Rusting with style - Curso básico de linguagem Rust

Cleuton Sampaio

Veja no GitHub

Menu do curso

VÍDEO DESTA AULA

Mais sobre variáveis

Propriedade e empréstimo

Em **Rust**, os conceitos de propriedade (**ownership**) e empréstimo (**borrow**) são fundamentais para a segurança e a concorrência da memória. Ao entender esses conceitos, você pode aproveitar ao máximo as garantias de segurança do Rust e escrever código eficiente e seguro. Vamos explorar juntos desde os fundamentos até técnicas avançadas, abordando propriedade, empréstimo e suas aplicações práticas em Rust.

Vamos a um pequeno exemplo (que está no código da aula):

```
fn main() {
    let nome = String::from("João");
    let p = nome;
    println!("{}", nome);
    println!("{}", p);
}
```

Tente compilar esse código. Rodou? Não! Deu um erro mais ou menos assim: "borrow of moved value: `nome`", certo? Ok. Mas o que há de errado com esse código? Se fosse em Java (devidamente transcrito) não daria erro algum.

Explicação

Vou explicar de forma bem simples por que o código gera um erro e como corrigir isso.

Imagine que você tem um objeto chamado `nome` que guarda o nome "João". Quando você faz `let p = nome;`, você está basicamente dizendo que `p` agora é responsável por "João". É como se você tivesse um brinquedo que passa de uma criança (`nome`) para outra (`p`).

Depois de passar o brinquedo para `p`, a criança original (`nome`) não tem mais o brinquedo. Portanto, quando você tenta dizer `println!("{}", nome);`, é como se a primeira criança ainda quisesse brincar com o brinquedo que já foi dado para outra. Isso não é permitido, e é por isso que o Rust gera um erro.

Como Corrigir Esse Erro?

Há duas maneiras simples de resolver esse problema:

1. Fazer uma Cópia do Valor
2. Usar uma Referência (empréstimo)

Vamos ver cada uma delas.

1. Fazer uma Cópia do Valor

Se você quer que tanto `nome` quanto `p` tenham "João", você precisa criar uma cópia do valor. Isso significa que ambos terão seu próprio "João" separado.

```
fn main() {
    let nome = String::from("João");
    let p = String::from(&nome); // Cria uma nova String com o mesmo
conteúdo
    println!("{}", nome); // Agora, nome ainda é válido
    println!("{}", p);    // E p também tem "João"
}
```

- **`let p = String::from(&nome);`**: Isso cria uma nova cópia de "João" para `p`. Agora, tanto `nome` quanto `p` têm seu próprio valor, e você pode usá-los separadamente.

2. Usar uma Referência

Outra maneira é **emprestar** o valor sem transferir a responsabilidade. Assim, tanto `nome` quanto `p` podem usar "João" sem que um deixe de ter acesso.

```
fn main() {
    let nome = String::from("João");
    let p = &nome; // p agora "observa" o valor de nome
    println!("{}", nome); // Válido
    println!("{}", p);    // Também válido
}
```

- **`let p = &nome;`**: Aqui, `p` está apenas olhando para o mesmo "João" que `nome` tem. Não está pegando o controle ou mudando nada. Assim, você pode usar tanto `nome` quanto `p` sem problemas.

O que você não pode fazer com empréstimo

Não Pode Modificar a Variável Emprestada Enquanto Ela Está Sendo Emprestada:

```
fn main() {
    let nome = String::from("João");
    let p = &nome; // Empréstimo imutável

    // Tentar modificar `nome` enquanto `p` está emprestando
    nome.push_str(" Silva"); // Erro! Você não pode alterar `nome` enquanto
`p` está referenciando.

    println!("{}", nome);
    println!("{}", p);
}
```

Vai dar um erro na linha onde tenta modificar o conteúdo do `String nome`. Você emprestou o valor para variável `p` e, enquanto o empréstimo estiver válido, não pode modificar a variável que emprestou o conteúdo.

