Rusting with style - Curso básico de linguagem Rust

Cleuton Sampaio

Veja no GitHub

Menu do curso

VÍDEO DESTA AULA

Borrow check hell

"Borrow Check Hell" é uma expressão utilizada pela comunidade Rust para descrever a frustração que muitos desenvolvedores enfrentam ao lidar com as rigorosas regras do *borrow checker* do Rust. O *borrow checker* é uma parte fundamental do compilador Rust que assegura a segurança de memória, garantindo que não haja referências inválidas ou concorrências inseguras no código. No entanto, essas mesmas regras podem, às vezes, tornar o desenvolvimento mais desafiador, especialmente para quem está começando ou para projetos mais complexos.

Essa "agonia" ocorre principalmente quando o *borrow checker* impede que certas operações sejam realizadas porque elas violam as regras de propriedade e empréstimo do Rust. Por exemplo, tentar modificar uma estrutura de dados enquanto ainda existem referências imutáveis a ela pode resultar em erros que, embora importantes para a segurança, podem ser difíceis de resolver inicialmente. Esses erros costumam ser acompanhados de mensagens de compilação detalhadas, mas que nem sempre são fáceis de entender, aumentando a sensação de "cegueira" para o desenvolvedor.

Para mitigar o "Borrow Check Hell", é essencial entender profundamente como o sistema de propriedade e empréstimo do Rust funciona. Reestruturar o código para reduzir a complexidade das referências, limitar o escopo das variáveis e utilizar ferramentas como `RefCell` ou `Rc` quando apropriado são estratégias eficazes. Além disso, com a prática e a experiência, os desenvolvedores aprendem a antecipar e evitar esses conflitos, tornando o processo de desenvolvimento mais fluido e menos frustrante. Em suma, embora o "Borrow Check Hell" represente um desafio inicial, ele é um reflexo das poderosas garantias de segurança que Rust oferece, contribuindo para a criação de softwares mais robustos e confiáveis.

Vamos explicar os conceitos de **movimento**, **propriedade** e **empréstimo** em Rust de forma simples e resumida, além de abordar como eles se aplicam a tipos primitivos.

Propriedade (Ownership)

- **O que é?**
  - Em Rust, cada valor tem **uma única variável** que "possui" esse valor. Essa variável é responsável por gerenciar a memória desse valor.
- **Por que é importante?**
  - Garante segurança na gestão de memória sem a necessidade de um coletor de lixo (garbage collector).

Movimento (Move)

- **O que é?**
  - Quando você atribui um valor de uma variável para outra, a **propriedade** desse valor é **transferida** para a nova variável. A variável original **não pode mais** ser usada.
- **Exemplo:**

```
let s1 = String::from("Olá");
let s2 = s1; // Movimento: s1 deixa de ser válido
// println!("{}", s1); // Erro! s1 não é mais válido
println!("{}", s2); // Funciona
```

- **Quando ocorre?**
  - Com tipos que **não implementam** o trait `Copy`, como `String` ou `Vec<T>`.

> Em Rust, toda atribuição é semanticamente um "**movimento**". Contudo, para tipos que implementam a trait Copy (como inteiros, bool, char, tipos pontuais como f32, f64, além de alguns compostos que só contenham tipos Copy), esse "**movimento**" funciona, na prática, como uma cópia (bitwise copy).

Empréstimo (Borrowing)

- **O que é?**
  - Permite que múltiplas partes do código acessem um valor sem **tomar a propriedade** dele. Isso é feito através de **referências**.
- **Tipos de Empréstimos:**
  - **Imutável (`&T`)**: Várias referências podem existir simultaneamente.
  - **Mutável (`&mut T`)**: Apenas uma referência mutável pode existir por vez.
- **Exemplo:**

```rust
let s = String::from("Olá");
let r1 = &s; // Empréstimo imutável
let r2 = &s; // Outro empréstimo imutável
println!("{} e {}", r1, r2); // Funciona

let mut s_mut = String::from("Olá");
let r_mut = &mut s_mut; // Empréstimo mutável
r_mut.push_str(" Mundo!");
println!("{}", r_mut); // Funciona
```

