Rusting with style - Curso básico de linguagem Rust

Cleuton Sampaio

Veja no GitHub

Menu do curso

VÍDEO DESTA AULA

Orientação a objetos e traits

Sente-se na posição de lótus, feche os olhos e repita comigo: "**Rust** não implementa **orientação a objetos** de forma tradicional." Repita isso umas 1000 vezes.

> **Rust** não implementa **orientação a objetos (OOP)** da maneira tradicional.

Os três pilares da OOP podem ser feitos em Rust, porém de modo diferente:

- **Encapsulamento**: Em Rust, o encapsulamento é feito a nível de módulos, não de classes. Campos públicos podem ser acessados externamente, e campos privados são limitados ao módulo onde são definidos.
- **Herança**: Rust não possui herança de classes como em linguagens OO tradicionais. Em vez disso, você utiliza traits para descrever comportamentos comuns, que podem ser implementados por diversas estruturas.
- **Polimorfismo**: Existe sim em Rust e em duas formas: `static dispatch` e `dynamic dispatch`, com o uso de `traits`.

Um exemplo simples

Vamos àquela famosa estrutura `Animal`, `Cachorro` e `Gato`:

```
trait Animal {
    fn nome(&self) -> String;
    fn fazer_barulho(&self) -> String;
```

```
}

struct Cachorro {
    nome: String,
}

impl Animal for Cachorro {
    fn nome(&self) -> String {
        self.nome.clone()
    }
    fn fazer_barulho(&self) -> String {
        String::from("Au Au")
    }
}

struct Gato {
    nome: String,
}

impl Animal for Gato {
    fn nome(&self) -> String {
        self.nome.clone()
    }
    fn fazer_barulho(&self) -> String {
        String::from("Miau")
    }
}
```

Ok. `Cachorro` e `Gato` são animais, certo? Sim. Mas isso é denotado pela implementação do `trait` **Animal** e não por terem propriedades comuns de animais. Ambos têm a propriedade `nome`, que pode ser obtida com a função `nome` do `trait` **Animal**.

Não temos **herança**, então replicamos as propriedades comuns em cada `struct` concreta e acrescentamos um método no `trait` para obtê-las.

Podemos usar o **polimorfismo** sim e de duas maneiras: Estática e dinâmica.

Polimorfismo estático

Neste caso, o método a ser invocado é decidido em tempo de compilação. Veja o exemplo do projeto **polistatic**:

```
fn fazer_barulho<T: Animal>(animal: &T) {
    println!("Na função: {} diz {}", animal.nome(),
animal.fazer_barulho());
}

fn main() {

    let cachorro = Cachorro {
        nome: String::from("Rex"),
```

```
    };
    let gato = Gato {
        nome: String::from("Miau"),
    };

    println!("{} diz {}", cachorro.nome(), cachorro.fazer_barulho());
    println!("{} diz {}", gato.nome(), gato.fazer_barulho());

    fazer_barulho(&cachorro);
    fazer_barulho(&gato);
}
```

Na função `main` temos variáveis de tipos concretos, então o **polimorfismo** não atua. Porém, temos uma função `fazer_barulho` que recebe uma referência a um **Animal** sem saber qual é o tipo concreto.

Essa função utiliza **generics** (semelhante à **Java** e **Go**) para ser parametrizável. Ela recebe uma referência a um tipo `T`, que é `Animal`. Essa função será **especializada** em tempo de compilação para `Cachorro` e `Gato`.

Polimorfismo dinâmico

É implementado através de **objetos trait** (**trait objects**): Um trait object é uma forma de usar um tipo em Rust que representa "qualquer coisa que implemente certo trait" em tempo de execução, permitindo que você chame métodos definidos no trait sem saber exatamente qual tipo concreto está sendo usado. Isso se dá através de referências ou ponteiros para `dyn Trait`, criando uma camada de indireção que possibilita polimorfismo dinâmico, semelhante a interfaces em outras linguagens.

