Замыкания и Итераторы

Андрей Ситников Илья Шпильков

27-10-2023

Спецкурс ФТШ

Closures

Анонимные функции, которые могут захватывать контекст, в котором их создали.

Зачем?

- Хочется передавать функции в аргументы, не пиша для этого ещё миллион новых функций: `.sort_by_key(…)`

Зачем?

- Хочется передавать функции в аргументы, не пиша для этого ещё миллион новых функций: `.sort_by_key(…)`
- Хочется использовать *локальные переменные* внутри этих функций.

Зачем?

- Хочется передавать функции в аргументы, не пиша для этого ещё миллион новых функций: `.sort_by_key(…)`
- Хочется использовать *локальные переменные* внутри этих функций.

Зачем?

- Хочется передавать функции в аргументы, не пиша для этого ещё миллион новых функций: `.sort_by_key(…)`
- Хочется использовать *локальные переменные* внутри этих функций.

Например, расположить элементы в массиве относительно друг друга в приоритете как в заданном массиве:

```
vec_2.sort_by_key(|value| vec.index(value))
```

Зачем?

- Хочется передавать функции в аргументы, не пиша для этого ещё миллион новых функций: `.sort_by_key(…)`
- Хочется использовать *локальные переменные* внутри этих функций.

Например, расположить элементы в массиве относительно друг друга в приоритете как в заданном массиве:

```
vec_2.sort_by_key(|value| vec.index(value))
```

Зачем?

- Хочется передавать функции в аргументы, не пиша для этого ещё миллион новых функций: `.sort_by_key(…)`
- Хочется использовать *локальные переменные* внутри этих функций.

Например, расположить элементы в массиве относительно друг друга в приоритете как в заданном массиве:

```
vec_2.sort_by_key(|value| vec.index(value))
```

Что здесь на самом деле используется?

Синтаксис

Все четыре замыкания эквивалентны:

```
1 let f1 = |x: i32, y: i32| → i32 { return x + y; };
2 let f2 = |x: i32, y: i32| { x + y };
3 let f3 = |x: i32, y: i32| x + y;
4 let f4 = |x, y| x + y;
```

- Все еще работает синтаксис с выражениями
- `{}` можно опускать, если в теле только одно выражение
- Тип аргументов может быть выведен из контекста
- Можно использовать паттерн матчинг на аргументах, например:

```
let data = vec![(2, 3), (4, 5)]
let sum = |(x, y)| x + y;
data.sort_by_key(sum);
```

Захват переменных

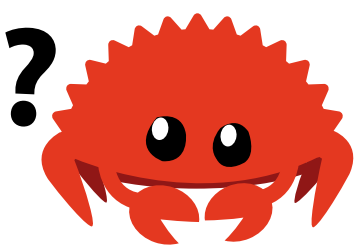
Когда внешняя переменная используется в замыкании, она **захватывается** в него.

- Если замыкание требуется *доступ на чтение* к переменной — при создании в замыкание «кладется» ссылка на чтение этой переменной.
- Если требуется *доступ на редактирование* —

Захват переменных

Когда внешняя переменная используется в замыкании, она **захватывается** в него.

- Если замыкание требуется *доступ на чтение* к переменной — при создании в замыкание «кладется» ссылка на чтение этой переменной.
- Если требуется *доступ на редактирование* —



Захват переменных

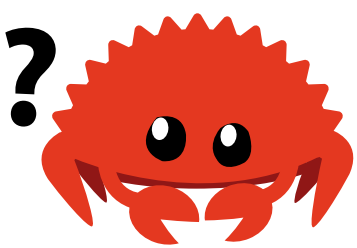
Когда внешняя переменная используется в замыкании, она **захватывается** в него.

- Если замыкание требуется *доступ на чтение* к переменной — при создании в замыкание «кладется» ссылка на чтение этой переменной.
- Если требуется *доступ на редактирование* — замыкание захватывает ссылку на редактирование.

Захват переменных

Когда внешняя переменная используется в замыкании, она **захватывается** в него.