> **Atenção:** Cada empréstimo tem uma "janela de vida" bem clara, e o compilador garante que nunca existam acessos concorrentes e/ou alterações simultâneas, mantendo a memória sempre segura. Tentar usar uma variável mutável, com empréstimo mutável dentro da sua "janela de vida", dá erro:

```rust
fn main() {
  // Atenção a essa linha:
  let mut s = String::from("Olá");

  // Empréstimos imutáveis (ok)
  let r1 = &s;

  // Deveria dar erro? Não. Você não está alterando "s".
  println!("Original {}", s); // Não dá erro
  let r2 = &s;
  println!("Imutáveis: {} e {}", r1, r2); // Funciona

  // Agora tentamos criar um empréstimo mutável...
  let r3 = &mut s;
  // E AO MESMO TEMPO usar `s` (ou mesmo as referências imutáveis) na
mesma "janela" de vida.
  println!("Tentando usar s e r3: {} e {}", s, r3);
}
```

Por que não dá erro na linha com o comando: `let mut s = String::from("Olá");`?

> Esse comando cria (aloca) uma nova `String` no heap a partir do literal estático `"Olá"` e atribui esse valor à variável `s`, tornando-a mutável caso queiramos alterá-la depois. Não há "movimento" de outra variável aqui: o literal `"Olá"` (um `&str` imutável em tempo de compilação) é apenas **copiado** para a área recém-alocada de `s`, resultando em um objeto `String` totalmente novo.

Tipos Primitivos e `Copy`

- **O que são tipos primitivos?**
  - Tipos básicos como `i32`, `f64`, `bool`, `char`, etc.
- **Trait `Copy`:**

    - Tipos que implementam o trait `Copy` são **copiados** em vez de movidos. Isso significa que, ao atribuir ou passar esses valores, uma **cópia** é feita e ambas as variáveis continuam válidas.

- **Por que tipos primitivos implementam `Copy`?**
    - Eles são pequenos e simples de copiar, não requerem gerenciamento complexo de memória.
- **Exemplo com `Copy`:**

```
let x = 10;
let y = x; // Cópia: x ainda é válido
println!("x: {}, y: {}", x, y); // Funciona
```

- **Propriedade:** Cada valor tem uma única variável que o possui.
- **Movimento:** Transferência de propriedade para outra variável; a original deixa de ser válida. Ocorre com tipos que **não implementam** `Copy`.
- **Empréstimo:** Acesso a um valor sem tomar a propriedade, usando referências (`&` ou `&mut`).
- **Tipos Primitivos:** Implementam `Copy`, são **copiados** em vez de movidos, permitindo que múltiplas variáveis acessem o mesmo valor sem problemas.

Entender esses conceitos é fundamental para escrever código seguro e eficiente em Rust, aproveitando ao máximo seu sistema de propriedade e gerenciamento de memória.

Regras de propriedade e empréstimo

Resumir de maneira simples as **principais regras de propriedade (ownership)** e **empréstimo (borrowing)** em Rust, acompanhadas de exemplos para facilitar a compreensão.

Regras de Propriedade (Ownership)

1. **Cada valor tem um único proprietário.**
2. **Só pode haver um proprietário de cada vez.**
3. **Quando o proprietário sai de escopo, o valor é descartado (drop).**

1. Cada valor tem um único proprietário

Cada valor em Rust é "possuído" por uma única variável. Essa variável é responsável por gerenciar a memória do valor.

Exemplo:

```
fn main() {
    let s = String::from("Olá, Rust!");
    // Aqui, `s` é o proprietário da String.
}
```

2. Só pode haver um proprietário por vez

Quando você atribui um valor de uma variável para outra, a propriedade é transferida (movida) para a nova variável. A variável original deixa de ser válida.

Exemplo:

```
fn main() {
    let s1 = String::from("Olá");
    let s2 = s1; // Movimento: s1 deixa de ser válido e s2 passa a ser o
proprietário.

    // println!("{}", s1); // Erro! `s1` não é mais válido.
    println!("{}", s2); // Funciona, `s2` é o novo proprietário.
}
```

3. Quando o proprietário sai de escopo, o valor é descartado

Quando a variável que possui um valor sai do escopo (termina sua execução), Rust automaticamente limpa a memória desse valor.

