Rust

Обработка ошибок и ООП

Типы ошибок

В большинстве ЯП ошибки(исключения) не делятся на типы и обрабатываются одинаково. В Rust нет исключений, а ошибки делятся на 2 категории. Каждые обрабатываются по разному.

Исправимые(recoverable)

`Result<T, E>`

Сообщаем о проблеме,
программа не прерывается

Неисправимые(unrecoverable)

`panic!`

Немедленная остановка
программы и *раскрутка стека*

Неисправимые ошибки с `panic!`

- Явный и неявный вызов `panic!`

```rust
fn main() {
    panic!("crash and burn");
}
```

```rust
fn main() {
    let v = vec![1, 2, 3];

    v[99];
}
```

- *Немедленное прерывание (aborting)*

```
[profile.release]
panic = 'abort'
```

→ Cargo.toml

- *Обратная трассировка (Backtracing)*

```
RUST_BACKTRACE = 1 cargo run
```

Исправимые ошибки с `Result<T, E>`

```rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

```rust
use std::fs::File;

fn main() {
    let greeting_file_result = File::open("hello.txt");
}
```

`Ok`(дескриптор файла)

`Err`(дополнительная информация о том, какая ошибка произошла)

Обработка ошибок. (1/2)

```rust
use std::fs::File;

fn main() {
    let greeting_file_result = File::open("hello.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => panic!("Problem opening the file: {:?}", error),
    };
}
```

```rust
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let greeting_file_result = File::open("hello.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Problem creating the file: {:?}", e),
            },
            other_error => {
                panic!("Problem opening the file: {:?}", other_error);
            }
        },
    };
}
```

Обработка ошибок. (2/2)

```rust
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt").unwrap();
}
```

```rust
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt")
        .expect("hello.txt should be included in this project");
}
```

Проброс ошибок.

```rust
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let username_file_result = File::open("hello.txt");

    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut username = String::new();

    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(e) => Err(e),
    }
}
```

```rust
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let mut username_file = File::open("hello.txt")?;
    let mut username = String::new();
    username_file.read_to_string(&mut username)?;
    Ok(username)
}
```

`panic!` или НЕ `panic!`
Вот в чем вопрос

Вызвать `panic!`
(паникует компилятор)

- Написание примеров, прототипов, тестов
- Некорректное состояние

Обрабатывать ошибку
(паникуем мы)

- Ожидаемые сбои

ООП в Rust

есть или нет…?

ООП в Rust

- Абстракция данных

  В Rust абстракция данных представлена структурами.

- Инкапсуляция

  В Rust есть инкапсуляция, работает через методы, определённые на структурах.

  Сокрытие также есть — существуют общие и частные поля структур и методы. Частные элементы доступны в реализации функциональности, но недоступны снаружи.

- Наследование

  В Rust нет классического наследования, но есть возможность изобразить его с помощью типажей.

- Полиморфизм подтипов

  В Rust есть полиморфизм подтипов и реализуется он через типажи и типажи-объекты.

Структуры. Пример

```
pub struct AveragedCollection {
    list: Vec<i32>,
    average: f64,
}
```

```
impl AveragedCollection {
    pub fn add(&mut self, value: i32) {
        self.list.push(value);
        self.update_average();
    }

    pub fn average(&self) -> f64 {
        self.average
    }

    fn update_average(&mut self) {
        let total: i32 = self.list.iter().sum();
        self.average = total as f64 / self.list.len()
as f64;
    }
}
```

Сокрытие частных полей

- Разделение частных и общих полей и методов работает на уровне **модулей**.
- Внутри модуля все функции, методы и поля структур доступны без ограничений — независимо от того, являются они частными или общими.
- Вне модуля частные элементы в общем случае не доступны
- Модули могут вкладываться друг в друга. При этом частные элементы вышестоящих модулей доступны во вложенных модулях

Пример:

В корневом модуле может быть модуль `a`, в нём модуль `b`, а в нём — модуль `c`.

Частные элементы модуля `a` доступны в модулях `a`, `b` и `c`, но не доступны в корневом модуле.

Частные элементы `b` видимы в `b` и `c`.

И, наконец, частные элементы `c` доступны только в `c`.

Сокрытие частных полей. Пример

```
mod aaa {
    fn foo(inner: bbb::Inner) {
        // Есть доступ к публичному полю.
        let a = inner.public;

        // Ошибка компиляции: попытка обращения к приватному полю.
        let b = inner.private;

        // Ошибка компиляции: попытка использования приватной структуры.
        let c = bbb::Private {};
    }

    mod bbb {
        pub struct Inner {
            private: i32,
            pub public: i32,
        }

        struct Private {}
    }
}
```

Наследование

- Наследование описывает отношение "является" между двумя объектами. При этом, дочерний объект может быть использован в любом контексте, в котором ожидается родительский объект. Для этого необходимо, чтобы функционал базового объекта присутствовал, также, и в дочернем.
- В Rust существуют отличия от классического подхода к реализации данной идеи — через классы и интерфейсы.
- **В Rust отсутствует наследование** структур, а, следовательно, и наследование данных.

Полиморфизм

- В Rust, полиморфизм достигается с использованием **трейтов (типаж, traits)**
- **Trait** это механизм, который позволяет определить совокупность методов, которые могут быть реализованы для различных типов данных. Типажи предоставляют абстрактный интерфейс, описывающий общее поведение, но не предоставляют собой конкретной реализации.
- В других языках программирования есть в некоторой степени похожая функциональность - интерфейсы.
- Типажи могут быть реализованы для любых типов данных.

Traits. Пример

- Сначала мы определяем сигнатуры методов типажа в коде
- Когда структура реализует типаж, она устанавливает контракт поведения, который позволяет нам косвенно взаимодействовать со структурой через тип данного типажа без необходимости знать реальный тип.

