Модули

Андрей Ситников Владимир Латыпов

09-11-2023

Структура занятия (построена вокруг пайплайна написания кода)

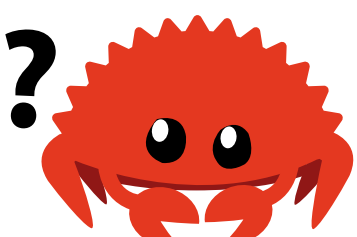
- Модули: структурировать код
- Система сборки *cargo*
- Библиотеки: публиковать и подключать
- Практика: вы будете писать *код*! (А я — *наблюдать*)

Архитектурные рассуждения

- Модули — более крупная единица группировки кода (подробнее — на Парадигмах)
- Соответствие абстракций в коде и в голове
  (применимо к любой лекции...)
- Когда хорошо структурировали код — можно опубликовать его как библиотеку
- Тёмное наследие C: не было поддержки правильных устремлений программистов на уровне языка
- В Rust поддержка есть, и вот она:

Модули

Как `cargo` собирает `crate`?



Модули

Как `cargo` собирает `crate`?

- Ищем файл `src/main.rs`

Модули

Как `cargo` собирает `crate`?

- Ищем файл `src/main.rs`
- Компилируем с помощью `rustc hello.rs` → получаем бинарник — машинный код для запуска в ОСи.

Модули

Как `cargo` собирает `crate`?

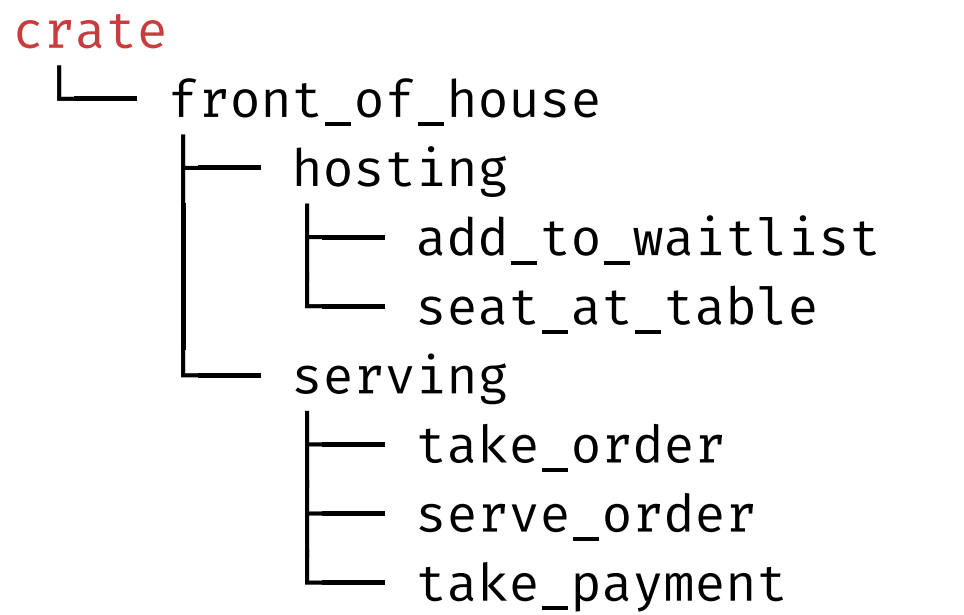
- Ищем файл `src/main.rs`
- Компилируем с помощью `rustc hello.rs` → получаем бинарник — машинный код для запуска в ОСи.
- А если хотим подключить дополнительные файлы, библиотеки?

