

Rust 2019

compscicenter.ru

aleksey.kladov@gmail.com



Лекция 5: Функции

Fn трейты

FnOnce

Так же, как и всё в Rust, оператор `()` определяется трейтом:

```
trait FnOnce<Args> { ❶  
    type Output; ❷  
    fn call_once(self, args: Args) -> Self::Output;  
}
```

❶ тип аргументов (кортеж)

❷ тип результата

FnOnce

```
fn apply<T, R, F: FnOnce(T) -> R>(x: T, f: F) -> R {  
    f(x)  
}
```

`FnOnce(U, V) -> R` это синтаксический сахар для
`FnOnce<(U, V), Output = R>`

map

```
fn map<T, R, F: FnOnce(&T) -> R>(xs: &[T], f: F) -> Vec<R> {  
    let mut res = Vec::with_capacity(xs.len());  
    for x in xs {  
        // value moved here in previous iteration of loop  
        let y = f(x);  
        res.push(y);  
    }  
    res  
}
```



call_once требует **self**, можно позвать функцию только один раз

FnMut, Fn

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

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

И FnMut, и Fn позволяют позвать функцию несколько раз, но FnMut требует &**mut**.

FnOnce

FnMut

Fn

функция может больше

удобнее вызывать

map

```
fn map<T, R, F: FnMut(&T) -> R>(xs: &[T], mut f: F) -> Vec<R> {  
    let mut res = Vec::with_capacity(xs.len());  
    for x in xs {  
        let y = f(x);  
        res.push(y);  
    }  
    res  
}
```


ФУНКЦИИ

```
fn map<T, R, F: FnMut(&T) -> R>(xs: &[T], mut f: F) -> Vec<R> {  
    ...  
}
```

```
fn sqrt(x: &i32) -> f64 {  
    (xs as f64).sqrt()  
}
```

```
fn main() {  
    let xs = vec![1, 2, 3];  
    let ys = map(&xs, sqrt);  
}
```

Размер функции

```
fn size_of_val<T>(_: T) -> usize {  
    std::mem::size_of::<T>()  
}
```

```
let s = size_of_val(next);  
assert_eq!(s, 0)
```

Функции — ZST объекты

`map<F>(f: F, ...)` мономорфизируется в `map_next(...)` и `map_prev(...)`

Везде — статические вызовы

Представление функции в run-time — ничего

Тип Функции



У каждой функции — уникальный, незаписываемый (non denotable) тип

zipmap

```
fn zipmap<F: Fn(&T) -> R, T, R>(xs: &[T], fs: &[F]) -> Vec<R> {  
    let iter = xs.iter().zip(fs);  
    let mut res = Vec::with_capacity(iter.len()); // ^^  
    for (x, f) in iter {  
        res.push(f(x));  
    }  
    res  
}
```

zipmap

```
fn zipmap<F: Fn(&T) -> R, T, R>(xs: &[T], fs: &[F]) -> Vec<R> {  
    let iter = xs.iter().zip(fs);  
    let mut res = Vec::with_capacity(iter.len());  
    for (x, f) in iter {  
        res.push(f(x));  
    }  
    res  
}
```

```
fn next(x: &i32) -> i32 { x + 1 }
```

```
fn prev(x: &i32) -> i32 { x - 1 }
```

```
fn main() {  
    let xs = vec![1, 2];  
    // expected fn item, found a different fn item  
    let fs = vec![next, prev];  
    let ys = zipmap(&xs, &fs);  
}
```

zipmap

```
fn zipmap<F: Fn(&T) -> R, T, R>(xs: &[T], fs: &[F]) -> Vec<R> {  
    let iter = xs.iter().zip(fs);  
    let mut res = Vec::with_capacity(iter.len());  
    for (x, f) in iter {  
        res.push(f(x));  
    }  
    res  
}
```

```
fn next(x: &i32) -> i32 { x + 1 }
```

```
fn prev(x: &i32) -> i32 { x - 1 }
```

```
fn main() {  
    let xs = vec![1, 2];  
    // :-(  
    let fs = vec![next, next];  
    let ys = zipmap(&xs, &fs);  
}
```



Как выглядит $V_{\text{ec}}\langle T \rangle$, если $T \rightarrow ZST$?



