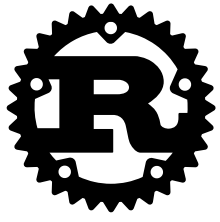
Rust 2019

compscicenter.ru

aleksey.kladov@gmail.com



Лекция 4
Крейты и Модули

Вывод типов

вывод типов смотрит на использование переменных!

```
let mut xs = Vec::new(); 1
xs.push(92i32);          2
```

1. понимаем, что тип `xs` это `Vec<?>`
2. понимаем, что `?` это `i32`

Вывод типов

Почти Хиндли — Милнер, но нужно знать тип ресивера при вызове метода:

```
trait Foo {
    fn foo(&self);
}

trait Bar {
    fn foo(&self);
}

fn main() {
    let x = Default::default();
    x.foo() (1)
}
```

(1) нельзя понять, про какой `foo` идёт речь, не узнав тип `x`

Модули

Модули

crate

единица компиляции Rust ("программа")

```
struct A;

mod bar {
    struct B;

    mod baz { }
}
```

Крейт это дерево модулей:

- безымянный корневой
- `crate::bar`
- `crate::bar::baz`

Области видимости

Каждый модуль изолирован, порядок объявления модулей значения не имеет

```
struct Foo

mod nested {
    type A = Foo; // Foo снаружи не виден
    type B = crate::Foo;
    type B = super::Foo;
}
```

- `crate::` — путь до корневого модуля
- `super::` — путь до родительского модуля

use

Можно использовать конструкцию **use** для импорта имён

```
mod shapes {
    pub struct Circle { ... }
    pub struct Square { ... }
}

mod algorithms {
    use crate::shapes::{Circle, Square};

    fn overlaps(circle: &Circle, square: &Square) -> bool {
        ...
    }
}
```

use

Формы use

```
use some::path::Foo;
use some::path::Fo as Bar;  // переименование при импорте
use some::{
    nested1::name1,
    nested2::{name2, name3};
}
use some::path::*;  // импорт всех символов
```

use

```
enum E {
    X, Y
}

use E::*;

fn main() {
    let e: E = X;
}
```

- можно импортировать варианты из **`enum`**
- порядок деклараций значения не имеет

Модули и Файлы

- содержимое модуля можно вынести в отдельный файл

./main.rs

```
struct S;
mod foo {
    struct Bar;
}
```

Модули и Файлы

- содержимое модуля можно вынести в отдельный файл

./main.rs

```
struct S;
mod foo;
```

./foo.rs

```
struct Bar;
```

- компилятор начинает работу с корневого файла (crate root) и рекурсивно парсит модули.

любой крейт можно сжать в один файл

```rust
mod one {
    mod nested {
        mod nested2 {
            struct Foo;
        }
    }
}
mod two {
    struct Bar;
}
```

```
./
  main.rs          # mod one; mod two;
  one.rs           # mod nested;
  one/
    nested.rs      # mod nested2
    nested/
      nested2.rs   # struct Foo
  two.rs           # struct Bar;
```

Compilation Unit

- крейт — единица компиляции
- один запуск компилятора `rustc lib.rs` компилирует все модули, достижимые из `lib.rs`
- между модулями могут быть произвольные циклические зависимости через **`use`**.

Зависимости

./foo.rs

```
pub fn hello() {
    println!("Hello, World!");
}
```

./bar.rs

```
fn main() {
    foo::hello();
}
```

```
$ rustc foo.rs --create-type rlib
$ rustc bar.rs --extern foo=./libfoo.rlib
$ ./bar
Hello, World!
```

Зависимости

- по умолчанию, `rustc` компилирует исполняемый файл
- `--crate-type rlib` делает библиотеку
- `--extern name=path` позволяет использовать библиотеку

У библиотеки нет собственного имени, оно определяется флагом `--extern`

Анонимность Крейта

./bar.rs

```
fn main() {
    spam::hello();
}
```

```
$ rustc bar.rs --extern spam=./libfoo.rlib
$ ./bar
Hello, World!
```

std

По-умолчанию, есть зависимость на крейт `std` — стандартную библиотеку.