`lib` и `bin`

- Библиотеки экспортируют функции для других `crate`-ов.
- Бинарники можно запускать

Добавляем модули

```rust
// Пример из RustBook
mod front_of_house {
    mod hosting {
        fn add_to_waitlist(){}
        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}
        fn serve_order() {}
        fn take_payment() {}
    }
}
```



- Корень — crate
- Промежутичные узлы — модули
- Конечные — объекты в коде

Файлы

```rust
// lib.rs
mod front_of_house;
```

```rust
// front_of_house/mod.rs
mod hosting;
mod serving;
```

Модули можно писать

- сразу в скобках
- в файле по имени модуля
- в `mod.rs` в папке по имени модуля

```rust
// front_of_house/hosting.rs
fn add_to_waitlist() {}

fn seat_at_table() {}
```

```rust
// front_of_house/serving/mod.rs
fn take_order() {}

fn serve_order() {}

fn take_payment() {}
```

Области видимости

По умолчанию функции и структуры видно только внутри модуля.

Ключевое слово `pub` позволяет сделать свойство доступным всем.

Области видимости

По умолчанию функции и структуры видно только внутри модуля.

Ключевое слово `pub` позволяет сделать свойство доступным всем.

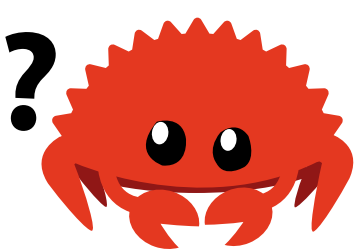
Зачем нужна приватность?

Области видимости

По умолчанию функции и структуры видно только внутри модуля.

Ключевое слово `pub` позволяет сделать свойство доступным всем.

Приватность незаменима для корректной
работы с большими проектами + библиотеками.



Зачем нужна приватность?

```rust
pub struct Vec<T, A: Allocator = Global> {
    buf: RawVec<T, A>,
    len: usize,
}
```

Корректность работы `Vec` *основана* на том, что данные в `buf` и `len` сихронизированы.

Что может случиться, если дать доступ к `buf`?

- В модуле могут содержаться функции и свойства, *неправильный вызов которых* может привести к ошибкам. Они *должны быть приватными.*
- Конструкторы нельзя использовать на приватных полях: `Vec{buf: ..., len: ...}` не сработает.
- «Безопасный» доступ к полям можно обеспечить с помощью методов вида `field` и `field_mut` (по сути — это сеттер).

Использование для `pub`:

- (очевидное) Чтобы функция/модуль была доступной извне, нужно, чтобы *модуль, в котором она лежит*, был публичным.
- (чуть менее очевидное) Все возвращаемые типы, аргументы, варианты `enum`-ов, … — тоже должны быть публично доступны.
- (кратко) В общем, компилятор сам все скажет.

Пути

Чаще всего пути встречаются в `use`:

- `use a::b::c`. Если, например, `c` — модуль, можно вызывать его элементы из кода как `c::d`.

Пути

Чаще всего пути встречаются в `use`:

- `use a::b::c`. Если, например, `c` — модуль, можно вызывать его элементы из кода как `c::d`.
- Можно вызывать одновременно: `use a::b::{c, d, e::f, g::h::i};`.

Пути

Чаще всего пути встречаются в `use`:

- `use a::b::c`. Если, например, `c` — модуль, можно вызывать его элементы из кода как `c::d`.
- Можно вызывать одновременно:
  `use a::b::{c, d, e::f, g::h::i};`.
- `self` импортирует модуль, полезно в связке:
  `use std::collections::hash_map::{self, HashMap};`

Пути

Чаще всего пути встречаются в `use`:

- `use a::b::c`. Если, например, `c` — модуль, можно вызывать его элементы из кода как `c::d`.
- Можно вызывать одновременно:
  `use a::b::{c, d, e::f, g::h::i};`.
- `self` импортирует модуль, полезно в связке:
  `use std::collections::hash_map::{self, HashMap};`
- `as` импорирует под другим именем:
  `use a::b::{self as ab, c as abc}`

Пути

Чаще всего пути встречаются в `use`:

- `use a::b::c`. Если, например, `c` — модуль, можно вызывать его элементы из кода как `c::d`.
- Можно вызывать одновременно:
  `use a::b::{c, d, e::f, g::h::i};`.
- `self` импортирует модуль, полезно в связке:
  `use std::collections::hash_map::{self, HashMap};`
- `as` импорирует под другим именем:
  `use a::b::{self as ab, c as abc}`
- `*` «импортирует всё»:
  `use a::b::{self as ab, c, d::{*, e::f}};`

Относительные пути

- Внутри крейта можно ссылаться как `crate::`
- На этот же модуль можно ссылаться как `self::`
- На родительский модуль — как `super::`.

