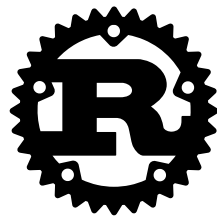


Rust 2019

compscicenter.ru

aleksey.kladov@gmail.com



Лекция 3: Трейты

Сопоставление с Образцом

```
enum E {  
    One(usize),  
    Two { value: Vec<u32>, other: Vec<u32> },  
    Tree,  
}
```

```
fn foo(e: E) {  
    match e {  
        E::One(x) => x,  
        E::Two { value: xs, .. } => xs.len(),  
        _ => 92,  
    }  
}
```

- нужно явно указывать все ветки
- `_` матчит всё, что угодно
- `..` игнорирует остальные поля

```
enum E {  
    One(usize),  
    Two { value: Vec<u32>, other: Vec<u32> },  
    Tree,  
}
```

```
fn foo(e: E) {  
    let xs: Vec<u32> = match e {  
        E::Two { value, other: _ } => value,  
        _ => return,  
    }  
}
```

- все ветки должны иметь один тип (но помним про !)
- **match** может поглощать аргумент

Range Patterns

```
let x = 92u8;  
match x {  
    0    ..= 128 => "small",  
    129 ..= 255 => "big",  
}
```

- `..=` матчит отрезок (включая концы)

Binding Modes

```
fn foo(r: Result<Vec<u32>, Error>) {  
    let xs: &Vec<u32> = match &r {  
        Ok(xs) => xs,  
        Err(_) => return,  
    };  
}
```

- если матчить `&T` а не `T`, то результат — `&` ссылки
- исходное значение не поглощается
- аналогично для `&mut T`

Caxap

```
if let Ok(res) = f() {  
    body  
}
```

```
match f() {  
    Ok(res) => body,  
    _ => (),  
}
```

```
while let Some(v) = it.next() {  
    body  
}
```

```
loop {  
    match it.next() {  
        Some(v) => body,  
        _ => break,  
    }  
}
```


Всё — Паттерны

let и аргументы функций — тоже паттерны:

```
struct Color { r: u8, g: u8, b: u8 }
```

```
fn red(Color { r, .. }: Color) -> u8 {  
    r  
}
```

Параметрический Полиморфизм

Quiz

Что является результатом компиляции функции?

```
fn scale(p: &mut Point, factor: f64) {  
    p.x *= factor;  
    p.y *= factor;  
}
```

Quiz

Что является результатом компиляции структуры?

```
struct Point(f64, f64);
```

Параметризованные функции

- в Rust нет перегрузки функций
- но функции могут быть параметризованы типами

```
fn identity<T>(x: T) -> T {  
    x  
}
```

```
fn drop<T>(_x: T) {  
}
```

Traits

Чтобы сделать что-то полезное с аргументом типа T , надо знать интерфейс T !

```
struct Cat;

trait Say {
    fn say(&self);
}

impl Say for Cat {
    fn say(&self) { println!("meow!") }
}

fn main() {
    let cat = Cat;
    cat.say();
}
```



Реализация трейта (**impl**) живёт отдельно от определения структуры

```
struct Cat;
impl Say for Cat {
    fn say(&self) { println!("meow!") }
}
```

```
struct Dog;
impl Say for Dog {
    fn say(&self) { println!("woof!") }
}
```

```
fn say_twice<T: Say>(t: &T) {
    t.say();
    t.say();
}
```

```
fn main() {
    let dog = Dog;
    say_twice(&dog);
}
```


Трейты слегка похожи на интерфейсы из ОО языков, но сходство скорее обманчиво

impl MyTrait for YourType

Можно реализовывать свои трейты для чужих типов:

```
impl Say for i32 {  
    fn say(&self) {  
        println!("hm... int-int?")  
    }  
}  
  
fn main() {  
    let x = 92;  
    92.say();  
}
```

