Cleuton Sampaio - Me siga! **RustingCrab.com**.

Trait, box e polimorfismo

Cara, eu costumo dizer que **Go** tem alguns "elefantes" na sala. Mas **Rust** tem: Elefantes, girafas, rinocerontes, Jabutis e muito mais bichos na sala.

Vamos falar de alguns desses "bichos"...

Polimorfismo e inversão de dependências

> **Wikipedia:** Na programação orientada a objetos, o `polimorfismo` permite que referências de tipos de classes mais abstratas representem o comportamento das classes concretas que referenciam. Assim, é possível tratar vários tipos de maneira homogênea (através da interface do tipo mais abstrato).

> **Wikipedia:** o Princípio da `inversão de dependência` refere-se a uma forma específica de desacoplamento de módulos de software que determina a inversão das relações de dependência:

> partem de módulos de alto nível, responsáveis pelas diretrizes, para os de baixo nível. Assim, os módulos de alto nível tornam-se independentes dos detalhes de implementação dos de baixo nível. O princípio estabelece que: Módulos de alto nível não devem incorporar (ou incluir) nada dos módulos de baixo nível. Ambos devem trabalhar apenas com abstrações, ou seja, interfaces. Abstrações não devem depender de detalhes de implementação, mas os detalhes é que devem depender de abstrações.

Juntando esses dois conceitos, podemos ter uma estrutura de Classes, descendendo de uma classe `abstrata` ou implementando uma `interface`, e lidar apenas no nível de **abstração**, sem ter que conhecer a classe concreta dos objetos.

Um exemplo

Vamos imaginar um jogo de **RPG**...

> **RPG**: Role-playing game, também conhecido como RPG ("jogo narrativo", "jogo de interpretação de papéis" ou "jogo de representação"), é um tipo de jogo em que os jogadores assumem papéis de personagens e criam narrativas colaborativamente.

Nesse RPG temos alguns papéis dos objetos **NPC**...

> **NPC**: Um personagem não jogável (em inglês: non-player character, sigla NPC) é um personagem de jogo eletrônico que não pode ser controlado por um jogador. NPCs fazem parte da história e cenário do jogo, podendo o usuário interagir com eles para completar missões, comprar, vender itens ou conhecer sobre a história do ambiente.

Imagine que temos um comportamento comum aos NPCs, mas cada um implementa de uma maneira específica. E digamos que, para conveniência do código, nós queremos instanciar e lidar com eles em uma estrutura única, um Vec por exemplo. Como faríamos isso?

Box

Antes de mais nada, precisamos entender o que seria um ponteiro `Box` em **Rust**. Em Rust, um `Box<T>` é um tipo especial de ponteiro inteligente que guarda valores na heap em vez da stack. Isso ajuda a lidar com dados grandes ou de tamanho desconhecido em tempo de compilação, permitindo que você mova valores entre funções sem cópias grandes, além de facilitar a criação de estruturas de dados recursivas. Ao sair do escopo, o `Box<T>` libera automaticamente a memória que ocupa, mantendo a segurança de memória típica do Rust.

dyn

Em Rust, `dyn` é usado para criar "trait objects", ou seja, tipos que permitem chamadas de métodos de forma dinâmica em tempo de execução, em vez de resolver tudo na compilação. Isso é útil quando você quer flexibilidade para lidar com tipos diferentes que implementam o mesmo trait, sem saber exatamente qual tipo estará em uso até o programa rodar.

Juntando os dois conceitos

Se quisermos instanciar nossos NPCs utilizando `Box` e utilizarmos `dyn` para armazenarmos em um `Vec`, poderemos lidar apenas com a abstração, invocando métodos nela sem nos preocuparmos com qual classe

concreta seria. Isso é o **polimorfismo** em ação.

O código

```rust
trait Personagem {
    fn falar(&self);
}

struct Campones {
    nome: String,
}

impl Campones {
    fn novo(nome: &str) -> Self {
        Campones {
            nome: nome.to_string(),
        }
    }
}

impl Personagem for Campones {
    fn falar(&self) {
        println!("{} diz: Oi!", self.nome);
    }
}

struct Soldado {
    patente: i32,
}

impl Soldado {
    fn novo(patente: i32) -> Self {
        Soldado { patente }
    }
}

impl Personagem for Soldado {
    fn falar(&self) {
        println!("Soldado de patente {} diz: Estou de guarda!",
self.patente);
    }
}

fn main() {
    let personagens: Vec<Box<dyn Personagem>> = vec![
        Box::new(Campones::novo("João")),
        Box::new(Soldado::novo(1)),
    ];

    for personagem in personagens.iter() {
        personagem.falar();
```

```
    }
}
```

Esse código mostra como usar um **trait** em Rust para permitir **polimorfismo** entre diferentes tipos de personagens. Aqui vai uma explicação passo a passo:

1. **Trait (Personagem):**
   Um *trait* é uma coleção de métodos que diferentes tipos podem implementar. No exemplo, o `trait Personagem` define apenas um método, `falar()`. Assim, qualquer tipo que implemente `Personagem` é obrigado a ter uma função `falar()`.

2. **Polimorfismo:**
   Polimorfismo é a capacidade de tratar objetos de tipos diferentes de maneira uniforme, desde que esses tipos implementem o mesmo *trait*. No código, tanto `Campones` quanto `Soldado` implementam `Personagem`. Isso nos permite guardá-los em uma mesma coleção e chamá-los da mesma forma, sem precisar saber qual é o tipo concreto por trás.

3. **Box:**
   O `Box` é um tipo especial que aloca o valor na *heap* ao invés da *stack*. Ao criar um `Box<dyn Personagem>`, nós empacotamos nossa instância concreta (por exemplo, um `Campones` ou um `Soldado`) dentro de uma caixa que guarda o valor na memória dinâmica. Isso é útil quando precisamos armazenar diferentes tipos que implementam o mesmo *trait* em uma única coleção.

4. **dyn (Trait Objects):**
   Ao usar `dyn Personagem`, criamos um *trait object*, ou seja, um objeto de *trait* que suporta *dispatch* dinâmico. Isso significa que, em tempo de execução, o Rust sabe qual método `falar()` chamar, de acordo com o tipo real armazenado no `Box`. Sem o `dyn`, o Rust tentaria resolver tudo em tempo de compilação, o que não funciona facilmente se não soubermos de antemão qual tipo concreto teremos. Com `dyn`, nós ganhamos essa flexibilidade.

O `trait` define o que cada personagem deve saber fazer; o `Box<dyn Personagem>` permite guardar diferentes tipos de personagens na mesma coleção; e o uso de `dyn` é o que torna possível chamar o método `falar()` de cada personagem em tempo de execução, sem precisar saber o tipo exato dele.