```rust
trait Shape {
    // У любой фрормы можно посчитать площадь.
    fn area(&self) -> f32;
}

trait HasAngles: Shape {
    // У любой фигуры с углами можно посчитать количество углов.
    fn angles_count(&self) -> i32;
}

struct Rectangle {
    x: f32,
    y: f32,
}
```

```rust
// Прямоугольник является формой.
impl Shape for Rectangle {
    fn area(&self) -> f32 {
        self.x * self.y
    }
}

// Прямоугольник является фигурой с углами.
impl HasAngles for Rectangle {
    fn angles_count(&self) -> i32 {
        4
    }
}
```

Зачем нужны типажи?

- Полиморфизм:

  Типажи позволяют достичь полиморфизма. Это означает, что вы можете использовать методы из типажа на объектах разных типов, предоставляющих этот типаж, не заботясь о конкретной реализации.

- Реализация общего поведения:

  Вы можете определить общие методы в типаже, а затем реализовать эти методы для разных типов данных. Это позволяет сгруппировать общее поведение в единый интерфейс.

- Расширение функциональности:

  Вы можете реализовать типажи для типов данных, к которым у вас нет доступа (например, сторонние библиотеки), чтобы добавить им функциональность, которой вам не хватает.

Поговорим про типажи…

Типажи как интерфейс:

```
trait Hash {
    fn hash(&self) -> u64;
}
```

```
impl Hash for bool {
    fn hash(&self) -> u64 {
        if *self { 0 } else { 1 }
    }
}

impl Hash for i64 {
    fn hash(&self) -> u64 {
        *self as u64
    }
}
```

В отличие от интерфейсов в таких языках, как Java, C# или Scala, новые **типажи могут быть реализованы для уже существующих типов** (как в случае с `Hash` в последнем примере). То есть абстракции могут быть созданы по необходимости, а затем применены к уже существующим библиотекам.

Поговорим про типажи…

Статическая диспетчеризация

```
fn print_hash<T: Hash>(t: &T) {
    println!("The hash is {}", t.hash())
}
```

```
print_hash(&true);      // instantiates T = bool
print_hash(&12_i64);    // instantiates T = i64
```

- Самый частый способ использования типажей — через использование типового параметризма
- Функция print_hash параметризована неизвестным типом T, но требует, чтобы этот тип реализовал типаж Hash
- **Параметризованные типами функции после компиляции разворачиваются в конкретные реализации**, в результате получаем статическую диспетчеризацию
- Здесь компилятор сгенерирует две копии функции print_hash: по версии для каждого используемого вместо типового аргумента типа. Это означает, что внутренний вызов к t.hash() имеет нулевую стоимость, так как он будет скомпилирован в прямой статический вызов к соответствующей реализации метода hash

Поговорим про типажи…

Динамическая диспетчеризация

```
struct Button {
    listeners: Vec<Box<ClickCallback>>,
    ...
}
```

```
struct Button<T: ClickCallback> {
    listeners: Vec<T>,
    ...
}
```

- Здесь мы используем типаж так, как будто это тип.
- Вообще-то в расте “*типажи — это «безразмерные» типы*”, что примерно означает, что их можно использовать только через указатели, например с помощью Box (указатель на кучу) или & (любой указатель куда угодно).

- &ClickCallback или Box называется **«объект-типаж»** и включает в себя указатель на экземпляр типа T, который реализует заданный типаж (ClickCallback), и указатель на таблицу виртуальных методов с указателями на все методы типажа, реализованные для типа T (в нашем случае только метод on_click)

Полиморфизм. Примеры

Пример статического полиморфизма

```rust
// Принимаем что угодно, реализующее трейт Shape.
fn areas_sum(shape1: impl Shape, shape2: impl Shape) -> f32 {
    shape1.area() + shape2.area()
}

fn foo(rectangle: Rectangle, circle: Circle) {
    // Можем передать две разные фигуры.
    let sum = areas_sum(rectangle, circle);
}
```

Пример динамического полиморфизма

```rust
// Принимаем что угодно, реализующее трейт Shape.
// В этот раз принимаем не сами объекты, а ссылки на них,
// так как не зная конкретный тип объекта, мы не знаем и его размер,
// а следовательно, не сможем выделить для него место на стеке.
fn areas_sum(shape1: &dyn Shape, shape2: &dyn Shape) -> f32 {
    shape1.area() + shape2.area()
}

fn foo(rectangle: Rectangle, circle: Circle) {
    // Можем передать ссылки на две разные фигуры.
    let sum = areas_sum(&rectangle, &circle);
}
```

Обобщим полиморфизм в Rust

- Статический полиморфизм
  - требует, чтобы при компиляции программы было известно, какие конкретные типы используются в каждом обобщённом контексте.
  - Мономорфизацию. (одна обобщённая сущность превращается в несколько сущностей с конкретными типами, используемыми в них.)
  - Размер исполняемого файла увеличивается
  - Высокая скорость выполнения, так как компилятору известны конкретные типы и адреса функций для каждой ситуации
- Динамический полиморфизм
  - работает посредством динамической диспетчеризации
  - Мы не знаем конкретного типа объекта и для получения адреса его методов в памяти используем дополнительную информацию — **таблицу функций**.
  - Исполняемый файл не увеличивается
  - Жертвуем производительностью — для вызова метода нам придётся сначала прочитать его адрес из памяти, что значительно затрудняет оптимизацию программы на этапе компиляции.