Как выглядит **`Ves<T>`**, если **`T`** — ZST?

Примерно как счётчик: `push = increment`, `pop = decrement`

Указатель на Функцию

fn(T, U) -> R

- `mem::size_of::<fn()>() == mem::size_of::<usize>()`
- представление в памяти — адрес
- вызов = call по адресу

zipmap

```
fn zipmap<F: Fn(&T) -> R, T, R>(xs: &[T], fs: &[F]) -> Vec<R> {  
    let iter = xs.iter().zip(fs);  
    let mut res = Vec::with_capacity(iter.len());  
    for (x, f) in iter {  
        res.push(f(x));  
    }  
    res  
}
```

```
fn next(x: &i32) -> i32 { x + 1 }
```

```
fn prev(x: &i32) -> i32 { x - 1 }
```

```
fn main() {  
    let xs = vec![1, 2];  
    // Функциональный тип приводится (coerce) к указателю  
    let fs: Vec<fn(&i32) -> i32> = vec![next, prev];  
    let ys = zipmap(&xs, &fs);  
}
```

Замыкания

Функции и указатели на функции — `Func` тип. Для них всегда выполняется `Fn`, в `self` ничего интересного не лежит.

Замыкание

пара из окружения и функции

Псевдокод:

```
struct Closure {  
    env: Env,  
    f: fn(Env, Foo) -> Bar  
}  
  
impl FnOnce<(Foo,)> for Closure {  
    type Output = Bar;  
    fn call_once(self, (arg, ): (Foo,)) -> Bar {  
        let Closure { env, f } = self;  
        f(env, arg)  
    }  
}
```

Замыкания

```
fn closest_point(xs: &[Point], tgt: Point) -> Option<&Point> {  
    xs.iter().min_by_key(|p| p.dist(tgt))  
}
```

```
|arg1: Type1, arg2: Type2| -> ResultType {  
    body  
}
```

- типы аргументов и результата опциональны
- тело — любое выражение (не обязательно блок)

Captures

```
let xs = vec![1, 2, 3];  
let f = || {  
    // что нибудь делаем с xs  
};
```

Для `f` компилятор генерирует скрытую структуру с полем `xs` и соответствующий **impl** `FnOnce`

Для каждого замыкания — уникальный тип (поле `f` из псевдокода — не указатель на функцию, а ZST)

Captures

```
struct F {  
    xs: Vec<i32>,  
}
```

```
struct F<'a> {  
    xs: &'a Vec<i32>,  
}
```

```
struct F<'a> {  
    xs: &'a mut Vec<i32>,  
}
```

Нужный вариант выбирается исходя из использования `xs` внутри функции

Captures

```
let xs = vec![1, 2, 3];  
let ys = vec![4, 5, 6];  
let mut zs = vec![7, 8, 9];  
let f = || {  
    drop(xs);  
    println!("{}", ys.len());  
    zs.push(10);  
};
```

Captures

```
struct Closure<'a, 'b> {  
    xs: Vec<i32>,  
    ys: &'a Vec<i32>,  
    zs: &'b mut Vec<i32>,  
}  
  
impl<'a, 'b> FnOnce() for Closure<'a, 'b> {  
    type Output = ();  
    fn call_once(mut self) {  
        drop(self.xs);  
        println!("{}", ys.len());  
        zs.push(10);  
    }  
}
```

