Системне Програмування

З використанням мови програмування **Rust.**
Fundamentals. Impl, Traits

Олексій Рихальський

10.10.2024

Impl block

У Rust блок **impl** (скорочено від implementation) використовується для визначення реалізації методів для конкретних типів. Цей блок дозволяє прив'язати функціонал до структур, перерахувань (enum) або інших типів, дозволяючи їм мати свої власні методи. Це ключовий елемент для організації коду в Rust, оскільки він сприяє інкапсуляції та полегшує роботу з різними типами даних.

Impl block. Основний функціонал

1. Реалізація методів
2. Реалізація ассоційованих методів
3. Місце для констант (типа)

Олексій Рихальський

Приклад без impl

```rust
struct Point {
    x: i32,
    y: i32,
}
```

```rust
fn move_point(p: Point, dx: i32, dy: i32) -> Point {
    Point {
        x: p.x + dx,
        y: p.y + dy,
    }
}
```

```rust
#[test]
fn test1() {
    let p = Point { x: 1, y: 2 };
    let p: Point = move_point(p, dx: 10, dy: 20);
    assert_eq!(p, Point { x: 11, y: 22 });
}
```

Приклад з impl (метод)

```rust
struct Point {
    x: i32,
    y: i32,
}
```

```rust
impl Point {
    fn shift(&mut self, dx: i32, dy: i32) {
        self.x += dx;
        self.y += dy;
    }

    fn copy(&self) -> Point {
        self.clone()
    }
}
```

```rust
#[test]
fn test1() {
    let mut p = Point { x: 1, y: 2 };
    let p2 : Point  = p.copy();
    p.shift( dx: 10,  dy: 20);
    assert_eq!(p, Point { x: 11, y: 22 });
    assert_eq!(p2, Point { x: 1, y: 2 });
}
```

Приклад з impl (метод)

```rust
struct Point {
    x: i32,
    y: i32,
}
```

```rust
impl Point {
    fn shift(&mut self, dx: i32, dy: i32) {
        self.x += dx;
        self.y += dy;
    }

    fn copy(&self) -> Point {
        self.clone()
    }
}
```

```rust
#[test]
fn test1() {
    let mut p = Point { x: 1, y: 2 };
    let p2 : Point  = p.copy();
    p.shift( dx: 10,  dy: 20);
    assert_eq!(p, Point { x: 11, y: 22 });
    assert_eq!(p2, Point { x: 1, y: 2 });
}
```

Семантика!

Приклад з impl (ассоційований метод)

```rust
struct Point {
    x: i32,
    y: i32,
}
```

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        Point { x, y }
    }

    fn random() -> Self {
        let mut rt : ThreadRng = rand::thread_rng();
        let x : i32 = rt.gen_range( range: 5.. < 50);
        let y : i32 = rt.gen_range( range: 5.. < 50);
        Point::new(x, y)
    }
}
```

```rust
#[test]
fn test2() {
    let p : Point = Point::new( x: 3,  y: 4);
    assert_eq!(p, Point { x: 3, y: 4 });

    let p : Point = Point::random();
    assert!(p.x >= 5 && p.x < 50);
    assert!(p.y >= 3 && p.x < 100);
}
```

Приклад з impl (ассоційований метод)

```rust
struct Point {
    x: i32,
    y: i32,
}
```

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        Point { x, y }
    }

    fn random() -> Self {
        let mut rt : ThreadRng = rand::thread_rng();
        let x : i32 = rt.gen_range( range: 5.. < 50);
        let y : i32 = rt.gen_range( range: 5.. < 50);
        Point::new(x, y)
    }
}
```

```rust
#[test]
fn test2() {
    let p : Point = Point::new( x: 3,  y: 4);
    assert_eq!(p, Point { x: 3, y: 4 });

    let p : Point = Point::random();
    assert!(p.x >= 5 && p.x < 50);
    assert!(p.y >= 3 && p.x < 100);
}
```

Семантика!

Приклад з impl (константи)

```rust
struct Point {
    x: i32,
    y: i32,
}
```

```rust
impl Point {
    const CENTER: Point = Point { x: 0, y: 0 };
}
```

```rust
#[test]
fn test3() {
    let p: Point = Point::CENTER;
    assert_eq!(p, Point { x: 0, y: 0 });
}
```

Приклад з impl (константи)

```rust
struct Point {
    x: i32,
    y: i32,
}
```

```rust
impl Point {
    const CENTER: Point = Point { x: 0, y: 0 };
}
```

Семантика!

```rust
#[test]
fn test3() {
    let p: Point = Point::CENTER;
    assert_eq!(p, Point { x: 0, y: 0 });
}
```

