценка происходит во время выполнения, и компилятор не сможет гарантировать, что `num` имеет значение во время компиляции.

```
error[E0381]: used binding `num` is possibly-uninitialized
 --> src/main.rs:6:41
  |
2 |     let num:i32;
  |         --- binding declared here but left uninitialized
3 |     if true{
  |        ---- if this `if` condition is `false`, `num` is not initialized
4 |         num = 100;
```

```
5 |     }
  |      - an `else` arm might be missing here, initializing `num`
6 |     println!("The value of num is: {}", num);
  |                                         ^^^ `num` used here but it is poss
```

Если вы внимательно посмотрите на сообщение об ошибке, то увидите, что с помощью оператора else мы можем гарантировать, что num будет иметь значение. Поэтому следующий код будет выполняться без ошибок. Потому что таким образом компилятор может определить, что num будет иметь значение – во время компиляции – поэтому ошибок не будет.

```
fn main() {
    let num:i32;
    if true{
        num = 100;
    }
    else{
        num = 50;
    }
    println!("The value of num is: {}", num);
}
```

```
The value of num is: 100
```

Неизменность переменных

Также полезно отметить, что переменные в Rust по умолчанию неизменяемы. Это означает, что вам, как разработчику, не нужно беспокоиться о том, что вы можете случайно переписать значение определенной переменной. Вот пример:

```
fn main() {
    let num1 = 10;
    num1 = 5;
    println!("The value of num1 is: {}", num1);
}
```

Поскольку num1 инициализирована значением 10, при попытке присвоить ей значение 5 вы получите сообщение об ошибке “cannot assign twice to immutable variable num1“.

```
error[E0384]: cannot assign twice to immutable variable `num1`
 --> src/main.rs:3:5
  |
2 |     let num1 = 10;
  |         ----
  |         |
  |         first assignment to `num1`
  |         help: consider making this binding mutable: `mut num1`
3 |     num1 = 5;
  |     ^^^^^^^^ cannot assign twice to immutable variable
```

Владение и заимствование

Владение обеспечивает безопасность памяти. Функционально владение в Rust можно описать следующим образом:

> У каждого объекта должен быть **один и только один** владелец. Если владелец выходит за пределы области видимости, то объект уничтожается.

Давайте рассмотрим простой пример. Здесь мы инициализируем строку `str1`, а затем перемещаем ее значение в `str2`. Поскольку у любого объекта может быть только один владелец, объект `str1` будет уничтожен, как только его значение переместится в `str2`.

```
fn main() {
    let str1 = String::from("Rust");
    let str2 = str1;
    println!("The value of str1 is: {}", str1);
}
```

```
error[E0382]: borrow of moved value: `str1`
 --> src/main.rs:4:42
  |
2 |     let str1 = String::from("Rust");
  |         ---- move occurs because `str1` has type `String`, which does not
3 |     let str2 = str1;
  |                ---- value moved here
4 |     println!("The value of str1 is: {}", str1);
  |                                          ^^^^ value borrowed here after mo
```

Хотя это кажется интуитивно понятным, чтобы лучше понять и оценить, как работает право собственности, полезно ознакомиться с понятиями заимствования и ссылки.

Быстрое развитие

До сих пор мы обсуждали несколько полезных возможностей языка программирования Rust. Рассмотрим некоторые из них:

- Rust оптимизирован для обеспечения скорости и безопасности.
- Он поставляется со встроенным инструментом управления пакетами и собирает систему.
- Он также имеет богатую стандартную библиотеку.

По сути, Rust предлагает все, о чем может попросить разработчик. Поэтому с помощью Rust можно быстро разрабатывать приложения с минимальной отладкой и ускоренной сборкой.

Кросс-платформенная разработка

С помощью Rust вы можете разрабатывать на любой платформе по своему усмотрению. Rust поддерживает самые распространенные платформы: Linux, MacOS и Windows. Разработка приложений в целом проста, поскольку вы можете скомпилировать исходный код Rust в исполняемый файл, не прибегая к помощи

других инструментов сборки и внешних компиляторов.

Поддерживающее сообщество

Поддерживающее сообщество значительно упрощает процесс обучения. У Rust огромная база пользователей, которая с каждым годом только растет. Популярность Rust в опросе разработчиков на StackOverflow говорит о том, что существует большое сообщество пользователей, в котором много опытных разработчиков, готовых поделиться своими знаниями и опытом. Помимо официальной документации, существует также сайт пользовательской документации и форум для обсуждения. Вы также можете проверить сабреддит Rust и группы LinkedIn, чтобы найти соответствующие обсуждения.