- Если замыкание требуется *доступ на чтение* к переменной — при создании в замыкание «кладется» ссылка на чтение этой переменной.
- Если требуется *доступ на редактирование* — замыкание захватывает ссылку на редактирование.
- Если при выполнении замыкания переменная *потребляется* —



Захват переменных

Когда внешняя переменная используется в замыкании, она **захватывается** в него.

- Если замыкание требуется *доступ на чтение* к переменной — при создании в замыкание «кладется» ссылка на чтение этой переменной.
- Если требуется *доступ на редактирование* — замыкание захватывает ссылку на редактирование.
- Если при выполнении замыкания переменная *потребляется* — захват полностью перемещает переменную внутрь замыкания.

Захват переменных

```
let x = vec![1];
let mut y = vec![2];
let z = vec![3];

let f = {
    let x = &x;
    let y = &mut y;
    let z = z;
    || {
        println!("{x}, {y}, {z}");
    }
}
```

Внутренние устройство замыканий

Замыкание — это:

- Захваченные переменные
- Указатель на вызываемую функцию

move

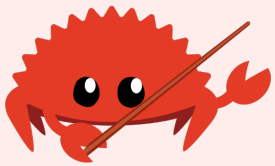
Все захваченные переменные *потребляются* `closure`.

```
 1 use std::thread;
 2
 3 fn main() {
 4     let list = vec![1, 2, 3];
 5     println!("Before defining closure: {:?}", list);
 6
 7     thread::spawn(move || println!("From thread: {:?}", list))
 8         .join()
 9         .unwrap();
10 }
```

move

Все захваченные переменные *потребляются* `closure`.

```
 1 use std::thread;
 2
 3 fn main() {
 4     let list = vec![1, 2, 3];
 5     println!("Before defining closure: {:?}", list);
 6
 7     thread::spawn(move || println!("From thread: {:?}", list))
 8         .join()
 9         .unwrap();
10 }
```



Очень полезно для потоков, чтобы можно было полностью передавать значения в новый поток, и не париться, что с ними сделает этот.

Виды замыканий

```rust
pub trait FnOnce<Args> {
  type Output;

  fn call_once(self, args: Args) → Self::Output;
}
```

```rust
pub trait FnMut<Args>: FnOnce<Args> {
  fn call_mut(&mut self, args: Args) → Self::Output;
}
```

```rust
pub trait Fn<Args>: FnMut<Args> {
  fn call(&self, args: Args) → Self::Output;
}
```

- `FnOnce` — *вообще любая функция*. Гарантируется, что её *можно вызвать один раз*.
- `FnMut` — что-то, что изменяет захваченные переменные.
- `Fn` — «чистая» функция.

- Нельзя реализовать для своих типов
- Можно использовать в сигнатурах

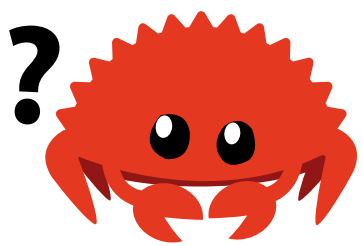
Использование `impl Trait`

```
1  fn wrap_closure<F: FnOnce()>(f: F) → impl FnOnce() {
2      || {
3          println!("here1");
4          f();
5      }
6  }
7
8  fn main() {
9      let foo = wrap_closure( || println!("here2"));
10
11     foo();
12 }
```

`impl` здесь нужен из-за того, что тип у каждого `closure` свой (и очень страшный!)

Использование `impl Trait`

```
1  fn wrap_closure<F: FnOnce()>(f: F, b: bool) → impl FnOnce() {
2      if b {
3          || f()
4      }
5      else {
6          || {}
7      }
8  }
9
10 fn main() {
11     let foo = wrap_closure( || println!("here"), false);
12
13     foo();
14 }
```



Скомпилируется?