Exemplo:

```
fn main() {
    {
        let s = String::from("Desaparecendo");
        // `s` é válido dentro deste bloco.
    } // `s` sai de escopo aqui e a memória é liberada.

    // println!("{}", s); // Erro! `s` não existe mais.
}
```

Regras de Empréstimo (Borrowing)

1. **Você pode ter múltiplas referências imutáveis ou uma única referência mutável, mas não ambas simultaneamente.**
2. **Referências devem sempre ser válidas.**

1. Múltiplas referências imutáveis ou uma única referência mutável

- **Referências Imutáveis (`&T`):** Permitem ler o valor sem modificá-lo. Você pode ter várias referências imutáveis ao mesmo tempo.

  Exemplo:

  ```
  fn main() {
      let s = String::from("Olá");
  ```

```
    let r1 = &s;
    let r2 = &s;
    let r3 = &s;

    println!("{}, {}, e {}", r1, r2, r3); // Funciona perfeitamente.
}
```

- **Referência Mutável (`&mut T`):** Permite modificar o valor. Apenas uma referência mutável pode existir por vez, e não pode coexistir com referências imutáveis.

  Exemplo:

```
fn main() {
    let mut s = String::from("Olá");

    let r1 = &mut s; // Única referência mutável.

    r1.push_str(", Mundo!");

    println!("{}", r1); // Funciona.
}
```

- **Tentando misturar referências mutáveis e imutáveis:**

  Exemplo com Erro:

```
fn main() {
    let mut s = String::from("Olá");

    let r1 = &s; // Referência imutável.
    let r2 = &mut s; // Erro! Não pode ter referência mutável enquanto
referências imutáveis existem.

    println!("{}, {}", r1, r2);
}
```

  Erro de Compilação:

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed
as immutable
 --> src/main.rs:6:14
  |
4 |     let r1 = &s;
  |              -- immutable borrow occurs here
5 |     let r2 = &mut s;
  |              ^^^^^ mutable borrow occurs here
```

```
 6 |     println!("{}, {}", r1, r2);
   |                         ^^ immutable borrow later used here
```

2. Referências devem sempre ser válidas

As referências devem apontar para valores que ainda estão válidos. Rust garante isso durante a compilação para evitar referências pendentes (dangling references).

Exemplo com Erro:

```rust
fn main() {
    let r;
    {
        let s = String::from("Desaparecendo");
        r = &s;
    } // `s` sai de escopo aqui.
    println!("{}", r); // Erro! `s` não existe mais.
}
```

Erro de Compilação:

```
error[E0597]: `s` does not live long enough
 --> src/main.rs:6:13
  |
5 |         let s = String::from("Desaparecendo");
  |             - `s` declared here
6 |         r = &s;
  |             ^^ borrowed value does not live long enough
7 |     }
  |     - `s` dropped here while still borrowed
8 |     println!("{}", r);
  |                    - borrow later used here
```

Corrigindo o erro:

```rust
fn main() {
    let r;
    {
        let s = String::from("Desaparecendo");
        r = s;
    } // `s` sai de escopo aqui.
    println!("{}", r);
}
```

Agora não dá erro porque estamos **movendo** o string apontado por "s" para "r". Mesmo "s" saindo de escopo, o string continua sendo de propriedade de "r".

Entendendo Copy e Clone

Vamos explorar os **traits `Copy` e `Clone`** em Rust utilizando uma **struct simples**. Explicarei de maneira clara e com exemplos para facilitar o entendimento.

O que são Traits `Copy` e `Clone`?**

Trait `Copy`

- **Definição:**
  - O trait `Copy` permite que tipos implementem uma **cópia por bit** (bitwise copy). Isso significa que, ao atribuir ou passar esses valores, uma cópia completa é feita automaticamente.
- **Características:**
  - Tipos que implementam `Copy` não requerem uma chamada explícita para copiar; a cópia acontece automaticamente.
  - Para um tipo implementar `Copy`, **todos os seus campos também devem implementar `Copy`**.
  - É usado para tipos **simples e de tamanho fixo**, como inteiros (`i32`), `bool`, `char`, etc.

Trait `Clone`

- **Definição:**
  - O trait `Clone` permite criar **cópias explícitas** de valores. Ele define o método `.clone()` que você pode chamar para duplicar um valor.
- **Características:**
  - Pode ser implementado para tipos que precisam de uma cópia profunda ou personalizada.
  - **Não é automático** como `Copy`; você deve chamar `.clone()` quando desejar duplicar o valor.
  - Útil para tipos mais complexos, como `String` e `Vec<T>`, que não implementam `Copy`.

Diferença Entre `Copy` e `Clone`**

- **`Copy`:**
  - Cópia automática e implícita.
  - Requer que todos os componentes também sejam `Copy`.
  - Usado para tipos simples e de baixo custo para copiar.
- **`Clone`:**
  - Cópia explícita e controlada pelo programador.
  - Pode ser implementado para tipos complexos.
  - Permite implementações personalizadas de cópia.

Exemplo Prático com uma Struct Simples

Vamos criar uma struct chamada `Ponto` que representa um ponto 2D com coordenadas `x` e `y`.

Implementando `Copy` e `Clone`

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

fn main() {
    let ponto1 = Ponto { x: 10, y: 20 };

    // Usando `Copy`
    let ponto2 = ponto1; // `ponto1` ainda é válido porque `Ponto`
implementa `Copy`

    println!("ponto1: {:?}", ponto1);
    println!("ponto2: {:?}", ponto2);

    // Usando `Clone`
    let ponto3 = ponto1.clone(); // Cria uma cópia explícita de `ponto1`

    println!("ponto3: {:?}", ponto3);
}
```