В начале каждого модуля неявно добавляется:

```
use std::prelude::*;
```

Содержимое `prelude`:

```
std::ops::Drop, std::mem::drop
std::boxed::Box, std::vec::Vec
std::clone::Clone, std::marker::Copy

std::option::Option::{self, *}, std::result::Result::{self, *}

std::cmp::{PartialEq, Eq, PartialOrd, Ord}

std::default::Default
...
```

Модификаторы Доступа

По-умолчанию, имена приватные, и видны только внутри текущего модуля и его детей:

```
mod a {
    struct Foo;
    mod b {
        use super::Foo;  // ok
    }
}

mod b {
    use super::a::Foo;   // error: struct Foo is private
}
```

Модификаторы Доступа

pub(super) делает имя доступным в родительском модуле:

```
mod a {
    pub(super) struct Foo;
    mod b {
        use super::Foo;  // ok
    }
}

mod b {
    use super::a::Foo;   // ok
}
```

Модификаторы Доступа

`pub(crate)` делает имя доступным во всём крейте:

```
mod a {
    pub(super) mod b {
        pub(crate) struct Foo;
    }
}

mod b {
    use super::a::b::Foo;
}
```

`pub(super) mod b` делает сам модуль доступным

Реэкспорт

Модификаторы доступа действуют на **`use`**. Это можно использовать для реэкспорта:

```
mod a {
    pub(crate) use b::Foo;
    mod b {
        pub(crate) struct Foo;
    }
}

mod b {
    use super::a::Foo;
}
```

pub

pub делает имя доступным в других крейтах:

./foo.rs

```
pub fn hello() {}
```

./main.rs

```
fn main() {
    foo::hello();
}
```

Паттерн façade

```
mod foo;
mod bar;

pub use crate::{
    foo::{Foo, Spam},
    bar::Bar,
};
```

- для пользователя удобно плоское API
- реализацию удобно организовывать иерархически

Модификаторы Доступа

`pub` работает на на полях и inherent-методах:

```
enum Foo {
    X(pub u32),
    Y { pub(crate) f: f32 },
}

impl Foo {
    pub(super) fn new(x: u32) -> Foo { Foo::X(x) }
}
```

`pub` **не** работает на:

- методах трейтов
- вариантах энумов

Модификаторы Доступа

- единицы инкапсуляции — модуль и крейт (**очень** удобно)
- большая разница между `pub` и всем остальным (публичное/приватное API)
- приватность по умолчанию — разумный выбор
- `pub(crate)` — хороший выбор для "чуть большей видимости"
- `pub(super)`, `pub(in some::path)` — как правило не нужно:
  - `pub(crate)`
  - вынести код в отдельный крейт

Крейты

- нет глобального пространства имён крейтов
- у крейта нет собственного имени
- крейт может быть известен под разными именами в разных зависимостях
- зависимости нужно указывать явно (нет `PYTHONPATH`)
- крейты образуют ациклический направленный граф

Можно использовать несколько версий крейта одновременно

Содержимое Крейта

- скомпилированный код простых функций
- "исходный" типизированный код параметрических функций
- "исходный" код функций, помеченных как `#[inline]`
- "исходный" код макросов

Cargo

Структура Cargo-пакета

```
Cargo.toml
src/
  main.rs  # главный исполняемый крейт
  lib.rs   # крейт-библиотека
tests/     # крейты -- интеграционные тесты
  one.rs
  two.rs
```

Cargo-пакет содержит один крейт-библиотеку и набор вспомогательных крейтов

Cargo компилирует граф зависимостей в порядке топологической сортировки и передаёт флаг `--extern` компилятору

./Cargo.toml

```
[package]
name = "my_package"
version = "0.1.0"
authors = []
edition = "2018"

[dependencies]
rand = "0.6.5"
```

./src/lib.rs

```
pub fn random_int() -> i32 {
    rand::random::<i32>()  # используем библиотеку rand
}
```

./src/main.rs

```
fn main() {
    let x = my_package::random_int();  # используем lib.rs
    println!("{}", x)
}
```

Crates.io

- https://crates.io/ архив библиотек
- semver
  - для `foo = "1.0.0"` Cargo выбирает максимальную совместимую версию (`1.3.5`)
  - изменение мажорной версии — сигнал об API несовместимости
  - среди зависимостей может быть несколько **разных** мажорных версий одного крейта
- сборка из исходников
- отсутствие конфликтов имён + semver + Cargo.lock = state of the art в управлении зависимостями?