Блоків impl може бути більше одного

```rust
struct Point {
    x: i32,
    y: i32,
}
```

```rust
impl Point {
    fn shift(&mut self, dx: i32, dy: i32) {
        self.x += dx;
        self.y += dy;
    }

    fn copy(&self) -> Point {
        self.clone()
    }
}
```

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        Point { x, y }
    }

    fn random() -> Self {
        let mut rt : ThreadRng = rand::thread_rng();
        let x : i32 = rt.gen_range( range: 5..50);
        let y : i32 = rt.gen_range( range: 3..100);
        Point::new(x, y)
    }
}
```

```rust
impl Point {
    const CENTER: Point = Point { x: 0, y: 0 };
}
```

Блок impl можна додавати:

- **struct**
- **named tuple**
- **enum**
- **trait**

Висновок

- Блок **impl** у Rust є ключовим інструментом для інкапсуляції та організації функціональності типів.
- Він дозволяє реалізовувати методи для структур та перерахувань, а також додавати поведінку через реалізацію trait.
- Завдяки impl можна визначати як методи екземпляра, що взаємодіють з конкретними об'єктами типу, так і статичні методи, доступні на рівні типу, що сприяє модульності та гнучкості коду.

Висновок

- Блок **impl** дозволяє розділяти реалізацію на кілька частин, зберігаючи читабельність коду навіть у великих проєктах.
- Він підтримує реалізацію асоційованих функцій і констант, що дозволяє створювати фабричні методи або визначати стандартні значення для типів.
- Завдяки своїй гнучкості, блок impl забезпечує високу модульність і робить Rust одним із найбільш потужних і продуктивних мов для системного програмування.

Traits

Олексій Рихальський

10.10.2024

Traits у Rust: ключовий функціонал

Traits у Rust відіграють роль контрактів, які визначають поведінку типів. Вони нагадують інтерфейси в Java або C#, але надають більш гнучкі можливості. За допомогою **trait** можна визначити набір методів, які тип повинен реалізувати, щоб відповідати цьому trait. Це дозволяє встановити стандарти поведінки для типів без прив'язки до конкретних реалізацій, сприяючи модульності та повторному використанню коду.

Traits у Rust

Однією з головних особливостей **trait** є можливість надавати реалізацію за замовчуванням для методів. Це означає, що тип, який реалізує trait, може використовувати ці методи без необхідності їх визначати знову. Однак, якщо потрібно специфічне поведінка, тип має можливість перевизначити такі методи. Це робить traits у Rust дуже гнучкими для проектування розширюваних та багаторазово використовуваних API.

Відмінності Traits у Rust від Interfaces Java/C

Відсутність явної підтримки спадкування: На відміну від інтерфейсів у Java, traits у Rust не підтримують явну ієрархію спадкування, що робить їх більш схожими на протоколи. У Rust можна комбінувати кілька traits разом, але вони не створюють традиційну ієрархію класів.
Дефолтні реалізації: Java інтерфейси почали підтримувати дефолтні методи лише з версії 8, але Rust з самого початку мав потужний механізм для цього. Окрім того, в Java/C зазвичай не можливо змінити поведінку за замовчуванням без рефакторингу коду, тоді як у Rust все є гнучкішим та більше орієнтоване на композицію.

Відмінності Traits у Rust від Interfaces Java/C

Механізм динамічного диспетчеризації: Rust надає як статичну, так і динамічну диспетчеризацію через dyn Trait. Це означає, що можна обирати між швидкістю компіляції та гнучкістю виконання. У Java диспетчеризація зазвичай динамічна, тоді як C++ дозволяє обирати між віртуальними методами (динамічна) та шаблонами (статична).

Відсутність об'єктно-орієнтованого підходу: Traits у Rust не залежать від класів, як інтерфейси в Java та C#. Вони можуть бути реалізовані для будь-якого типу, включаючи примітиви та власні структури даних, що робить їх гнучкішими для розробки більш функціональних або системних рішень.

Traits - багато спільного з концепціями type classes і implicit Scala/Haskell

Як і traits у Rust, type classes у Haskell (та аналог у Scala) дозволяють визначати поведінку, яку повинні реалізувати типи. Ключова ідея полягає в тому, що типи не повинні бути частиною класової ієрархії, щоб реалізовувати методи певної поведінки. Усі типи можуть мати власну реалізацію методів для конкретних type classes (або trait у випадку Rust). Це дозволяє уникати жорсткої прив'язки до класів та сприяє кращій модульності.

Декларація