Пути для `pub`

- `pub(in path)`

Пути для `pub`

- `pub(in path)`
- `pub(crate)`

Пути для `pub`

- `pub(in path)`
- `pub(crate)`
- `pub(super)`

Пути для `pub`

- `pub(in path)`
- `pub(crate)`
- `pub(super)`
- `pub(self)`

pub(self)?

Пути для `pub`

- `pub(in path)`
- `pub(crate)`
- `pub(super)`
- `pub(self)` — то же самое, что и без `pub`

Тесты

```
#[test]
  fn internal() {
      assert_eq!(4, internal_adder(2, 2));
  }
```

mod tests

```
1 pub fn add(a: i32, b: i32) →
  i32 {
2   a + b
3 }
4
5 #[cfg(test)]
6 mod tests {
7   use super::*;
8
9   #[test]
10  fn test_add() {
11      assert_eq!(add(1, 2),
  3);
12  }
13 }
```

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 1.31s
     Running unittests src/lib.rs (target/debug/deps/adder-1082c4b063a8fbe6)

running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

     Running tests/integration_test.rs (target/debug/deps/integration_test-1082c4b063a8fbe6)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

Сборка

Компилятор `Rust` — `rustc`. Им можно собрать что-то простое:

Сборка

Компилятор `Rust` — `rustc`. Им можно собрать что-то простое:

```
$ rustc hello.rs
$ ./hello
Hello, world!
```

(компилятор за один раз собирает один *crate*)

Сборка

Компилятор `Rust` — `rustc`. Им можно собрать что-то простое:

```
$ rustc hello.rs
$ ./hello
Hello, world!
```

(компилятор за один раз собирает один *crate*)

У компилятора куча опций для сборки под разные цели, и собирать в общем-то неплохо.

Но что-то большое с зависимостями собирать больно.

`C++` обычно использует `cmake`. Работать с ним не очень приятно. У `Rust` есть уже знакомый нам...

cargo

`cargo` позволяет работать с `packages`, а не отдельными крейтами.

`cargo` делает четыре вещи (взято из *The Cargo Book*):

- Работает с двумя файлами с информацией о пакетах.
- Подгружает, если нужно, и собирает зависимости.
- Вызывает `rustc` (на самом деле не обязательно) с нужными параметрами для сборки пакета.
- Добавляет команды для более удобной работы.

`toml` и `lock`

Cargo.toml:

```toml
[package]
name = "example"
version = "0.1.0"
edition = "2021"

[dependencies]
csv = "1.2.2"
itertools = "*"
```

Здесь можно настраивать компиляцию, зависимости и фичи.

Cargo.lock:

```
# This file is automatically @generated by
Cargo.
# It is not intended for manual editing.
version = 3

[[package]]
name = "autocfg"
version = "1.1.0"
source = "registry+https://github.com/ ... "
checksum = "d468802bab17cbc0cc575e ... "

[[package]]
name = "cfg-if"
version = "1.0.0"
```

Устройство `toml`

TOML

```
[name]
key = value
key2 = {v = 79.5}
```

```
[[name]]
key = value
[[name]]
["delta","phi"]
```

JSON

```
{"name":
  {"key": "value",
  {"key2": 79.5}}
}
```

```
{"name":
  [{"key": "value"},
   ["delta", "phi"]],
}
```

`--debug` и `--release`

`debug` оставляет настоящие имена функций, адреса. В случае паники легче отдебажить.

`release` сильно оптимизирует код.

`--debug` и `--release`

`debug` оставляет настоящие имена функций, адреса. В случае паники легче отдебажить.

`release` сильно оптимизирует код.

Не надо замерять скорость кода под `debug`!

Настройка профилей

Профили можно создавать и кастомизировать.