Coherence

Проблема

```rust
trait Say {
    fn say(&self);
}

struct S;

impl Say for S {
    fn say(&self) { println!("A") }
}

impl Say for S {
    fn say(&self) { println!("B") }
}
```

Компилятор должен проверить, что для каждой пары `(Trait, Type)` есть только один impl блок

Проблема

crate foo:

```
trait Say { fn say(&self); }
struct S;
```

crate x:

```
impl foo::Say for foo::S {
    fn say(&self) { println!("A") }
}
```

crate y:

```
impl foo::Say for foo::S {
    fn say(&self) { println!("B") }
}
```

крейты x, y "работают" по отдельности, но не вместе

Coherence

Компилятор должен доказать, что в любой программе есть не более одного impla

Проверять все крейты программы не удобно:

- нет раздельной компиляции
- медленно (глобальный поиск)
- **добавление крейта X к компиляции может сломать независимый крейт Y**

Простое правило

Если `Trait` живёт в крейте `X`, а `Type` в крейте `Y`,

то `impl Trait for Type` может жить только в `X` или `Y`

Так как `X` и `Y` зависят друг от друга (в каком-то направлении), то можно проверить отсутствие конфликта

Локальное (для каждого крейта) правило гарантирует отсутствие конфликтов между **любыми** двумя крейтами

Сложное правило

```
struct Person { ... }
struct PersonId(u32);

impl Index<PersonId> for Vec<Person> {
    fn index(&self, idx: PersonId) -> &Person {
        let idx = idx.0 as usize;
        &self[idx]
    }
}
```

- `Index` — трейт из std
- `Vec` — тип из std
- мы можем написать impl, потому что `PersonId` в `Index<PersonId>` — локальный тип
- подробности в RFC-2451: re-rebalancing coherence

FFI

ABI

ABI(calling convention)/модель линковки C это де-факто стандарт для взаимодействия программных компонент.

Application Binary Interface

> Набор соглашений про передачу параметров в функцию, расположение структур в памяти, etc.

Виды линковки

Статическая Линковка

Собрали объектный файл, зная только сигнатуры функций, линкер склеил с `.a` файлом.

Динамическая Линковка

Собрали объектный файл, зная только сигнатуры функций, загрузчик нашёл и добавил `.so` в адресное пространство процесса и подставил символы

dlopen

Разновидность динамической линковки: можно загрузить библиотеку по имени, и достать символ по имени

libc

По умолчанию, Rust динамически линкуется с системной `libc` — стандартной библиотекой C / ABI операционной системы:

```
fn main() {
    println!("hello");
}
```

```
$ rustc main.rs
$ ldd main
        linux-vdso.so.1 (0x00007ffc3b5bb000)
        libdl.so.2 => /nix/store/.../lib/libdl.so.2
        librt.so.1 => /nix/store/.../lib/librt.so.1
        libpthread.so.0 => /nix/store/.../lib/libpthread.so.0
        libgcc_s.so.1 => /nix/store/.../lib/libgcc_s.so.1
        libc.so.6 => /nix/store/.../lib/libc.so.6
```

libc

На Linux, можно статически слиноковаться с musl libc:

```
$ rustup target add x86_64-unknown-linux-musl
$ rustc main.rs --target x86_64-unknown-linux-musl
$ ldd main
        not a dynamic executable
$ ./main
hello
```

Можно ли написать libc на Rust? Можно, но не сильно нужно:
https://github.com/rust-lang/rfcs/issues/2610

На windows нет стабильного интерфейса системных вызовов, нужно использовать `libmsvcrt.a`.

Позвать C из Rust

./foo.c

```c
#include "stdint.h" (1)

int32_t add(int32_t x, int32_t y) {
  return x + y;
}
```

(1) используем `int32_t` для гарантированного ABI

Позвать C из Rust

./main.rs

```rust
extern "C" { (2)
    fn add(x: i32, y: i32) -> i32; (1)
}

fn main() {
    let x = unsafe { add(62, 30) }; (3)
    println!("{}", x);
}
```

1. декларируем ABI внешней функции
2. специфицируем ABI
3. вызов внешней функции требует **`unsafe`**: она может нарушить любые инварианты

Всё Вместе

./foo.c

```c
#include "stdint.h"

int32_t add(int32_t x, int32_t y) {
  return x + y;
}
```

./main.rs

```rust
extern "C" {
    fn add(x: i32, y: i32) -> i32;
}

fn main() {
    let x = unsafe { add(62, 30) };
    println!("{}", x);
}
```