```rust
trait Animal {
    fn sound(&self) -> String;
}
```

Імплементація

```rust
struct Sheep(String);
struct Cow;

impl Animal for Sheep {
    fn sound(&self) -> String { format!("I'm {:?}, maa-maa", self) }
}

impl Animal for Cow {
    fn sound(&self) -> String { String::from( s: "moo-moo") }
}

impl Animal for i32 {
    fn sound(&self) -> String { format!("I'm i32 and hold value: {self}") }
}
```

Використання (параметр)

```rust
fn notify(item: &impl Animal) {
    let msg : String = item.sound();
    println!("{msg}");
}

#[test]
fn test() {
    notify(&Sheep("York".to_string()));
    notify(&Sheep("Junky".to_string()));
    notify(&Cow);
    notify( item: &42);
}
```

```
I'm Sheep("York"), maa-maa
I'm Sheep("Junky"), maa-maa
moo-moo
I'm i32 and hold value: 42
```

Використання (результат)

```
fn mk_sheep() -> impl Animal {
    Sheep("Gorg".to_owned())
}

fn mk_cow() -> impl Animal {
    Cow {}
}
```

Але...

```
fn mk_random1(value: f64) -> &dyn Animal {
    if value < 0.5 {
        &Sheep
    } else {
        &Cow
    }
}
```

...не працює.

Але...

```
fn mk_random1(value: f64) -> &dyn Animal {
    if value < 0.5 {
        &Sheep
    } else {
        &Cow
    }
}
```

...не працює.
Бо Rust не знає скільки виділити пам'яті

Але...

```rust
fn mk_random2(value: f64) -> Box<dyn Animal> {
    if value < 0.5 {
        Box::new(Sheep)
    } else {
        Box::new(Cow)
    }
}
```

...працює!
Бо Rust знає скільки виділити пам'яті на Box
а що у тому Box - вже інша історія

До імплементації trait можна додавати тільки ті методи яки описані в trait

```
trait Animal {
    fn sound(&self) -> String;
}
```

```
impl Animal for Sheep {
    fn sound(&self) -> String { format!("I'm {:?}, maa-maa", self) }
    fn sound2() {}
}
```

За допомогою trait в масив можна покласти різні елементи

```
let a: [Sheep; 3] = [
    Sheep("York".to_string()),
    Cow,
    43,
];
```

За допомогою trait в масив можна покласти різні елементи

```
let a:[...] = [
    Box::new(Sheep("York".to_string())),
    Box::new(Cow),
    Box::new( x: 43),
];
```

За допомогою trait в масив можна покласти різні елементи

```
let a: [Box<dyn Animal>; 3] = [
    Box::new(Sheep("York".to_string())),
    Box::new(Cow),
    Box::new( x: 43),
];

a.iter() : impl Iterator<Item=&Box<...>>
    .for_each(|x : &Box<dyn Animal> | println!("{}", x.sound()));
```

```
I'm Sheep("York"), maa-maa
moo-moo
I'm i32 and hold value: 43
```

Висновок

Traits у Rust – це гнучка і потужна концепція, яка поєднує кращі риси інтерфейсів, type classes та implicit з інших мов. Вони сприяють створенню модульних, розширюваних та високопродуктивних рішень, дозволяючи розробникам ефективно працювати з різними типами даних, забезпечуючи високу безпеку на рівні компіляції.

Приклад

```
trait Add<Rhs = Self> {
    type Output;
    fn add(self, rhs: Rhs) -> Self::Output;
}
```

Приклад 1

```rust
struct Cart {
    numbers: u8,
    total: f32,
}
```

```rust
impl Add for Cart {
    type Output = Cart;

    fn add(self, rhs: Cart) -> Self::Output {
        Cart {
            numbers: self.numbers + rhs.numbers,
            total: self.total + rhs.total,
        }
    }
}
```

```rust
#[test]
fn code() {
    let c1 = Cart {
        numbers: 3,
        total: 14.99,
    };
    let c2 = Cart {
        numbers: 5,
        total: 24.99,
    };
    assert_eq!(
        c1 + c2,
        Cart {
            numbers: 8,
            total: 39.98
        }
    );
}
```

Приклад 2

```rust
trait Add<Rhs = Self> {
    type Output;
    fn add(self, rhs: Rhs) -> Self::Output;
}
```

```rust
struct Electric;
struct Car;
struct Tesla;
```

```rust
impl Add<Electric> for Car {
    type Output = Tesla;

    fn add(self, rhs: Electric) -> Self::Output {
        Tesla
    }
}
```

```rust
#[test]
fn code() {
    assert_eq!(
        Tesla,
        Car + Electric
    );
}
```

Код з лекцій,
презентації Keynote,
PDF-файли
знаходяться на GitHub:

https://github.com/djnzx/rust-course