Это работает, потому что Rust не использует таблицы виртуальных функций

Return type polymorphism

Невозможный интерфейс

```
trait Default {  
    fn default() -> Self;  
}
```

- `Self` — тип, для которого реализуется трейт
- у функции `default` нет параметров, но тип результата не фиксирован

```
trait Default {  
    fn default() -> Self;  
}
```

```
struct Circle {  
    center: Point,  
    radius: f64,  
}
```

```
impl Default for Circle {  
    fn default() -> Circle {  
        Circle {  
            center: Point::default(),  
            radius: 1.0,  
        }  
    }  
}
```

Static Dispatch

```
fn make_default<T: Default>() -> T {  
    T::default()  
}
```

```
fn main() {  
    let c: Circle = make_default();  
}
```

- не можем сделать виртуальный вызов — нет объекта
- можем статически понять тип и вызвать нужный код

Мономорфизация

- в исполняемом файле `make_default::<Circle>` и `make_default::<Square>` это две разные функции (два разных фрагмента машинного кода)
- при компиляции известны конкретные значения всех параметров типа, можно подставить вызов нужной функции

```
fn main() {  
    let circle: Circle = make_default::<Circle>();  
    let square: Square = make_default::<Square>();  
}
```

Мономорфизация

```
fn default_pair<T: Default>() -> (T, T) {  
    (make_default(), make_default())  
}  
  
fn main() {  
    let _: (Circle, Circle) = default_pair();  
}
```

В процессе компиляции генерируются только нужные варианты функций

Дженерики в Java

- у всех объектов одинаковое представление — указатель
- в начале каждого объекта лежит указатель на таблицу виртуальных функций (энергичное размахивание руками)
- существует только один вариант каждой функции: стирание типов в runtime
- разное поведение достигается за счёт виртуальных вызовов


```

trait Area {
    fn area(&self) -> f64;
}

struct Circle { center: Point, radius: f64 }
impl Area for Circle { ... }

struct Rectangle { bottom_left: Point, top_right: Point }
impl Area for Rectangle { ... }

fn main() {
    let mut shapes: Vec<Area> = Vec::new();
    shapes.push(Circle::default());
    shapes.push(Rectangle::default());
    for shape in shapes.iter() {
        println!("area: {}", shape.area())
    }
}

```

Мономорфизация

- нельзя складывать в вектор объекты разных типов — у них разный `mem::size_of`



trait это не тип, это свойство типа.

Inlining

inlining

оптимизация, подстановка тела функции в место её вызова

- экономим вызов функции, но это не важно
- оптимизируем тело функции в контексте места вызова

```
fn multiply(x: i32, y: i32) -> { x * y }
```

```
fn foo(x: i32) {  
    let x = multiply(x, 2);  
}
```



Статические вызовы (и мономорфизация) допускают inline

Мономорфизация vs. Стирание Типа

Статический полиморфизм

- скорость (you couldn't hand code any better):
 - inline
 - оптимальные структуры данных (`Vec<i32>` vs `Vec<i64>`)
- return type polymorphism
- code bloat: много копий каждой функции

Динамический полиморфизм

- раздельная и быстрая компиляция
- интерфейсы являются типами (можно сложить в коллекцию)
- можно добавлять новые типы в run time

Трейты в Rust vs. Шаблоны в C++

- компилируются одинаково (мономорфизация)
- проверка типов в момент определения vs в момент инстанцирования
- подход C++ более могущественный, подход Rust гарантирует больше свойств

Проверка типов

```
fn add<T>(x: T, y: T) {  
    x + y; // binary operation `+` cannot be applied to type `T`  
}
```

Проверка типов

```
fn add<T: std::ops::Add>(x: T, y: T) {  
    x + y;    // ok  
}
```

Проверка типов

```
template <typename T>  
void add(T x, T y) {  
    x + y;  
}
```

```
struct S {};
```

```
int main() {  
    add(1, 2);           // ok  
    add(S {}, S {});    // error: no match for 'operator+'  
}
```


Swift Generics

В Swift используется интересная комбинированная стратегия, сочетающая отдельную компиляцию и эффективные оптимизации:

[Implementing Swift Generics](#)

Разное

Fully Qualified Syntax

```
struct S { ... }  
impl S {  
    fn foo(&self) { ... }  
}
```

```
fn bar(s: &S) {  
    s.foo();  
    S::foo(s)  
}
```