Использование `impl Trait`

```
1  fn wrap_closure<F: FnOnce() + 'static>(f: F, b: bool) → Box<dyn
   FnOnce()> {
2      if b {
3          || f()
4      }
5      else {
6          || {}
7      }
8  }
9
10 fn main() {
11     let foo = wrap_closure( || println!("here"), false);
12
13     foo();
14 }
```

Примеры функций из стандартной библиотеки

Итераторы

Мотивация

> Пусть хотим посчитать средний балл всех учеников, которые посещали меньше половины всех занятий.

```
{
  let point_sum: usize = 0;
  for i in 0..students.len() {
    let count: usize = 0;
    let student = students[i];
    for j in 0..student.attendance.len() {
      if (student.attendance[j] == Attendance::Visited) {
        count += 1;        }
    }
    point_sum += student.score;
  }
```

```
12     point_sum / students.len()
13 }
14
```

Мотивация

> Пусть хотим посчитать средний балл всех учеников, которые посещали меньше половины всех занятий.

Можно написать на циклах в процедурном стиле, но

- писать много
- абстракции: если писать для связных списков или для деревьев/хэш-таблиц, будет либо неэффективное индексирование, либо вообще непонятно, чем индексироваться (что перебирать?). (и даже можно распараллелить, заменив одно слово).
- Философский: на циклах можно написать кучу всего, и будет выглядеть похоже.

Мотивация

> Пусть хотим посчитать средний балл всех учеников, которые посещали меньше половины всех занятий.

Можно лучше!

```
students.iter().filter(|s| {
    s.attendance.iter()
    .filter(|a| a == Attendance::Visited)
    .count() <= NUM_LECTURES / 2
}).map(Student::score).mean()
```

Ещё немного мотивации

А если кто-то из вас считает меня посредственностью или материалистом, валите на хрен учиться на Парадигмы программирования, потому что там вам самое место. Но прежде, чем вы покинете клуб победителей, я хочу, чтобы каждый из вас посмотрел на соседа.

Потому, что настанет день, в недалёком будущем, когда вы тормознёте на красный свет на старом, раздолбаном Форде, а этот человек остановится рядом с вами на блестящем Порше. И с ним будет красавица жена.

А кто будет сидеть рядом с вами? Отвратительная хабалка с трёхдневной щетиной на ногах, в растянутой майке, стиснутая между

пакетами с едой из дешёвого супермаркета. Вот кто будет сидеть рядом с вами!

Я хочу, чтобы вы решали проблемы, набивая кошелёк! А для этого вам нужно только одно: поднять крышку ноутбука и накодить слова, которым я вас учил! Я сделаю вас богаче, чем самый могучий олигарх Соединённых Штатов Америки!

Так что, слушайте меня как следует!

Вы задолжали по платежам за кредит?
Круто! Подняли крышку, открыли IDE!

Вас выгоняют со съёмной квартиры?
Круто! Подняли крышку, открыли IDE!

Ваша девушка считает вас жалким, ничтожным лузером?
Круто! Подняли крышку, открыли IDE!

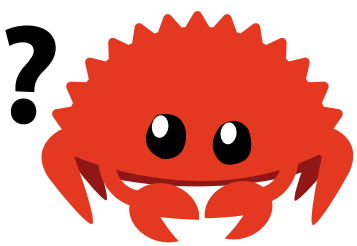
Будьте безжалостны, будьте жестоки!

Вы компьютерные, мать вашу, террористы!

Ещё немного мотивации

Пример 2: Хотим сделать `map` на массиве с преобразованием типов.

Нужно создавать новый массив,
тип другой.



Ещё немного мотивации

Пример 2: Хотим сделать `map` на массиве с преобразованием типов.

Нужно создавать новый массив,
тип другой. Наивная реализация:

```
1 let a = vec![1, 2, 3]
2 let mut b = Vec::new();
3
4 for i in 0..a.len() {
5   b.push(a[i] as f32);
6 }
```

Ещё немного мотивации

Пример 2: Хотим сделать `map` на массиве с преобразованием типов.