Não Pode Criar um Empréstimo Mutável Enquanto Há Empréstimos Imutáveis Ativos:

```
fn main() {
    let mut nome = String::from("João");
    let p = &nome; // Empréstimo imutável
    let q = &mut nome; // Erro! Não pode ter um empréstimo mutável enquanto
há empréstimos imutáveis ativos

    println!("{}", p);
    println!("{}", q);
}
```

Ele vai reclamar porque você quer criar um empréstimo **mutável** (`let q = &mut nome;`) quando ainda existe um empréstimo **imutável** válido (`let p = &nome;`).

Não Pode Mover a Variável Emprestada Enquanto Ela Está Sendo Emprestada:

```
fn main() {
    let nome = String::from("João");
    let p = &nome; // Empréstimo imutável

    let outro = nome; // Erro! Você está tentando mover `nome` enquanto `p`
ainda está emprestando

    println!("{}", p);
    println!("{}", outro);
}
```

Você não pode transferir a propriedade de `nome` para outra variável (`mover`) enquanto p ainda está emprestando nome.

> `Mover` em **Rust** é como transferir a posse de algo de uma pessoa para outra. Imagine que você tem um brinquedo que é seu. Quando você dá esse brinquedo para um amigo, agora ele é o dono do brinquedo, e você não pode mais usá-lo a menos que o amigo devolva.

Agora você pensa que entendeu

Então me diga se esse código daria erro e qual erro seria:

```
fn main() {
    let v = ["João", "Pedro", "Maria"];
    let x = v[1];
    println!("v {:?}", v);
    println!("x {}", x);
}
```

Não deu erro?! E essa versão agora, daria erro?

```
fn main() {
    let v = [String::from("João"),
             String::from("Pedro"),
             String::from("Maria")];
    let x = v[1];
    println!("v {:?}", v);
    println!("x {}", x);
}
```

Sim, essa daria erro. Você "emprestou" `v[1]` (na verdade, todo o `v`) para `x` e tentou usar `v`, que não tem mais a posse do valor. Mas por que não deu erro na primeira versão?

Na primeira versão, criamos nosso vetor assim:

```
let v = ["João", "Pedro", "Maria"];
```

Quais os tipos de dados dos elementos? Não são do tipo `String`, mas são `&str`.

> Em Rust, `&str` e `String` são dois tipos de representações de strings com diferenças fundamentais: `&str` é uma fatia de string imutável que referencia dados de string existentes, como literais ou partes de uma `String`, sem possuir a propriedade desses dados, o que a torna mais leve e eficiente para leitura; por outro lado, `String` é uma estrutura de dados mutável e alocada na heap que **possui** os dados da string, permitindo modificações como adicionar ou remover caracteres, e gerencia dinamicamente o tamanho da string. Enquanto `&str` é geralmente usado para referências temporárias e passagens eficientes de parâmetros, `String` é utilizado quando você precisa de uma

> string que possa ser alterada ou que precise ser armazenada de forma independente. Essa distinção entre referências imutáveis e propriedades ownership permite que Rust gerencie a memória de forma segura e eficiente.

Essa foi a explicação mais simples. A mais complexa é:

> `&str` implementa o **trait Copy** significando que referências para strings (&str) podem ser copiadas facilmente sem transferir a posse dos dados que estão referenciando.

trait?!

Um **trait** em Rust é como uma lista de habilidades ou ações que diferentes tipos podem ter Imagine que você tem várias ferramentas, como uma furadeira e uma serra. Um trait chamado "Ferramenta" poderia listar ações como "furar" e "cortar". Se uma ferramenta pode **furar** e **cortar**, ela "implementa" o trait "Ferramenta" e pode usar essas ações definidas na lista.