По умолчанию есть `dev` (`debug`), `release`, `test`, и `bench`.

```
[profile.dev]
opt-level = 1                # Use slightly better
optimizations.
overflow-checks = false      # Disable integer overflow
checks.
```

Настройка профилей

Много всего, см. `cargo book`.

- Оптимизации на размер бинарника.
- `lto`, `link-time-optimization`.
- Поведение при `panic`.
- Отделение `debug` информации от бинарника

Features

```
[features]
default = ["ico", "webp"]
bmp = []
png = []
ico = ["bmp", "png"]
webp = []
```

Фичи — глобальные настройки вашего приложения. Они могут зависеть друг от друга, задаваться по умолчанию.

Чтобы использовать, надо передать `--features ...` в вызов.

`cfg`

Макрос `#[cfg]` или `cfg!` позволяет компилировать программу по-разному в зависимости от параметров компиляции.

```
1   // This conditionally includes a module which implements WEBP
  support.
2 #[cfg(feature = "webp")]
3 pub mod webp;
```

Другие параметры cfg

```rust
  // The function is only included in the build when compiling for macOS
#[cfg(target_os = "macos")]
fn macos_only() {}

// This function is only included when either foo or bar is defined
#[cfg(any(foo, bar))]
fn needs_foo_or_bar() {}

// This function is only included when compiling for a unixish OS with a 32-bit
// architecture
#[cfg(all(unix, target_pointer_width = "32"))]
fn on_32bit_unix() {}

// This function is only included when foo is not defined
#[cfg(not(foo))]
fn needs_not_foo() {}

// This function is only included when the panic strategy is set to unwind
#[cfg(panic = "unwind")]
fn when_unwinding() {}

fn check_machine() {
  let machine_kind = if cfg!(unix) {
    "unix"
  } else if cfg!(windows) {
    "windows"
  } else {
```

```
28      "unknown"
29    };
30
31    println!("Running on {}",
machine_kind);
32  }
```

`build.rs`

Можно выполнять кастомные скрипты при компиляции.

```rust
// Example custom build script.
fn main() {
    // Tell Cargo that if the given file changes, to rerun this build script.
    println!("cargo:rerun-if-changed=src/hello.c");
    // Use the `cc` crate to build a C file and statically link it.
    cc::Build::new()
        .file("src/hello.c")
        .compile("hello");
}
```

Инкрементальная компиляция

- Разработчи часто вносит небольшие изменения, и хочет, чтобы компилятор его понимал и быстро перекомпилировал
- На уровне crate-ов (особенно полезно для библиотек)
- На уровне запросов (DAG зависимостей)

Дополнительные инструменты

`cargo add`

Добавляет последнюю версию библиотеки в `Cargo.toml`.

`rustfmt`

```
$ rustup component add rustfmt
```

Отформатировать код: `cargo fmt`

`cargo clippy`

```
$ rustup component add clippy
```

`cargo clippy (--fix)`

```rust
let mut dst = vec![1, 2, 3];
let src = vec![4, 5];

for i in 0..src.len() {
    dst[i + 1] = src[i];
}
```

```rust
let mut dst = vec![1, 2, 3];
let src = vec![4, 5];

for i in 0..src.len() {
    dst[i + 1] = src[i];
}
```

```
warning: it looks like you're manually copying between slices
  --> src\main.rs:85:5
   |
85 | /     for i in 0..src.len() {
86 | |         dst[i + 1] = src[i];
87 | |     }
   | |_____^ help: try replacing the loop by: `dst[1..(src.len() +
1)].copy_from_slice(&src[..]);`
   |
   = help: for further information visit https://rust-lang.github.io/rust-
```

```
clippy/master/index.html#manual_memcpy
   = note: `#[warn(clippy::manual_memcpy)]` on by default
```

rustdoc

Чтобы ваш было не стыдно публиковать:

Напишите документацию внутри кода (ещё можно тесты):

```
1 //! Library for computing sum of 32-bit integers
2
3 /// Function that computes sum of 32-bit integers
4 fn compute_sum(a: i32, b: i32) { a + b }
```

И сгенерите веб-страницу для её просмотра (на уровне полноценных библиотек):

```
$ rustdoc src/lib.rs
```

Работа с библиотеками

Установка

Все подгружается автоматически по `Cargo.toml`.