Fully Qualified Syntax

```
struct S { ... }  
trait T {  
    fn foo(&self);  
}  
impl T for S { ... }
```

```
fn bar(s: &S) {  
    s.foo();  
    S::foo(s)  
    T::foo(s);  
    <S as T>::foo(s);  
}
```

- для вызова метода трейт должен быть импортирован
- при коллизии inherent метода и трейта побеждает метод
- при коллизии двух трейтов — ошибка компиляции

Where clauses

```
fn make_default<T: Default>() -> T {  
    Default::default()  
}
```

МОЖНО ЗАПИСАТЬ КАК

```
fn make_default<T>() -> T  
where  
    T: Default,  
{  
    Default::default()  
}
```

- удобно, когда ограничений и параметров много
- можно указать больше ограничений

"Наследование" трейтов

```
trait A {  
    fn a(&self);  
}
```

```
trait B: A {  
    fn b(&self);  
}
```

Любой тип, реализующий B , должен реализовывать A .

"Наследование" трейтов

"Сахар" для ограничения на Self тип

```
trait B  
where  
    Self: A,  
{  
    fn b(&self);  
}
```

"Наследование" трейтов

```
trait A {  
    fn a(&self);  
}  
trait B: A {  
    fn b(&self);  
}
```

```
impl B for Spam {  
    fn b(&self) {}  
}
```

```
impl A for Spam {  
    fn a(&self) {  
        self.b(); // вызываем метод B!  
    }  
}
```


Полезные Трейты

Default

```
trait Default {  
    fn default() -> Self;  
}
```

- инициализирует объект по умолчанию
- название микро-трейтов — глагол
- МОЖНО ВЫВЕСТИ АВТОМАТИЧЕСКИ:

```
#[derive(Default)]  
struct Point(f64, f64);
```

- `default_constructible` из C++

Clone

```
trait Clone {  
    fn clone(&self) -> Self;  
    // Опциональная оптимизация  
    fn clone_from(&mut self, source: &Self) {  
        *self = source.clone();  
    }  
}
```

- создаёт копию объекта
- аналог copy ctor / copy assignment из C++
- МОЖНО ВЫВЕСТИ (derive)

Copy

```
#[lang = "copy"]  
trait Copy: Clone { }
```

- пустой "marker" trait
- lang item: компилятор знает про Copy
- derive

Derive

```
#[derive(Clone, Copy)]  
struct Wrapper<T> {  
    ptr: *const T,  
}
```

Derive

```
#[derive(Clone, Copy)]
struct Wrapper<T> {
    ptr: *const T,
}

// #[derive(Clone)]
impl<T: Clone> Clone for Wrapper<T> {
    fn clone(&self) -> Self {
        ...
    }
}
```

"условный" impl

Derive

```
#[derive(Clone, Copy)]  
struct Wrapper<T> {  
    ptr: *const T,  
}
```



Wrapper<Vec<i32>> не будет : Copy,
ХОТЯ * const Vec<i32>: Copy

Derive

```
struct Wrapper<T> {  
    ptr: *const T,  
}  
  
impl<T> Copy for Wrapper<T> {  
}  
  
impl<T> Clone for Wrapper<T> {  
    fn clone(&self) -> Wrapper<T> {  
        *self // делегируем к `Copy`  
    }  
}
```


PartialEq, Eq

```
pub trait PartialEq<Rhs = Self> {  
    fn eq(&self, other: &Rhs) -> bool;  
    fn ne(&self, other: &Rhs) -> bool { !self.eq(other) }  
}
```

```
pub trait Eq: PartialEq<Self> { }
```

- перегрузка операторов == и !=
- "partial": f64: PartialEq, но f64::NAN != f64::NAN
- нельзя сравнивать яблоки и апельсины
- **impl** PartialEq<Ipv4Addr> **for** IpAddr

PartialOrd, Ord

```
pub trait PartialOrd<Rhs = Self>: PartialEq<Rhs> {  
    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 { ... }  
}
```

