ГЛАВА 1

Типы

Первая глава книги содержит рекомендации, связанные с системой типов Rust. Его система типов более экспрессивна, чем ее аналоги в других популярных языках, и имеет больше общего с системой типов в академических языках вроде OCaml или Haskell. Одной из основных составляющих этой системы является тип `enum`, который значительно более выразителен, чем типы перечислений в других языках, и делает возможным использование *алгебраических типов данных*.

Рекомендации, приводимые в этой главе, посвящены фундаментальным типам, которые предоставляет Rust, и объясняют, как комбинировать их в структуры данных, точно выражающие семантику вашей программы. Этот принцип кодирования поведения в систему типов помогает снизить объем проверок и кода, необходимого для выявления ошибок пути, поскольку недействительные состояния отклоняются цепочкой инструментов в процессе компиляции, а не программой в среде выполнения.

В этой главе также описываются некоторые распространенные структуры данных, предоставляемые стандартной библиотекой Rust: `Options`, `Results`, `Error` и `Iterators`. Знакомство с этими стандартными инструментами поможет вам писать идиоматический код Rust, отличающийся эффективностью и компактностью, — в частности, они позволяют использовать оператор вопросительного знака, дающий возможность обрабатывать ошибки необременительным, но при этом типобезопасным способом.

Имейте в виду, что рекомендации в отношении *трейтов* Rust мы рассмотрим в следующей главе, но они будут неизбежно пересекаться с рекомендациями этой главы, поскольку трейты описывают поведение типов.

Рекомендация 1. Используйте систему типов для выражения структур данных

> Кто назвал их программистами, а не составителями типов?
>
> *@thingskatedid (https://oreil.ly/hHj5c)*

В этом разделе мы кратко рассмотрим систему типов Rust, начиная с фундаментальных типов, которые предоставляет компилятор, и заканчивая различными способами совмещения значений в структуры данных.

Далее мы переключимся на тип `enum`. И хотя в базовой версии он равнозначен своим аналогам в других языках, возможность комбинировать вариации `enum` с полями данных обеспечивает повышенную гибкость и экспрессивность.

Фундаментальные типы

Основы системы типов Rust будут знакомы всем, кто имеет опыт работы со статически типизируемыми языками, например C++, Go или Java. Здесь мы имеем набор целочисленных типов конкретных размеров — как знаковых (`i8`, `i16`, `i32`, `i64`, `i128`), так и беззнаковых (`u8`, `u16`, `u32`, `u64`, `u128`).

Кроме того, есть знаковые (`isize`) и беззнаковые (`usize`) целые, чей размер соответствует размеру указателя в целевой системе. В Rust вам не потребуется часто заниматься преобразованием между указателями и целыми, так что равнозначность их размера значения не имеет. Тем не менее стандартные коллекции возвращают свой размер в виде `usize` (из `.len()`), поэтому при индексировании коллекции значения `usize` являются типичными, что естественно с точки зрения емкости, поскольку находящаяся в памяти коллекция не может содержать больше элементов, чем имеется адресов памяти в системе.

Целочисленные типы дают нам первое указание на то, что Rust устроен строже, чем C++. В Rust попытка поместить более крупный целочисленный тип (`i32`) в аналогичный тип меньшего размера (`i16`) ведет к ошибке при компиляции:

НЕ КОМПИЛИРУЕТСЯ

```
let x: i32 = 42;
let y: i16 = x;
```

```
error[E0308]: mismatched types
  --> src/main.rs:18:18
```

```
   |
18 |     let y: i16 = x;
   |            ---   ^ expected `i16`, found `i32`
   |            |
   |            expected due to this
   |
help: you can convert an `i32` to an `i16` and panic if the converted value
      doesn't fit
   |
18 |     let y: i16 = x.try_into().unwrap();
   |                   ++++++++++++++++++++
```

Такое поведение успокаивает: Rust не будет сидеть молча, пока программист делает что-то рискованное. И хотя мы можем видеть, что значения в этом конкретном преобразовании вполне для него подходят, компилятор должен позволять использовать такие, при которых преобразование будет *недопустимо*.

НЕ КОМПИЛИРУЕТСЯ

```
let x: i32 = 66_000;
let y: i16 = x; // Каким будет это значение?
```

Этот вывод ошибки также на раннем этапе показывает, что, несмотря на повышенную строгость Rust, он содержит полезные сообщения компилятора, которые указывают, как сделать все в соответствии с правилами. Предлагаемое решение поднимает вопрос о том, как быть в ситуациях, когда преобразованию приходится изменять значение, приводя его в соответствующий вид, и позже мы более подробно поговорим об обработке ошибок (см. рекомендацию 4) и использовании `panic!` (см. рекомендацию 18).