Features библиотек

```
[dependencies]
# Enables the `derive` feature of serde.
serde = { version = "1.0.118", features = ["derive"] }
```

Примеры

Примеров много, экосистема с библиотеками весьма богатая. Мы сейчас рассмотрим несколько самых полезных, ещё некоторые рассмотрим потом.

`itertools`

Реализует дополнительные `trait`-ы для итераторов с огромным количеством новых функций.

```
use itertools::Itertools;

let it = (1..3).interleave(vec![-1, -2]);
itertools::assert_equal(it, vec![1, -1, 2, -2]);
```

Очень полезно, если не хватает методов для итераторов.

`serde`

Если коротко — библиотека для (де)сериализации примерно всего.

Одна из важных фич — позволяет выгружать файлы данных *прямо в структуру* с помощью *процедурных макросов*.

```rust
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize, Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point = Point { x: 1, y: 2 };

    // Convert the Point to a JSON string.
    let serialized = serde_json::to_string(&point).unwrap();

    // Prints serialized = {"x":1,"y":2}
    println!("serialized = {}", serialized);

    // Convert the JSON string back to a Point.
    let deserialized: Point = serde_json::from_str(&serialized).unwrap();
```

```
20
21      // Prints deserialized = Point { x: 1, y: 2 }
22      println!("deserialized = {:?}", deserialized);
23  }
```

lazy-static

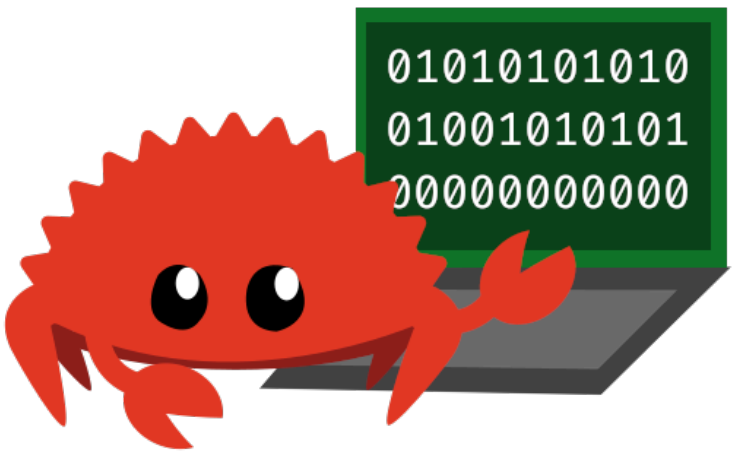
Библиотека, позволяющая лениво создавать статические переменные *во время исполнения*.

```rust
use lazy_static::lazy_static;
use std::collections::HashMap;

lazy_static! {
    static ref HASHMAP: HashMap<u32, &'static str> = {
        let mut m = HashMap::new();
        m.insert(0, "foo");
        m.insert(1, "bar");
        m.insert(2, "baz");
        m
    };
}

fn main() {
    // First access to `HASHMAP` initializes it
    println!("{}", HASHMAP.get(&0).unwrap());

    // Any further access to `HASHMAP` just returns the computed value
```

```
19      println!("{}", HASHMAP.get(&1).unwrap());
20  }
```

Live coding (если успеем)

- Выбираем и подключаем библиотеки
- Смотрим документацию
- Копируем в свой код
- Profit!



Специально выбрал рандомную библиотеку, с которой никогда не работал.

Работаем руками

- Trait&generics
- Iterators

Кто сделал практику
про итераторы?

Работаем руками

- Trait&generics
- Iterators

Кто посмотрел лекцию
про итераторы?

Работаем руками

- Trait&generics
- Iterators

Кто открывал лекцию
про итераторы?

Работаем руками

- Trait&generics
- Iterators

Понятно ли условие 3-го ДЗ про traits, generics?