Учебные ресурсы для начала работы с Rust

В этом разделе перечислены некоторые полезные ресурсы, которые помогут вам начать работу с Rust. Это не исчерпывающий список, но в него включены некоторые рекомендуемые учебники, курсы и книги, которые помогут вам в освоении языка.

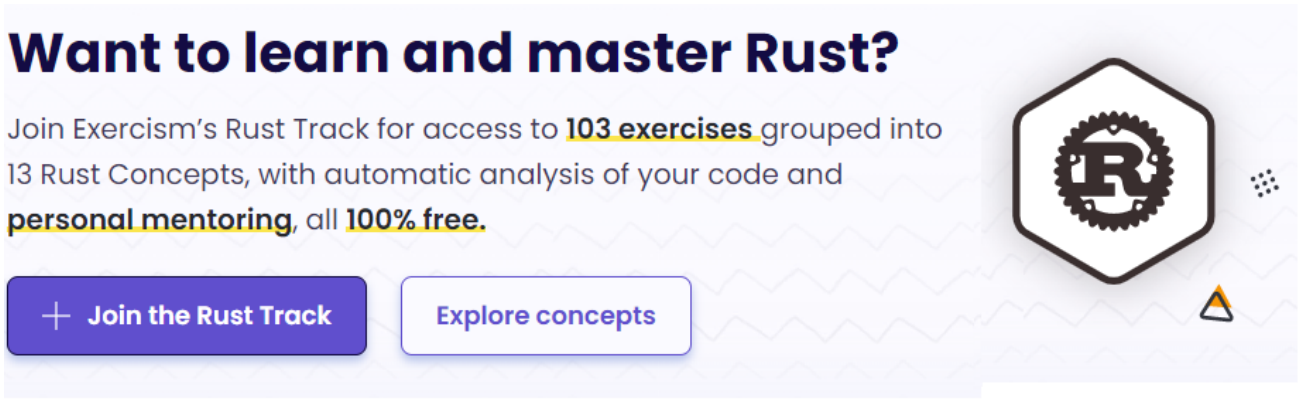
Rust By Example

Rust By Example научит вас основам Rust и стандартным библиотекам с помощью серии примеров, которые вы сможете написать в онлайн-редакторе. Будут рассмотрены такие темы, как crates, cargo: инструмент управления пакетами для Rust, generics, traits, обработка ошибок и многое другое.

Rustlings

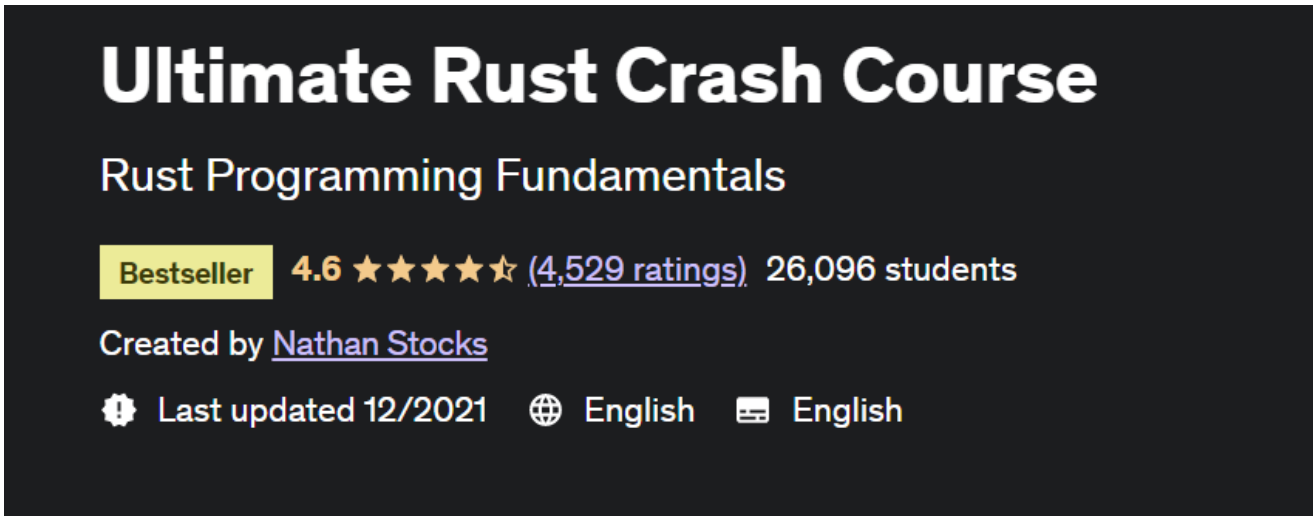
Rustlings – это еще один официальный ресурс для изучения языка программирования Rust. Он похож на “Rust на примерах”. Однако для изучения этих концепций вам потребуется создать локальную среду разработки, клонировать репозиторий примеров и решить простые задачи.