Captures

```
struct Closure<'a, 'b> {  
    xs: Vec<i32>,  
    ys: &'a Vec<i32>,  
    zs: &'b mut Vec<i32>,  
}
```

```
impl<'a, 'b> FnOnce() for Closure<'a, 'b> {  
    type Output = ();  
    fn call_once(mut self) {  
        drop(self.xs);  
        println!("{}", ys.len());  
        zs.push(10);  
    }  
}
```



Можно ли написать **FnMut** для **Closure<'a, 'b>**?

move

Иногда хочется сделать `move` , даже если хватает ссылки:

```
let has_gc = {  
    let no_gc = vec!["C", "C++", "Rust"];  
    |lang: &str| -> bool {  
        !no_gc.contains(&lang)  
        //    ^^^^^ borrowed value does not live long enough  
    }  
};  
  
assert!(has_gc("Java"));
```

move

Можно использовать ключевое слово **move** :

```
let has_gc = {  
    let no_gc = vec!["C", "C++", "Rust"];  
    move |lang: &str| -> bool {  
        !no_gc.contains(&lang)  
    }  
};  
  
assert!(has_gc("Java"));
```

move

Можно использовать ключевое слово **move** :

```
let has_gc = {  
    let no_gc = vec!["C", "C++", "Rust"];  
    move |lang: &str| -> bool {  
        !no_gc.contains(&lang)  
    }  
};  
  
assert!(has_gc("Java"));
```



Какие из **FnOnce** , **FnMut** , **Fn** трейтов реализованы для **has_gc** ?

Capture Clause

В C++ нужно явно указывать, как захватываются переменные

В Rust компилятор проверит, что автоматический вывод корректен

Capture Clause

move позволяет указать окружение явно:

```
let xs = vec![1, 2, 3];
let ys = vec![4, 5, 6];
let mut zs = vec![7, 8, 9];

let f = {
    let xs = xs;
    let ys = &ys;
    let zs = &mut zs;
    move || {
        drop(xs);
        println!("{}", ys.len());
        zs.push(10);
    }
};
```


Итераторы

Личная история: я осознал прелесть Rust, когда понял, как работают итераторы

Пифагоровы Тройки

```
let triplets = (1u32..)
    .flat_map(|z| (1..=z).map(move |y| (y, z)))
    .flat_map(|(y, z)| (1..=y).map(move |x| (x, y, z)))
    .filter(|(x, y, z)| x*x + y*y == z*z);

let first_ten: Vec<(u32, u32, u32)> =
    triplets.take(10).collect();

// [(3, 4, 5), (6, 8, 10) ... (20, 21, 29)]
println!("{}", first_ten)
```

- `map`: трансформирует элементы последовательности
- `flat_map`: превращает элемент в последовательность
- `filter`, `take`, ...
- `collect` превращает "ленивый" итератор в коллекцию

Пифагоровы Тройки



Для пифагоровых троек есть параметризация!

std::iter::Iterator

```
trait Iterator {  
    type Item;  
    fn next(&mut self) -> Option<Self::Item>;  
  
    ... // методы с реализацией по умолчанию  
}
```

Типы-суммы нужны!

- Java: два метода, `next` и `hasNext`
- Python: исключения для управления потоком управления

Минимальный итератор

```
struct Countdown(u32);

impl Iterator for Counter {
    type Item = u32;
    fn next(&mut self) -> Option<u32> {
        if self.0 == 0 {
            None
        } else {
            self.0 -= 1;
            Some(self.0 + 1);
        }
    }
}

fn main() {
    for x in Countdown(10) {
        println!("{}", x)
    }
}
```

Intolterator

for работает через IntoIterator:

```
pub trait IntoIterator {  
    type Item;  
    type IntoIter: Iterator<Item=Self::Item>;  
    fn into_iter(self) -> Self::IntoIter;  
}
```

```
for x in xs {  
    body  
}
```

```
let mut it = xs.into_iter();  
while let Some(x) = it.next() {  
    body  
}
```

Intolterator

for работает через IntoIterator:

```
pub trait IntoIterator {  
    type Item;  
    type IntoIter: Iterator<Item=Self::Item>;  
    fn into_iter(self) -> Self::IntoIter;  
}
```

```
for x in xs {  
    body  
}  
  
let mut it = xs.into_iter();  
while let Some(x) = it.next() {  
    body  
}
```



Где ошибка в сахараивании?