- "partial" из-за std::f64::NAN

```
let mut xs: Vec<f32> = ...;  
xs.sort(); // the trait Ord is not implemented for f32  
xs.sort_by(|a, b| a.partial_cmp(b).unwrap())
```

- нет необходимости в новом операторе (<=>), как в C++

PartialOrd, Ord

```
pub trait Ord: Eq + PartialOrd<Self> {  
    fn cmp(&self, other: &Self) -> Ordering;  
  
    // Note: self  
    fn max(self, other: Self) -> Self { ... }  
    fn min(self, other: Self) -> Self { ... }  
}  
  
impl Ord for Foo {  
    fn cmp(&self, other: &Foo) -> Ordering {  
        ...  
    }  
}  
  
impl PartialOrd for Foo {  
    fn partial_cmp(&self, other: &Foo) -> Option<Ordering> {  
        Some(self.cmp(other))  
    }  
}
```

std::hash::Hash

```
trait Hash {  
    fn hash<H: Hasher>(&self, state: &mut H)  
    fn hash_slice<H: Hasher>(data: &[Self], state: &mut H) { ... }  
}
```

```
trait Hasher {  
    fn finish(&self) -> u64;  
    fn write(&mut self, bytes: &[u8]);  
  
    fn write_u8(&mut self, i: u8) { ... }  
    fn write_u16(&mut self, i: u16) { ... }  
    ...  
}
```

- методы трейта могут быть параметризованными
- пользователь выбирает хэш функцию

Drop

```
trait Drop {  
    fn drop(&mut self);  
}
```

- вызывается при уничтожении значения
- компилятор не знает ничего про `Vec<i32>`, кроме наличия `Drop`
- деструктор / **Resource Acquisition Is Initialization**

Drop

- нельзя позвать руками

```
fn drop_twice(mut xs: Vec<i32>) {  
    xs.drop();  
    xs.drop(); // explicit destructor calls not allowed  
}
```

- есть `std::mem::drop`: **fn** drop<T>(_x: T)

```
struct App {  
    db: Database,  
}  
  
impl Database {  
    fn shutdown(self) { ... }  
}  
  
impl Drop for App {  
    fn drop(&mut self) {  
        let db = self.db; // cannot move out of here  
        db.shutdown()  
    }  
}
```

```

struct App {
    db: Option<Database>,
}

impl Database {
    fn shutdown(self) { ... }
}

impl Drop for App {
    fn drop(&mut self) {
        let db = self.db.take().unwrap();
        assert!(self.db.is_none());
        db.shutdown()
    }
}

```

- **fn** take(&mut **self**) -> Option<T> — полезная функция у Option

std::mem Tricks

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

меняет два значения местами

```
fn replace<T>(dest: &mut T, src: T) -> T
```

получить T из &mut T

Обобщим Option::take:

```
fn take<T: Default>(x: &mut T) -> T {  
    std::mem::replace(x, T::Default())  
}
```

Quiz



Как написать `replace`?

```
fn replace<T>(dest: &mut T, src: T) -> T {  
  
}
```

Quiz



Как написать `replace`?

```
fn replace<T>(dest: &mut T, mut src: T) -> T {  
    mem::swap(dest, &mut src);  
    src  
}
```

std::mem Tricks

```
fn forget(x: T)
```

поглощает значение без вызова Drop

Эмуляция Линейных Типов

Drop не возвращает значение и не принимает аргументов

Классический пример — буферизованный вывод. При закрытии потока нужно записать все данные, но нужно обработать ошибки.



Как заставить пользователя не забыть вызвать `flush` ?