- **Trait** = Interface
- **Tipos que têm essas ações** = Implementam o trait
- Permite que diferentes tipos compartilhem comportamentos comuns

Exemplo Rápido:

```
// Definindo um trait chamado "Desenhavel"
trait Desenhavel {
    fn desenhar(&self);
}

// Implementando o trait para um tipo chamado "Circulo"
struct Circulo {
    raio: f64,
}

impl Desenhavel for Circulo {
    fn desenhar(&self) {
        println!("Desenhando um círculo com raio {}", self.raio);
    }
}

fn main() {
    let c : &dyn Desenhavel = &Circulo { raio: 5.0 };
    c.desenhar(); // Usa a ação definida pelo trait
}
```

O que está acontecendo:

1. **Definição do Trait:** "Desenhavel" lista a ação "desenhar".
2. **Implementação:** O tipo "Circulo" diz que pode "desenhar" seguindo o que o trait exige.
3. **Uso:** Podemos chamar `desenhar` em um "Circulo" porque ele implementa o trait "Desenhavel".
4. **Substituição de Liskov**: É um tipo de "Substituição de Liskov" ou um exemplo de uso de polimorfismo.

> Na linha `let c: &dyn Desenhavel = &Circulo { raio: 5.0 };`, o `&Circulo { raio: 5.0 }` cria uma **referência** para uma nova instância de `Circulo`, ou seja, está "emprestando" o círculo. O `&dyn Desenhavel` indica que `c` é uma referência para **qualquer tipo** que implemente o trait `Desenhavel`. Usando `&dyn Desenhavel`, você permite que `c` chame métodos definidos no trait sem precisar saber exatamente que tipo específico está sendo referenciado. Isso torna o código mais flexível, permitindo que diferentes tipos que implementam `Desenhavel` possam ser usados de forma intercambiável através dessa referência genérica.

Permitir cópia ou clone em outros tipos

Em Rust, **Copy** e **Clone** são duas maneiras de duplicar valores, mas funcionam de formas diferentes. **Copy** permite que tipos simples, como números e referências, sejam copiados automaticamente quando atribuídos a outra variável, sem mover a propriedade. Para permitir isso, você simplesmente adiciona `#[derive(Copy, Clone)]` acima da definição do seu tipo, como uma `struct` ou `enum`. Já **Clone** é usado para criar cópias profundas de tipos mais complexos, onde você precisa explicitamente chamar o método `.clone()`. Para habilitar o Clone, você adiciona `#[derive(Clone)]` ao seu tipo. Por exemplo:

```
#[derive(Copy, Clone)]
struct Ponto {
    x: i32,
    y: i32,
}

#[derive(Clone)]
struct Pessoa {
    nome: String,
    idade: u8,
}
```

Com essas derivadas, `Ponto` pode ser copiado automaticamente, enquanto `Pessoa` pode ser clonada quando necessário. Isso facilita a duplicação de valores de forma segura e eficiente, garantindo que seu código gerencie a memória corretamente sem complicações.

Exemplo prático

No exemplo `blackjack`, da aula de desafio, temos um exemplo de uso de `derive` no arquivo `carta.rs`:

```
#[derive(Clone)]
pub struct Carta {
    pub valor: i32, // No blackjack, o valor das cartas é o mesmo que o número de pontos que ela vale, exceto pelas figuras que valem 10 pontos e o Ás que vale 11 pontos.
    pub naipe: String,
}
```

Por que? Porque no arquivo `jogo.rs` especificamente nas funções jogador_compra e mesa_compra, você está clonando uma carta do baralho antes de adicioná-la à mão do jogador ou do croupier:

```
pub fn jogador_compra(&mut self) -> bool {
    if self.baralho.len() > 0 {
        let indice = rand::thread_rng().gen_range(0..self.baralho.len());
        self.mao_do_jogador.push(self.baralho[indice].clone()); //
**Clonagem ocorre aqui**
        self.baralho.remove(indice);
        return true;
    }
    false
}

pub fn mesa_compra(&mut self) -> bool {
    if self.baralho.len() > 0 {
        let indice = rand::thread_rng().gen_range(0..self.baralho.len());
        self.mao_do_croupier.push(self.baralho[indice].clone()); //
**Clonagem ocorre aqui**
        self.baralho.remove(indice);
        return true;
    }
    false
}
```