Intolterator

```
{ 1  
    let mut it = xs.into_iter();  
    while let Some(x) = it.next() {  
        body  
    }  
}
```

1 создаём блок, чтобы вовремя уничтожить `it`

Intolter

```
impl<I: Iterator> IntoIterator for I {  
    type Item = I::Item;  
    type IntoIter = I;  
    fn into_iter(self) -> I { self }  
}
```

Любой итератор также IntoIterator (blanket impl)

Ровно поэтому Countdown работает с **for**

Intolter

```
impl<T> IntoIterator for Vec<T> {  
    type Item = T;  
    type IntoIter = std::vec::IntoIter<T>;  
    fn into_iter(self) -> Self::IntoIter { ... }  
}  
  
fn process(xs: Vec<String>) {  
    for x in xs {  
        println!("{}", x);  
        // освободили память строки  
    }  
    // xs уже нет  
}
```

Коллекции реализуют IntoIterator, передавая владение содержимым

Iter

```
impl<'a, T> IntoIterator for &'a Vec<T> {  
    type Item = &'a T;  
    type IntoIter = std::vec::Iter<'a, T>;  
    fn into_iter(self) -> Self::IntoIter { ... }  
}  
  
fn process(xs: Vec<String>) {  
    for x in &xs {  
        *x = String::new();  
    }  
    for x in xs.iter() {  
        println!("{}", x);  
    }  
}
```

Итерация по ссылке на коллекцию возвращает &T ссылки

IterMut

```
impl<'a, T> IntoIterator for &'a mut Vec<T> {  
    type Item = &'a mut T;  
    type IntoIter = std::vec::IterMut<'a, T>;  
    fn into_iter(self) -> Self::IntoIter { ... }  
}  
  
fn process(mut xs: Vec<String>) {  
    for x in &mut xs {  
        println!("{}", x);  
    }  
    for x in xs.iter_mut() {  
        println!("{}", x);  
    }  
}
```

Аналогично для **&mut**.

map

```
trait Iterator {  
    type Item;  
    fn next(&mut self) -> Option<Self::Item>;  
  
    fn map<R, F>(self, f: F) -> ???  
    where  
        F: FnMut(Self::Item) -> R, // 80% правды  
        {}  
  
    ...  
}
```

Вместо ??? нельзя написать `Iterator<Item = R>`, ведь `Iterator` это не тип

```
pub trait Iterator {  
    fn map<B, F>(self, f: F) -> Map<Self, F>  
        where F: FnMut(Self::Item) -> B  
    {  
        Map { iter: self, f }  
    }  
}
```

```
pub struct Map<I, F> {  
    iter: I,  
    f: F,  
}
```

```

pub struct Map<I, F> {
    iter: I,
    f: F,
}

impl<B, I, F> Iterator for Map<I, F>
where
    I: Iterator,
    F: FnMut(I::Item) -> B,
{
    type Item = B;

    fn next(&mut self) -> Option<B> {
        match self.iter.next() {
            None => None,
            Some(value) => Some((self.f)(value)),
        }
    }
}

```


Map

map возвращает конкретный тип, Map , параметризованный итератором и функцией

Map реализует итератор, если составные части нужной формы

Цепочки итераторов образуют сложный, "телескопический" тип:

```
use std::{  
    iter::{Map, Filter},  
    ops::Range  
};  
  
fn main() {  
    let _it: Filter<Map<Range<i32>, _>, _> =  
        (1..10).map(|i| i * 2).filter(|i| i > 5);  
}
```

Итераторы в Runtime

Параметризация предыдущим итератором (`Map<Self, _>`):