Эмуляция Линейных Типов

```
struct SafeBufWrite { ... }

impl SafeBufWrite {
    fn flush(&mut self) -> Result { ... }
    fn close(mut self) -> Result {
        self.flush()?;
        mem::forget(self) // обезвредили drop
    }
}

impl Drop for SafeBufWrite {
    fn drop(&mut self) {
        let _ = self.flush(); // игнорируем ошибки
        panic!("should be flushed explicitly")
    }
}
```

Подробнее — в лекции про обработку ошибок

Порядок Drop

- локальные переменные уничтожаются в обратном порядке (LIFO):

```
{  
    let xs = Vec::new();  
    let ys = Vec::new();  
    // drop(ys);  
    // drop(xs);  
}
```

- можно руками позвать drop :

```
{  
    let xs = Vec::new();  
    let ys = Vec::new();  
    drop(xs);  
    // drop(ys);  
}
```

Порядок Drop



Поля уничтожаются в порядке объявления

```
#[derive(Default)]  
struct S {  
    xs: Vec<i32>,  
    ys: Vec<i32>,  
}
```

```
fn main() {  
    let s = S::default();  
    // drop(s.xs);  
    // drop(s.ys);  
}
```

- есть `ManuallyDrop` для тонкого контроля, но придётся использовать **`unsafe`**.

dropck

Drop влияет на анализ ВЖ

```
#[derive(Default)]
struct Wrapper<'a> {
    r: Option<&'a i32>
}

impl<'a> Wrapper<'a> {
    fn push(&mut self, x: &'a i32) { self.r = Some(x); }
}

fn main() {
    let mut xs = Wrapper::default();
    let x = 92;
    xs.push(&x);
    // drop(x);
    // drop(xs);
}
```

В последней строке `r: Option<&i32>` висит, но это не страшно

```

#[derive(Default)]
struct Wrapper<'a> {
    r: Option<&'a i32>
}

impl<'a> Wrapper<'a> {
    fn push(&mut self, x: &'a i32) { self.r = Some(x); }
}

impl<'a> Drop for Wrapper<'a> {
    fn drop(&mut self) {
        println!("UAF? {:?}", self.r);
    }
}

fn main() {
    let mut xs = Wrapper::default();
    let x = 92;
    xs.push(&x);
} // borrow might be used here, when xs is dropped and
   // runs the Drop code for type Wrapper

```

Deref

```
trait Deref {  
    type Target;  
  
    fn deref(&self) -> &Self::Target;  
}
```

- **type** Target — ассоциированный тип
- перегрузка операторов * и .
- &P неявно приводится к &T если P: Deref<Target = T>

Примеры

```
impl<T> Deref for Vec<T> {  
    type Target = [T];  
    fn deref(&self) -> &[T] {  
        ...  
    }  
}
```

```
fn foo(xs: &Vec<i32>) {  
    let _n = xs.len(); // len -- метод на [T]  
    bar(xs); // неявное приведение (coercion)  
}
```

```
fn bar(xs: &[i32]) {  
}
```



&Vec<T> "бесполезный" тип, предпочитайте &[T]

Примеры

```
impl<T> Deref for Box<T> {  
    type Target = T;  
    fn deref(&self) -> &T {  
        ...  
    }  
}
```

```
fn foo(b: Box<i32>) {  
    let p: &i32 = &*b;    // &*Deref::deref(&b)  
}
```



`&Box<T>` двойная косвенность, лучше `&T`

DerefMut

```
trait DerefMut: Deref {  
    fn deref_mut(&mut self) -> &mut Self::Target;  
}
```

- то же самое, но для &**mut** P -> &**mut** T

std::ops::Index / IndexMut

```
pub trait Index<Idx> {  
    type Output;  
    fn index(&self, index: Idx) -> &Self::Output;  
}
```

```
pub trait IndexMut<Idx>: Index<Idx> {  
    fn index_mut(&mut self, index: Idx) -> &mut Self::Output;  
}
```

- перегрузка []
- `xs[idx]` это сахар для `*xs.index(idx)`


```

impl<T> ops::Index<usize> for [T] {
    type Output = T;
    fn index(&self, index: usize) -> &T {
        ...
    }
}

```

```

impl<T> ops::Index<ops::Range<usize>> for [T] {
    type Output = [T];
    fn index(&self, index: usize) -> &[T] {
        ...
    }
}

```

```

fn main() {
    let xs = [1, 2, 3, 4, 5];
    let ys: &[i32] = &xs[2..4];
}

```

В реальности чуть сложнее!

Мораль

- трейты — главный customization point языка
- все аспекты поведения типа описываются трейтами
- нет разницы между примитивными, встроенными и пользовательскими типами