Rust также не позволяет выполнять кое-что, что может показаться безопасным, например помещать значение из меньшего целочисленного типа в больший:

НЕ КОМПИЛИРУЕТСЯ

```
let x = 42i32; // Целочисленный литерал с суффиксом типа
let y: i64 = x;
```

```
error[E0308]: mismatched types
  --> src/main.rs:36:18
   |
36 |     let y: i64 = x;
   |            ---   ^ expected `i64`, found `i32`
   |            |
   |            expected due to this
```

```
   |
help: you can convert an `i32` to an `i64`
   |
36 |     let y: i64 = x.into();
   |                  +++++++
```

Здесь предлагаемое решение не подразумевает обработку ошибки, но преобразование все равно должно быть явным. Мы более подробно поговорим о преобразованиях в рекомендации 5.

Список стандартных примитивных типов в Rust продолжают `bool`, типы с плавающей запятой (`f32`, `f64`) и единичный тип `()` (как `void` в C). Но более интересен символьный тип, который содержит значение Юникода (аналогичен типу `rune` в Go). И хотя он сохраняется в виде 4 байт, его также нельзя беспрепятственно преобразовать в 32-битное целое или из него.

Точность в системе типов вынуждает вас явно выражать свои намерения — значение `u32` отличается от `char`, которое отличается от последовательности байтов UTF-8, а она, в свою очередь, отличается от последовательности произвольных байтов, и именно вам нужно точно указывать, что имеется в виду[1]. Лучше разобраться в том, какой тип в какой ситуации подходит, вам поможет известная статья Джоэля Спольски (Joel Spolsky) (https://oreil.ly/wWy7T).

Естественно, существуют вспомогательные методы, которые позволяют выполнять преобразование между этими типами, но их сигнатуры вынуждают вас обрабатывать (или открыто игнорировать) возможность сбоя. Например, кодовую точку Юникода всегда можно представить 32 битами[2], значит, `'a' as u32` использовать допустимо, но в обратном направлении все несколько сложнее (есть значения `u32`, которые не являются валидными кодовыми точками Юникода).

`char::from_u32`

Возвращает `Option<char>`, вынуждая вызывающий код обрабатывать сбой.

`char::from_u32_unchecked`

Делает допущение в отношении валидности, но, если оно окажется ложным, это может привести к неопределенному поведению. В результате данная функция обозначается как `unsafe`, заставляя вызывающий код использовать также `unsafe` (см. рекомендацию 16).

---

[1] Ситуация становится еще более запутанной, если в процесс вовлечена файловая система, поскольку имена файлов на популярных платформах представляют собой нечто среднее между произвольными байтами и последовательностями UTF-8 (читайте документацию std::ffi::OsString по адресу https://doc.rust-lang.org/std/ffi/struct.OsString.html).

[2] Технически речь идет о скалярном значении Юникода (https://www.unicode.org/glossary/#unicode_scalar_value), а не о кодовой точке.

Составные типы

Далее идут составные типы. В Rust есть разные способы совмещения связанных значений. Большинство из них представляют собой знакомые эквиваленты механизмов агрегации, доступных в других языках.

Массивы
: Содержат множество экземпляров одного типа, количество которых известно на этапе компиляции. Например, `[u32; 4]` — это четыре 4-байтовых целых подряд.

Кортежи
: Содержат экземпляры разных типов, когда количество элементов и их типов известно на этапе компиляции, например `(WidgetOffset, WidgetSize, WidgetColor)`. Если типы в кортеже повторяются, например `(i32, i32, &'static str, bool)`, лучше дать каждому элементу имя и использовать структуру.

Структуры
: Также содержат экземпляры разных типов, известные на этапе компиляции, но при этом позволяют использовать как общий тип, так и отдельные поля для обращения по имени.