Нужно создавать новый массив, тип другой. Наивная реализация:

- Некрасиво

```
let a = vec![1, 2, 3]
let mut b = Vec::new();

for i in 0..a.len() {
  b.push(a[i] as f32);
}
```

Ещё немного мотивации

Пример 2: Хотим сделать `map` на массиве с преобразованием типов.

Нужно создавать новый массив, тип другой. Наивная реализация:

- Некрасиво
- Дополнительные аллокации

```
1 let a = vec![1, 2, 3]
2 let mut b = Vec::new();
3
4 for i in 0..a.len() {
5   b.push(a[i] as f32);
6 }
```

Ещё немного мотивации

Пример 2: Хотим сделать `map` на массиве с преобразованием типов.

Нужно создавать новый массив, тип другой. Наивная реализация:

```
1 let a = vec![1, 2, 3]
2 let mut b = Vec::new();
3
4 for i in 0..a.len() {
5   b.push(a[i] as f32);
6 }
```

- Некрасиво
- Дополнительные аллокации
- Плохо масштабируется и комбинируется на сложных операциях (куча `boilerplate`)

Ещё немного мотивации

Пример 2: Хотим сделать `map` на массиве с преобразованием типов.

Нужно создавать новый массив, тип другой. Наивная реализация:

```
1 let a = vec![1, 2, 3]
2 let mut b = Vec::new();
3
4 for i in 0..a.len() {
5   b.push(a[i] as f32);
6 }
```

- Некрасиво
- Дополнительные аллокации
- Плохо масштабируется и комбинируется на сложных операциях (куча `boilerplate`)
- Хочется воспользоваться *zero-cost abstractions*

Итераторы — структуры, которые управляют потоками элементов.

- Контейнеры многообразны (эффективная вставка/удаление/подсчёты статистики/балансирование деревьев)
- но итераторы позволяют посмотреть в разрезе итерации
- общий API — пройтись по всем элементам и что-то сделать

Generics, Traits, Dynamic Polynorphysm

trait Iterator

```rust
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;

    // Provided methods
    ...
}
```

- Из итератора можно извлекать элементы через `.next()`
- `.next()` отдает объект во владение
- Provided methods можно переопределить для эффективности
- *Методы-комбинаторы!*

Базовый пример

- МО реализации итераторов

```
struct Countdown {
  n: u32
}
impl Iterator for Countdown {
    type Item = u32;

    fn next(&mut self) → Option<u32> {
        if self.n == 0 {
            None
        }
        else {
            self.n -= 1;
```

```
13                 Some(self.n)
14             }
15         }
16 }
```

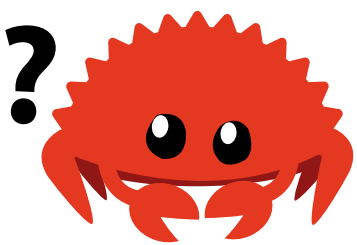
Взаимодействие с `for`

- `for` позволяет итерироваться по любому объекту, который реализует `IntoIterator`. В частности... по итератору.

```rust
pub trait IntoIterator {
  type Item;
  type IntoIter: Iterator<Item = Self::Item>;

  fn into_iter(self) -> Self::IntoIter;
}
```

```rust
impl<I: Iterator> IntoIterator for I { ... }
for good_student in students.iter().filter( ... ) { ... }
```

Типы из `std`, у которых есть итераторы

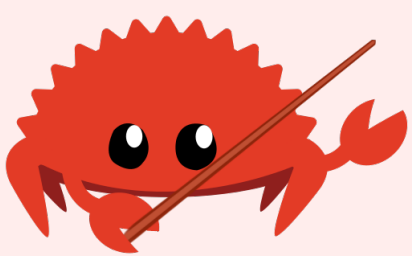


Типы из `std`, у которых есть итераторы

- `Vec`, `VecDeque`, `HashMap`, `BTreeMap`, `LinkedList`...

Типы из `std`, у которых есть итераторы

- `Vec`, `VecDeque`, `HashMap`, `BTreeMap`, `LinkedList`...
- `Option`, `Result`



`trait` можно реализовать не только для некого типа `MyType`, но и для `&'_ MyType`. Поэтому `IntoIterator` реализован не только для коллекций, но и ссылок на них.