- `size_of` сложного итератора — сумма `size_of` частей
- всё состояние — один конечный автомат на стэке

Параметризация функцией (`Map<_, F>`):

- не нужно аллоцировать объект замыкания в куче
- замыкание состоит **только** из окружения, функция — ZST
- компилятор статически знает, какой код вызывается — итераторы компилируются в быстрые циклы
- тип сложных итераторов нельзя написать

API Итераторов, Трансформации

Трансформации конструируют новые итераторы:

`map, filter, flat_map, flatten, zip, unzip, chain, take, skip, enumerate, ...`

```
/// трансформируем элементы, но трансформация не всегда применима
fn filter_map<B, F: FnMut(Self::Item) -> Option<B>>(self, f: F)
/// смотрим на текущий элемент (для дебага)
fn inspect<F: FnMut(&Self::Item)>(self, f: F)
/// превращаем итератор ссылок в итератор значений
fn cloned<'a, T>(self)
    Self: Iterator<Item = &'a T>,
    T: 'a + Clone,
```

API Итераторов, Терминальные операции

Терминальные операции запускают итератор, вызывая `.next` :
for, `for_each`, `count`, `min`, `max`, `sum`, `product`, `any`, `all`,
`find_map`, `fold`, `last`, ...

`collect` : используя `return type polymorphism`, позволяет
конструировать что-нибудь из итератора

```
fn collect<B: FromIterator<Self::Item>>(self) -> B {  
    FromIterator::from_iter(self)  
}
```

```
trait FromIterator<T> {  
    fn from_iter<I>(iter: I) -> Self  
    where  
        I: IntoIterator<Item = T>;  
}
```

collect

```
let a = [1, 2, 3];
```

```
let doubled: Vec<i32> = a.iter().map(|&x| x * 2).collect();
```

```
let doubled = a.iter().map(|&x| x * 2).collect::<HashSet<i32>>();
```

```
let doubled = a.iter().map(|&x| x * 2).collect::<HashSet<_>>();
```

Можно написать FromIter для своего типа:

```
struct IterLen(usize);
```

```
impl<T> FromIter<T> for IterLen {
```

```
    fn from_iter<I: IntoIterator<Item=i32>>(iter: I) -> IterLen {
```

```
        let mut len = 0;
```

```
        iter.into_iter().for_each(|_| len += 1 );
```

```
        IterLen(1)
```

```
    }
```

```
}
```

Можно собирать итераторы в Result или Option:

```
impl<T, V: FromIterator<T>> FromIterator<Option<T>> for Option<V> {  
    fn from_iter<I: IntoIterator<Item=Option<T>>>(iter: I)  
    -> Option<V> {  
        ...  
    }  
}
```

```
fn main() {  
    let items = vec![2_u16, 1, 0];  
  
    let res: Option<Vec<u16>> = items  
        .iter()  
        .map(|x| x.checked_sub(1))  
        .collect();  
  
    assert_eq!(res, None);  
}
```

Итераторы: Итоги

1. итераторные адаптеры *делают* ничего, только описывают трансформацию
2. терминальные методы запускают итератор
3. $1 + 2 =$ итераторы ленивые и бесплатные: нет аллокаций, нет косвенности
4. записать тип итератора проблематично...
5. `collect` полиморфный метод, позволяет расширять возможности итераторов снаружи
6. для коллекций, бывают итераторы по ссылкам и по значениям

ExactSizeIterator

```
trait Iterator {  
    ...  
    fn size_hint(&self) -> (usize, Option<usize>) { (0, None) }  
}  
  
trait ExactSizeIterator: Iterator {  
    fn len(&self) -> usize {  
        let (lower, upper) = self.size_hint();  
        assert_eq!(upper, Some(lower));  
        lower  
    }  
}
```

ExactSizeIterator — маркер-трейт для итераторов, которые точно знают свой размер.