Rust также включает в себя *структуру кортежа*, представляющую смесь `struct` и кортежа: в ней есть имя для общего типа, но нет имен для отдельных полей. Обращение к ним происходит по номеру — `s.0`, `s.1` и т. д.:

```
/// Структура с двумя неименованными полями
struct TextMatch(usize, String);

// Построение путем упорядоченного предоставления содержимого
let m = TextMatch(12, "needle".to_owned());

// Обращение по номеру поля
assert_eq!(m.0, 12);
```

Перечисления

Мы подошли к бриллианту в короне системы типов Rust — `enum`. Рассматривая базовую форму этого типа, сложно понять, что в нем такого восхитительного. Как и в других языках, `enum` позволяет указывать набор взаимно исключающих значений, в том числе с прикреплением числа:

```
enum HttpResultCode {
    OK = 200,
    NotFound = 404,
```

```
    Teapot = 418,
}

let code = HttpResultCode::NotFound;
assert_eq!(code as i32, 404);
```

Поскольку каждое определение `enum` создает отдельный тип, это можно использовать для повышения читаемости и обслуживаемости функций, получающих аргументы `bool`. Используемая вместо:

```
print_page(/* both_sides= */ true, /* color= */ false);
```

версия с парой `enum`:

```
pub enum Sides {
    Both,
    Single,
}

pub enum Output {
    BlackAndWhite,
    Color,
}

pub fn print_page(sides: Sides, color: Output) {
    // ...
}
```

будет более типобезопасной и лучше читаемой в точке вызова:

```
print_page(Sides::Both, Output::BlackAndWhite);
```

В отличие от версии `bool`, если пользователь библиотеки случайно изменит порядок аргументов, компилятор тут же начнет ругаться:

```
error[E0308]: arguments to this function are incorrect
   --> src/main.rs:104:9
    |
104 | print_page(Output::BlackAndWhite, Sides::Single);
    | ^^^^^^^^^^ -------------------- ------------- expected `enums::Output`,
    |            |                                  found `enums::Sides`
    |            |
    |            expected `enums::Sides`, found `enums::Output`
    |
note: function defined here
   --> src/main.rs:145:12
    |
145 |    pub fn print_page(sides: Sides, color: Output) {
    |           ^^^^^^^^^^ ------------  -------------
help: swap these arguments
    |
104 | print_page(Sides::Single, Output::BlackAndWhite);
    |             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Используя паттерн newtype (читайте рекомендацию 6) для обертывания `bool`, вы получаете также безопасность типов и обслуживаемость. Лучше всего задействовать этот паттерн, если семантика всегда будет логической, и применять `enum`, если есть вероятность возникновения в будущем новой альтернативы, например `Sides::BothAlternateOrientation`.

Безопасность типов при использовании `enum` в Rust сохраняется также за счет выражения `match`:

НЕ КОМПИЛИРУЕТСЯ

```
let msg = match code {
    HttpResultCode::OK => "OK",
    HttpResultCode::NotFound => "Not found",
    // Забыл обработать важный код "I'm a teapot"
};
```

```
error[E0004]: non-exhaustive patterns: `HttpResultCode::Teapot` not covered
  --> src/main.rs:44:21
   |
44 |     let msg = match code {
   |                     ^^^^ pattern `HttpResultCode::Teapot` not covered
   |
note: `HttpResultCode` defined here
  --> src/main.rs:10:5
   |
7  | enum HttpResultCode {
   |      --------------
...
10 |     Teapot = 418,
   |     ^^^^^^ not covered
   = note: the matched value is of type `HttpResultCode`
     help: ensure that all possible cases are being handled by adding
     a match arm with a wildcard pattern or an explicit pattern as shown
   |
46 ~         HttpResultCode::NotFound => "Not found",
47 ~         HttpResultCode::Teapot => todo!(),
   |
```

Компилятор заставляет программиста учесть *все* возможности, предоставленные `enum`[1], даже если результатом будет просто добавление предустановленной

[1] Необходимость учитывать все возможные варианты также означает, что добавление нового варианта в существующее enum в библиотеке приведет к *критическому изменению* (см. рекомендацию 21): клиентам библиотеки придется изменять свой код, чтобы он обрабатывал и новый вариант. Если же enum является просто C-подобным списком связанных численных значений, такого поведения можно избежать, сделав его non_exhaustive enum (подробнее — в рекомендации 21).

ветки `_ => {}`. (Заметьте, что современные компиляторы C++ тоже предупреждают об отсутствии веток `switch` для `enum`.)

Перечисления с полями

Истинная сила `enum` в Rust основывается на том, что каждый содержащийся в нем вариант может иметь сопутствующие данные. Это делает перечисление составным типом, который выступает в качестве *алгебраического типа данных* (algebraic data type, ADT). Эта концепция может быть не знакома программистам, работающим с мейнстримными языками. Например, в системе C/C++ это подобно комбинации `enum` с `union`, только типобезопасной.

Это означает, что инварианты структур данных программы можно закодировать в систему типов Rust. Состояния, которые не соответствуют этим инвариантам, просто не скомпилируются. Грамотно построенное перечисление отчетливо демонстрирует намерение его создателя как для людей, так и для компилятора:

```
use std::collections::{HashMap, HashSet};

pub enum SchedulerState {
    Inert,
    Pending(HashSet<Job>),
    Running(HashMap<CpuId, Vec<Job>>),
}
```

Исходя из определения типа, разумно предположить, что, пока планировщик полностью активен, `Job` добавляются в очередь в состоянии `Pending` и присваиваются пулу того или иного ядра процессора. Это подчеркивает центральную тему данной рекомендации, которая заключается в использовании системы типов Rust для выражения принципов, связанных с проектированием вашего ПО.

Явным признаком случая, когда этого не происходит, является комментарий, который объясняет, когда определенное поле или параметр валидны.

НЕЖЕЛАТЕЛЬНОЕ ПОВЕДЕНИЕ

```
pub struct DisplayProps {
    pub x: u32,
    pub y: u32,
    pub monochrome: bool,
    // `fg_color` должен быть (0, 0, 0), если `monochrome` равен true
    pub fg_color: RgbColor,
}
```

Это первый кандидат на замену с помощью `enum`, содержащего данные:

```
pub enum Color {
    Monochrome,
    Foreground(RgbColor),
}

pub struct DisplayProps {
    pub x: u32,
    pub y: u32,
    pub color: Color,
}
```

Этот небольшой пример иллюстрирует суть рекомендации: *делайте невалидные состояния в ваших типах невыражаемыми*. Типы, которые поддерживают только валидные комбинации значений, свидетельствуют о том, что компилятор будет отвергать целый класс ошибок, обеспечивая более компактный и безопасный код.

Распространенные типы перечислений

Возвращаясь к потенциалу `enum`, следует сказать, что есть два настолько широко распространенных принципа, что стандартная библиотека Rust содержит типы `enum` для их выражения. В коде Rust эти типы встречаются повсеместно.

Option<T>

Первый принцип выражается в типе `Option`, который показывает, что значение определенного типа либо присутствует (`Some(T)`), либо нет (`None`). *Всегда используйте* `Option` *для значений, которые могут отсутствовать*. Никогда не прибегайте к использованию сигнальных значений (`-1`, `nullptr`, …), стараясь выразить тот же принцип в рамках диапазона.

Но здесь нужно учитывать один нюанс. Если вы работаете с *коллекцией*, то следует решить, будет ли присутствие нуля элементов в ней равнозначно ее отсутствию. В большинстве ситуаций различий не будет, и вы вполне можете использовать, скажем, `Vec<Thing>`, когда ноль элементов подразумевает их отсутствие.

Тем не менее есть и другие, редкие сценарии, когда два этих случая нужно разделить с помощью `Option<Vec<Thing>>`, — например, криптографической системе может потребоваться различать полезную нагрузку, передаваемую отдельно (https://oreil.ly/vuL006Co), и пустую переданную полезную нагрузку. (Это относится к спорам вокруг маркера `NULL` для столбцов SQL.)

В том же смысле как лучше всего поступить в случае `String`, которая может отсутствовать? Как разумнее указывать на отсутствие значения — с помощью `""` или `None`? Сработает и то и другое, но `Option<String>` отчетливо покажет вероятность того, что это значение может отсутствовать.

Result<T, E>

Второй типичный принцип возникает в контексте обработки ошибок: если функция дает сбой, как об этом нужно сообщать? Традиционно для этого использовались специальные сигнальные значения (например, `-errno`, возвращаемое от системных вызовов в Linux) или глобальные переменные (`errno` в системах POSIX). В последние же годы языки, которые поддерживают множественные или кортежные возвращаемые значения (как в Go) от функций, могут по соглашению возвращать пару `(result, error)`, предполагая существование некоего подходящего нулевого значения для `result`, когда `error` ненулевое.

В Rust конкретно для этого есть специальное `enum`: *всегда кодируйте результат операции, которая может провалиться, в виде* `Result<T, E>`. Тип `T` содержит успешный результат (в варианте `OK`), а тип `E` — детали об ошибке (в варианте `Err`) в случае сбоя.

Использование стандартного типа отчетливо проясняет замысел кода. Это, в частности, позволяет применять стандартные трансформации (см. рекомендацию 3) и обработку ошибок (см. рекомендацию 4), которые, в свою очередь, дают возможность упрощать обработку ошибок с помощью оператора `?`.

Рекомендация 2. Используйте систему типов для выражения типичного поведения

В рекомендации 1 мы разобрали, как в системе типов выражать структуры данных. Текущая же рекомендация посвящена кодированию в системе типов *поведения* Rust.

Описанные здесь механизмы покажутся знакомыми, поскольку имеют прямые аналоги в других языках.

Функции
: Универсальный механизм для ассоциирования фрагмента кода с именем и списком параметров.

Методы
: Функции, которые ассоциируются с экземпляром конкретной структуры данных. Методы типичны для языков, разработанных на основе парадигмы объектно-ориентированного программирования.

Указатели функций
: Поддерживаются в большинстве языков семейства C, включая C++ и Go, в качестве механизма, позволяющего добавлять еще один уровень косвенности при вызове другого кода.

Замыкания

Изначально характерны преимущественно для семейства Lisp, но были модифицированы для многих популярных языков программирования, включая C++ (начиная с C++ 11) и Java (начиная с Java 8).

Трейты

Описывают коллекцию связанной функциональности, применяемой к одному элементу. Трейты имеют примерные эквиваленты во многих других языках, включая абстрактные классы в C++, а также интерфейсы в Go и Java.

Естественно, все эти механизмы в Rust имеют характерные особенности, которые будут раскрыты в данной рекомендации.

Из всего перечисленного списка трейты играют в книге наиболее важную роль, поскольку описывают значительную часть поведения, предоставляемого компилятором и стандартной библиотекой Rust. Глава 2 будет посвящена рекомендациям по проектированию и реализации трейтов, но то, что они встречаются повсеместно, означает, что они будут всплывать в других рекомендациях и текущей главы.

Функции и методы

Как и в любом другом языке программирования, в Rust *функции* служат для организации кода в виде именованных частей, предполагающих повторное использование и получающих входные данные в виде параметров. Как и в любом статически типизируемом языке, здесь типы параметров и возвращаемое значение указываются явно:

```
/// Возвращает `x`, деленное на `y`
fn div(x: f64, y: f64) -> f64 {
    if y == 0.0 {
        // Завершает функцию и возвращает значение
        return f64::NAN;
    }
    // Неявно возвращается последнее выражение в теле функции
    x / y
}

/// Функция вызывается только ради своих побочных эффектов
/// без возвращаемого значения
/// Можно также записать возвращаемое значение как`-> ()`
fn show(x: f64) {
    println!("x = {x}");
}
```

Если функция тесно вписана в какую-то структуру данных, она выражается в виде *метода*. Он действует в отношении элемента этого типа, определяемого `self`, и включается в блок `impl DataStructure`. Таким образом связанные данные и код по

аналогии с другими языками инкапсулируются вместе объектно-ориентированным способом. Тем не менее в Rust методы можно добавлять в типы `enum` и `struct`, сохраняя всепроникающую природу `enum` в этом языке (см. рекомендацию 1):

```
enum Shape {
    Rectangle { width: f64, height: f64 },
    Circle { radius: f64 },
}

impl Shape {
    pub fn area(&self) -> f64 {
        match self {
            Shape::Rectangle { width, height } => width * height,
            Shape::Circle { radius } => std::f64::consts::PI * radius * radius,
        }
    }
}
```

Имя метода создает метку кодируемого им поведения, а его сигнатура несет информацию о типе его ввода и вывода. Первым вводом метода будет некий вариант `self`, указывающий, что метод может делать с этой структурой данных.

- Параметр `&self` указывает, что содержимое структуры данных можно считывать, но нельзя изменять.
- Параметр `&mut self` указывает, что метод может изменять содержимое структуры данных.
- Параметр `self` указывает, что метод получает структуру данных.

Указатели функций

В предыдущем разделе было описано, как ассоциировать имя и список параметров с неким кодом. Однако вызов функции всегда приводит к выполнению одного и того же кода. Между вызовами меняются только данные, с которыми работает функция. Это охватывает множество возможных сценариев, но что, если в среде выполнения *код* должен изменяться?

Простейшей поведенческой абстракцией, которая позволяет это реализовать, является *указатель функции* — указатель только на какой-то код с типом, отражающим сигнатуру его функции:

```
fn sum(x: i32, y: i32) -> i32 {
    x + y
}
// Требуется явное приведение к типу `fn`...
let op: fn(i32, i32) -> i32 = sum;
```