Vejamos esse mesmo exemplo reescrito:

```
fn fazer_barulho(animal: &dyn Animal) {
    println!("Na função: {} diz {}", animal.nome(),
animal.fazer_barulho());
}

fn main() {

    let animal1: &dyn Animal = &Cachorro {
        nome: String::from("Rex"),
    };
    let animal2: &dyn Animal = &Gato {
        nome: String::from("Sapeca"),
    };

    println!("{} diz {}", animal1.nome(), animal1.fazer_barulho());
    println!("{} diz {}", animal2.nome(), animal2.fazer_barulho());

    fazer_barulho(animal1);
    fazer_barulho(animal2);
}
```

Neste segundo caso, `&dyn Animal` é um **trait object** que armazena em tempo de execução um ponteiro para o dado e um ponteiro para a **vtable** do tipo, permitindo chamadas a métodos virtualmente (dinamicamente). Isso facilita a criação de coleções heterogêneas de objetos que implementam o mesmo trait.

As duas variáveis (`animal1` e `animal2`) são referências a **trait objects** dos tipos `Cachorro` e `Gato`, respectivamente. A função `fazer_barulho` recebe uma referência dinâmica a um **trait object** `Animal`.

Box e coleções polimórficas

`Box` é um tipo inteligente do Rust que permite alocar dados na heap em vez da stack. Ele armazena um ponteiro para o dado real e garante a propriedade e limpeza automática desses dados. Além de ser útil para tamanhos de dados desconhecidos em tempo de compilação, `Box` é comumente usado para criar trait objects e para armazenar estruturas grandes de forma mais flexível.

Você utilizaria um `Box` quando precisa armazenar um objeto que implementa um determinado trait na heap, mantendo a posse dele. Isso é útil em casos como guardar diversos tipos diferentes que implementam o mesmo trait em uma única coleção ou passar objetos entre funções sem copiar seus dados. O `Box<dyn Trait>` permite que você trabalhe com polimorfismo dinâmico, garantindo que o objeto seja gerenciado na memória heap ao invés da stack.

Vejamos como ficaria o nosso exemplo:

```
trait Animal {
    fn nome(&self) -> &str;
    fn fazer_barulho(&self) -> &str;
}

struct Cachorro {
    nome: String,
}

impl Animal for Cachorro {
    fn nome(&self) -> &str {
        &self.nome
    }
    fn fazer_barulho(&self) -> &str {
        "Au Au"
    }
}

struct Gato {
    nome: String,
}

impl Animal for Gato {
    fn nome(&self) -> &str {
        &self.nome
    }
    fn fazer_barulho(&self) -> &str {
        "Miau"
```

```
    }
}

fn main() {
    // Criação de uma coleção heterogênea de objetos implementando Animal
    let animais: Vec<Box<dyn Animal>> = vec![
        Box::new(Cachorro { nome: String::from("Rex") }),
        Box::new(Gato { nome: String::from("Felix") }),
    ];

    // Interagindo com cada objeto da coleção de forma polimórfica
    for animal in animais.iter() {
        println!("{} diz {}", animal.nome(), animal.fazer_barulho());
    }
}
```

Calma! Lembre-se: Respire fundo. Tem muita coisa acontecendo aqui...

Neste exemplo simples usamos `Box` para armazenar vários tipos de animais que implementam o mesmo `trait` **Animal** em um único vetor, demonstrando polimorfismo dinâmico com armazenamento no `heap`.

A coleção **animais** é **polimórfica** pois armazena ponteiros dinâmicos para instâncias de **trait objects**.

Nesse exemplo, tanto `Cachorro` como `Gato` implementam **Animal**, e são armazenados em `Box<dyn Animal>` dentro de um `Vec`, permitindo que tipos diferentes coabitem na mesma coleção e sejam acessados de forma uniforme via o `trait`.

Exemplo mais detalhado

Vamos examinar o **game Ferris Hunter**, que eu fiz em Rust para dar um exemplo disso tudo.