FromIterator

```
pub trait FromIterator<A>: Sized {
  fn from_iter<T>(iter: T) → Self
     where T: IntoIterator<Item = A>;
}
```

- `trait` параметризован по типу *элемента*, и это неспроста!
- Один из самых полезных адаптеров – `Iterator::collect`

collect

```
fn collect<B>(self) → B
where
  B: FromIterator<Self::Item>,
  Self: Sized,
```

- Создание коллекции (выбранной пользователем) из элементов итератора
- Еще один пример *return type polymorhism*

Применение `map` к массиву

```
1 let a = vec![1, 2, 3];
2 let b: Vec<i32> = a.into_iter().map(|n| n as f32).collect();
```

Применение `map` к массиву

```
1 let a = vec![1, 2, 3];
2 let b = a.into_iter().map(|n| n as f32);
3 println!("{:?}", b.collect());
```

Применение `map` к массиву

```rust
let a = vec![1, 2, 3];
let b = a.into_iter().map(|n| n as f32);
println!("{:?}", b.collect::<Vec<_>>());
```

Классификация функциональности, связанной с итераторами

Вход в мир итераторов Коллекции или генераторы
Обработка Адапторы (aka комбинаторы)
Получение результата Финалайзеры

Финалайзеры

- `count` — подсчитать. Чтобы подсчитать, нам придется «вытащить» из итератора все элементы, «потребляя» его.
- `last` — вытаскивать элементы, пока не закончатся, если закончились — вернуть предыдущее значение.
- `sum, product`, если знаем, что элементы можно складывать/умножать.
- `reduce`, чтобы провести произвольную «схлопывающую» операцию.
- … (куча других + свои)

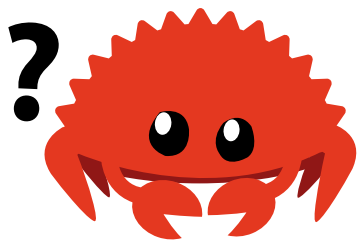
Комбинаторы на итераторах

- `map` — применить `FnMut` к каждому элементу
- `filter` — оставить только те, где `FnMut` дает `true`.
- `flatten` `,(Iter<Iter<T>>) → Iter<T>` — итератор, который проходит по каждому итератору внутри.
- `chain` — «склеить» два итератора в один.
- `zip` — склеить два итератора по двум типам `T` и `U` в один итератор по `(T, U)`
- *и так далее...*

Устройство комбинаторов

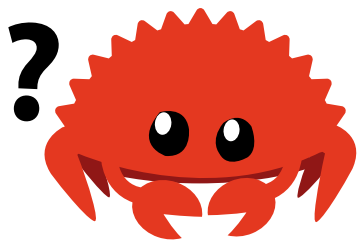
Комбинаторы возвращают *новый итератор*, новую структуру, хранящие в себе старый итератор, но по-новому определяя для него `next()` и другие методы.

```rust
use std::iter;

let vec = vec![5, 3, 1, 4, 5];
vec.iter()
   .filter(|&v| v%2 == 1)
   .enumerate()
   .map(|(i, v)| 0..(v + i))
   .flatten()
   .chain(iter::once(0))
```



Что делает?

```rust
use std::iter;

let vec = vec![5, 3, 1, 4, 5];
vec.iter()
   .filter(|&v| v%2 == 1)
   .enumerate()
   .map(|(i, v)| 0..(v + i))
   .flatten()
   .chain(iter::once(0))
```



Какой тип?

```rust
use std::iter;

let vec = vec![5, 3, 1, 4, 5];
vec.iter()
   .filter(|&v| v%2 == 1)
   .enumerate()
   .map(|(i, v)| 0..(v + i))
   .flatten()
   .chain(iter::once(0))
```

```
std::iter::Chain<Flatten<Map<Enumerate<Filter<std::slice::Iter<'_,
usize>, [closure@src/main.rs:8:13: 8:17]>>, [closure@src/main.rs:
10:10: 10:18]>>, std::iter::Once<usize>>
```

Типы неплохо выводятся, но иногда их надо проставлять, и тут можно запутаться.