Упражнение Rust Track



Трек Rust на Exercism содержит более 100 упражнений, которые помогут вам изучить и проверить свое понимание Rust. Exercism – это бесплатная платформа, где вы можете получить наставничество от опытных программистов, а также кодить самостоятельно, выполняя упражнения.

Ultimate Rust Crash Course



Ultimate Rust Crash Course, преподаваемый Натаном Стоксом на Udemy, охватывает следующее:

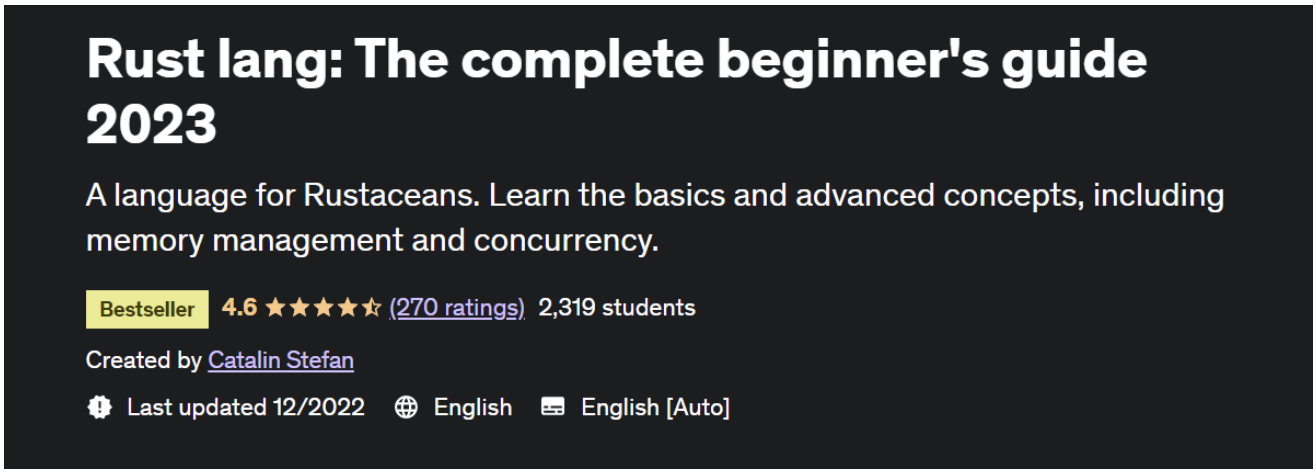
- Основы программирования на языке Rust
- Система модулей в Rust
- Типы данных и поток управления
- Справка и заимствование
- Структуры, черты и коллекции

Ultimate Rust 2

Ultimate Rust 2 является продолжением курса Ultimate Rust Crash Course и охватывает следующие темы:

- Закрытия
- Итераторы
- Обработка ошибок
- Модульное и интеграционное тестирование
- Ведение журнала, многопоточность и каналы

Rust lang: Полное руководство для начинающих 2023



Этот курс Udemy, который ведет Каталин Стефан, представляет собой всеобъемлющий курс по программированию на языке Rust. Некоторые из рассматриваемых тем включают:

- Основы Rust
- Типы данных, управляющие структуры
- Функции, черты
- Управление памятью
- Concurrency

Заключение

В этой статье был представлен обзор Rust как языка системного программирования, включая такие особенности, как безопасность памяти, улучшенное управление пакетами и многое другое. В качестве следующего шага вы можете выбрать один или несколько из упомянутых учебных ресурсов, чтобы освоить основы Rust. Счастливого программирования на Rust!

Дата Создания
19.05.2024