Explicação do Código

1. **Derivando `Copy` e `Clone`:**

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

   - Usamos `#[derive(Debug, Copy, Clone)]` para automaticamente implementar os traits `Copy` e `Clone` para a struct `Ponto`.
   - Como `i32` implementa `Copy`, a struct `Ponto` também pode implementar `Copy`.

2. **Usando `Copy`:**

   ```
   let ponto2 = ponto1; // Cópia automática
   ```

   - Atribuição de `ponto1` para `ponto2` cria uma cópia completa de `ponto1`.
   - Ambas as variáveis (`ponto1` e `ponto2`) são válidas e independentes.

3. **Usando `Clone`:**

```
let ponto3 = ponto1.clone(); // Cópia explícita
```

- Chama o método `.clone()` para criar uma cópia de `ponto1`.
- Útil quando você quer deixar claro que está criando uma nova instância.

4. **Imprimindo os Pontos:**

```
println!("ponto1: {:?}", ponto1);
println!("ponto2: {:?}", ponto2);
println!("ponto3: {:?}", ponto3);
```

- Usa o trait `Debug` para imprimir os valores das structs.

Saída do Programa:

```
ponto1: Ponto { x: 10, y: 20 }
ponto2: Ponto { x: 10, y: 20 }
ponto3: Ponto { x: 10, y: 20 }
```

Quando Usar `Copy` ou `Clone`?

Use `Copy` Quando:

- O tipo é pequeno e simples (como inteiros, floats, chars, etc.).
- A cópia é barata e não envolve alocação de memória dinâmica.
- Você deseja que a cópia aconteça automaticamente sem necessidade de chamar `.clone()`.

Use `Clone` Quando:

- O tipo possui dados complexos ou alocação dinâmica (como `String`, `Vec<T>`, etc.).
- Você precisa de uma cópia profunda ou personalizada.
- Deseja ter controle explícito sobre quando a cópia ocorre.

Exemplo com Tipo que Não Implementa `Copy`

Vamos ver o que acontece quando tentamos usar `Copy` com um tipo que não o implementa, como `String`.

```
#[derive(Debug, Clone)]
struct Pessoa {
    nome: String,
    idade: u32,
}

fn main() {
    let pessoa1 = Pessoa {
```

```
        nome: String::from("Alice"),
        idade: 30,
    };

    // let pessoa2 = pessoa1; // ERRO! Movimento: `pessoa1` não é mais
válido
    let pessoa2 = pessoa1.clone(); // Cópia explícita

    println!("pessoa1: {:?}", pessoa1);
    println!("pessoa2: {:?}", pessoa2);
}
```

Explicação:

- **Movimento vs. Clone:**
  - Se tentarmos atribuir `pessoa1` para `pessoa2` sem implementar `Copy`, ocorrerá um **movimento**, e `pessoa1` não poderá mais ser usado.
  - Com `Clone`, podemos criar uma cópia explícita, mantendo `pessoa1` válida.

Saída do Programa:

```
pessoa1: Pessoa { nome: "Alice", idade: 30 }
pessoa2: Pessoa { nome: "Alice", idade: 30 }
```