ExactSizeIterator

```
impl<A, B> ExactSizeIterator for Zip<A, B>
  where A: ExactSizeIterator, B: ExactSizeIterator {}

fn zipmap<F: Fn(&T) -> R, T, R>(xs: &[T], fs: &[F]) -> Vec<R> {
  let iter = xs.iter().zip(fs);
  let mut res = Vec::with_capacity(iter.len());
  for (x, f) in iter {
    res.push(f(x));
  }
  res
}
```

- МОЖЕМ ПОЗВАТЬ `.len`, так как `.zip` сохраняет `ExactSizeIterator`

DoubleEndedIterator

```
trait DoubleEndedIterator: Iterator {  
    fn next_back(&mut self) -> Option<Self::Item>;  
    ...  
}  
  
trait Iterator {  
    fn rev(self) -> Rev<Self> where Self: DoubleEndedIterator {  
        Rev { iter: self }  
    }  
    ...  
}
```

- `DoubleEndedIterator` позволяет итерироваться с обеих КОНЦОВ
- нет `RandomAccessIterator`: в общем случае, нельзя проитерироваться дважды

Домашнее задание: восхитится организацией итераторов

<https://doc.rust-lang.org/std/iter/index.html>

Инвалидация Итераторов

В Rust не бывает инвалидации итераторов:

```
fn process(mut xs: Vec<i32>) {  
    for x in &xs { ❶  
        xs.pop(); ❷  
    }  
}
```

❶ & ссылка

❷ &mut ссылка

Проверка Выхода за Границу

`Index` и `IndexMut` для `[T]` всегда выполняют проверку выхода за границу.

Так как итераторы гарантированно не инвалидируются, то их можно реализовать без проверок

Итератор по `[T]` — пара указателей, так же, как и в C++

В отличие от `[T]`, случайное изменение контейнера во время итерации не ведёт к UB

Стоимость Проверок

Проверка индекса — тривиально предсказуемая ветка, непосредственная цена не значительна.

Но проверки могут сломать оптимизации, например, автовекторизацию

Невекторизуемый Код

```
#[inline(never)]
fn sum_indirectly(xs: &[u32], indexes: &[usize]) -> u32
{
    let mut sum = 0u32;
    for &idx in indexes {
        let x = xs[idx];
        sum = sum.wrapping_add(x);
    }
    sum
}
```

Время работы: 63.633349ms

`indexes` — массив `[0, 1, ... xs.len())`, но компилятор об этом не знает

Невекторизуемый Код

```
#[inline(never)]
fn sum_indirectly_unchecked(xs: &[u32], indexes: &[usize]) -> u32
{
    let mut sum = 0u32;
    for &idx in indexes {
        let x = unsafe { *xs.get_unchecked(idx) };
        sum = sum.wrapping_add(x);
    }
    sum
}
```

Время работы: 64.861459ms

Векторизуемый код

```
#[inline(never)]
fn sum(xs: &[u32], lo: usize, hi: usize) -> u32 {
    let mut sum = 0u32;
    for idx in lo..hi {
        let x = xs[idx];
        sum = sum.wrapping_add(x);
    }
    sum
}
```

Время работы: 34.513286ms

`lo = 0, hi = xs.len()`, но компилятор про это не знает

Векторизуемый код

```
#[inline(never)]
fn sum_unchecked(xs: &[u32], lo: usize, hi: usize) -> u32 {
    let mut sum = 0u32;
    for idx in lo..hi {
        let x = unsafe { *xs.get_unchecked(idx) };
        sum = sum.wrapping_add(x);
    }
    sum
}
```

Время работы: 17.653021ms

Векторизуемы код

Не нужно писать **unsafe** для достижения производительности:

```
#[inline(never)]
fn sum(xs: &[u32], lo: usize, hi: usize) -> u32 {
    let mut sum = 0u32;
    for &x in &xs[lo..hi] {
        sum = sum.wrapping_add(x);
    }
    sum
}
```