Передача функциий по значению вместо замыканий

Часто возникающий паттерн:

```
1 iterator.map(|x| Some(x))
2 iterator.filter(
3   |s| s.is_useful()
4 )
```

Передача функциий по значению вместо замыканий

Часто возникающий паттерн:

```
iterator.map(|x| Some(x))
iterator.filter(
  |s| s.is_useful()
)
```

```
iterator.map(Some)
iterator.filter(
  Student::is_useful
)
```

Можно просто передать функцию/конструктор enum-а

- Меньше текста
- Более ясное намерение

Связь с семантикой владения для итераторов по контейнерам

Естественно следует из семантики владения (можно понять на необитаемом острове)

```
1 let vec: Vec<Student> = vec![ ... ];
2 for v in vector { ... } // By value. v: Student
3 for v in &vector { ... } // By reference. v: &Student
4 for v in &mut vector { ... } // By mutable reference. v: &mut
  Student
```

- Поглотил итератором контейнер — всё, контейнер сплыл, но зато владение элементами перешло к итератору
- Нельзя мутабельно заимствовать одновременно с другим заимствованием

> Правила borrowing выполняются на контейнере за счёт проверки на итераторах (они привязаны лайфтаймами к контейнеру)

Ссылки и преобразование в итераторы

```
1 impl<T> IntoIterator for Vec<T> {}
2 impl<'a, T> IntoIterator for &'a Vec<T> {}
3 impl<'a, T> IntoIterator for &'a mut Vec<T> {}
```

- Потребляет: `Vec::<T>::into_iter()`
- Заимствует
  - `[T]::iter()`
  - `<&Vec<i32>>::into_iter`
  - `<&mut Vec<i32>>::into_iter`

DoubleEndedIterator

```rust
pub trait DoubleEndedIterator: Iterator {
    fn next_back(&mut self) -> Option<Self::Item>;
}
```

- Итератор с двумя курсорами: на начало и на конец

На таком итераторе можно делать операции *с другой стороны* (например, `rfind`).

А ещё его можно просто… *перевернуть*: `rev(self) -> Rev<Self>`.

Использование:

```rust
for i in (1..10).rev()
```

DoubleEndedIterator

```rust
pub trait DoubleEndedIterator: Iterator {
    fn next_back(&mut self) -> Option<Self::Item>;
}
```

- Итератор с двумя курсорами: на начало и на конец
- Вызов `next*()` поглощает элемент с соответствующего конца

На таком итераторе можно делать операции *с другой стороны* (например, `rfind`).

А ещё его можно просто… *перевернуть*: `rev(self) -> Rev<Self>`.

Использование:

```rust
for i in (1..10).rev()
```

DoubleEndedIterator

```rust
pub trait DoubleEndedIterator: Iterator {
    fn next_back(&mut self) -> Option<Self::Item>;
}
```

- Итератор с двумя курсорами: на начало и на конец
- Вызов `next*()` поглощает элемент с соответствующего конца
- Итератор кончается, когда начало и конец *встречаются*

На таком итераторе можно делать операции *с другой стороны* (например, `rfind`).

А ещё его можно просто… *перевернуть*: `rev(self) -> Rev<Self>`.

Использование:

```rust
for i in (1..10).rev()
```

ExactSizeIterator

```rust
pub trait ExactSizeIterator: Iterator {
    // Provided methods
    fn len(&self) -> usize { ... }
    fn is_empty(&self) -> bool { ... }
}
```

- Не обязательно реализовывать методы: так как `.size_hint()` у `Iterator` обязан быть реализован корректно.
- В отличие от `.count()` берет `&self`, а не `self`.
- В общем случае итератор не только может не знать свою длину, но и быть *бесконечным*.