Позвать C из Rust

```
$ gcc -fPIC -shared \ (1)
      -o libfoo.so  \ (2)
      foo.c
$ rustc -l foo      \ (3)
        -L .        \ (4)
        main.rs
$ env LD_LIBRARY_PATH="$LD_LIBRARY_PATH:." ./main  (5)
92
```

1. компилируем динамическую библиотеку
2. выбираем имя файла, с префиксом `lib` и расширением `.so`
3. просим `rustc` слинковаться с динамической библиотекой `foo`
4. указываем компилятору, где искать библиотеку (в текущей папке)
5. при запуске, указываем загрузчику, где искать библиотеку

Позвать C из Rust

```
$ ls -l
.rw-r--r--   75 matklad  7 Mar 22:08 foo.c
.rwxr-xr-x 9.1k matklad  8 Mar 12:01 libfoo.so
.rwxr-xr-x 194k matklad  8 Mar 12:08 main
.rw-r--r--  123 matklad  8 Mar 12:08 main.rs

$ ldd main
        linux-vdso.so.1 (0x00007fff48fc0000)
        libfoo.so => not found
        ...

$ env LD_LIBRARY_PATH="$LD_LIBRARY_PATH:." ldd main
        linux-vdso.so.1 (0x00007ffed9790000)
        libfoo.so => ./libfoo.so (0x00007f84e3ac3000)
        ...
```

Позвать Rust из C (и откуда угодно)

./foo.rs

```
#[no_mangle] (1)
pub
extern "C" (2)
fn add(x: i32, y: i32) -> i32 {
    x + y
}
```

1. отключаем name mangling
2. специфицируем ABI

Позвать Rust из C (и откуда угодно)

./main.c

```c
#include "stdio.h"
#include "stdint.h" (1)

int32_t add(int32_t x, int32_t y); (2)

int main(void) {
    int32_t x = add(62, 30);
    printf("%d\n", (int)x);
    return 0;
}
```

(1) используем `i32` в ABI

(2) декларируем ABI функции

Всё вместе

./foo.rs

```rust
#[no_mangle]
pub extern "C" fn add(x: i32, y: i32) -> i32 {
    x + y
}
```

./main.c

```c
#include "stdio.h"
#include "stdint.h"

int32_t add(int32_t x, int32_t y);

int main(void) {
    int32_t x = add(62, 30);
    printf("%d\n", (int)x);
    return 0;
}
```

Позвать Rust из C (и откуда угодно)

```
$ rustc foo.rs --crate-type cdylib (1)
$ gcc -l foo \ (2)
      -L . \ (3)
      -o main \
      main.c
$ env LD_LIBRARY_PATH="$LD_LIBRARY_PATH:." ./main (4)
92
```

1. компилируем динамическую библиотеку (`lib` и `.so` добавляются автоматически)
2. просим `gcc` слинковаться с `libfoo.so`
3. указываем компилятору где найти `libfoo.so`
4. указываем загрузчику где найти `libfoo.so`

Разное

- `[repr(C)]` для ABI структур

```
#[repr(C)]
struct Point {
    x: i32,
    y: i32,
}
```

- `build.rs` в Cargo для сборки библиотек на C/C++
- `bindgen` для генерации `extern "C"` блоков по `.h` файлам
- `cbndgen` для генерации `.h` файлов из `extern fn`
- крейт `libc` с типами вроде `c_int`
- экосистема безопасных обёрток: `libgit2-sys` + `git2`

no_std

По-умолчанию, стандартная библиотека включает сервисы операционной системы:

- поддержка динамических аллокаций ("куча")
- IO (файлы, сокеты)
- потоки
- аргументы и окружение

От этого можно отказаться, заменив `std` на `core`.

core

```
#![no_std]

#[no_mangle]
pub extern "C" fn add(x: i32, y: i32) -> i32 { x + y }

#[panic_handler] (1)
fn panic(_info: &core::panic::PanicInfo) -> ! {
    loop {}
}

$ rustc --crate-type cdylib \
        -C panic=abort      \ (2)
        foo.rs
```

1. функция, которая будет вызвана при критических ошибках (выход за границу массива)
2. запрещаем разматывание стека

Итоги

Модуль

Единица логической изоляции (видимость/доступность), допускает циклические зависимости

Крейт

Единица физической изоляции, допускает раздельную компиляцию, не допускает циклические зависимости

Статическая/Динамическая библиотека

Big picture, обеспечивает взаимодействие компонент на, возможно, разных языках программирования