Библиотеки

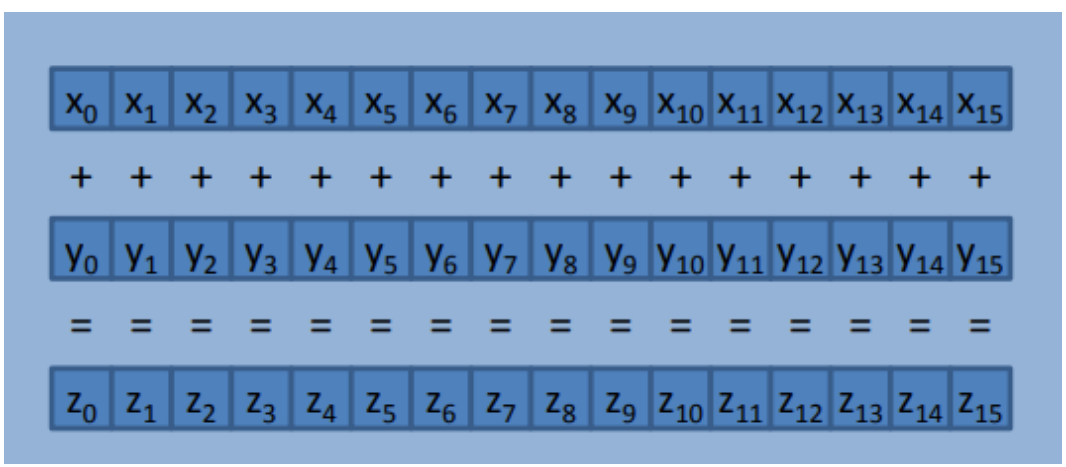
> Читайте доки

Itertools Библиотека с более широким набором итераторов.

rayon Библиотека, позволяющая сделать ваш код на итераторах многопоточным без мыслительных усилий

Что с производительностью

- Вычисления на итераторах «ленивые» ⟶ бесконечные итераторы.
- Компилятор статически знает конкретные вызовы функций ⇒ и может все заинлайнить
- Избегание аллокаций
- Можно применять дополнительные оптимизации вроде *векторизации* («паралелизация на уровне данных»), tiling-а…



Как вернуть итератор?

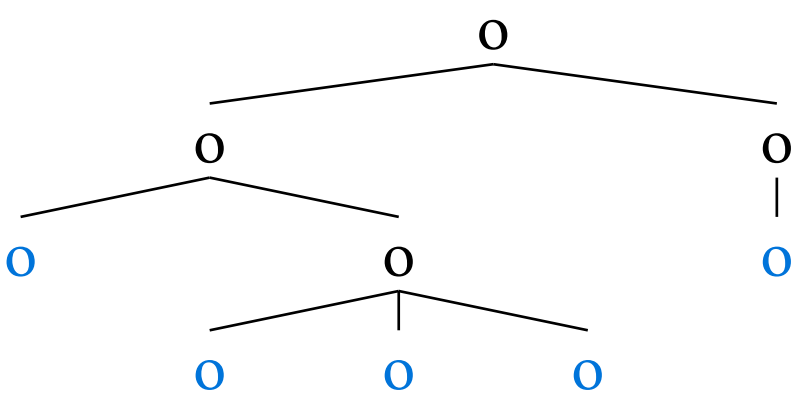
Итератор — `trait`, такой же, как и все.

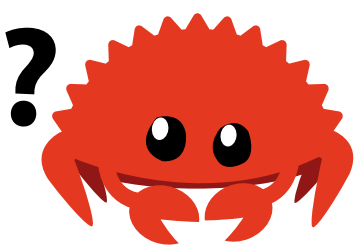
Поэтому если хочется *вернуть* значение, нужно использовать либо `impl`, либо `dyn`.

```
1 fn new_iter() → Box<dyn Iterator> { ... }
2 fn new_iter() → impl Iterator { ... }
```

Рекурсивная итерация

Например, хотим пройтись по всем листьям дерева.





Рекурсивная итерация

Например, хотим пройтись по всем листьям дерева.