Rusting with style - Curso básico de linguagem Rust

Cleuton Sampaio

Veja no GitHub

Menu do curso

VÍDEO DESTA AULA

Rc, Refcell e derreferência de ponteiros

Amiga, amigo dev, tem muita coisa ainda para aprender em **Rust** antes de entrarmos na parte mais **saborosa** do curso. Vamos arrematar o assunto agora.

Derreferência

Derreferenciar um ponteiro, de maneira geral, é a operação de "seguir" o endereço de memória armazenado nesse ponteiro para acessar (ler ou escrever) o valor que está efetivamente guardado ali. Em linguagens como C, por exemplo, usamos o operador *ptr para derreferenciar um ponteiro ptr. Isso significa "ir até o endereço em ptr e obter (ou modificar) o conteúdo encontrado naquele local da memória".

```
#include <stdio.h>

int main(void) {
    int x = 42;
    int* p = &x;  // p recebe o endereço de x

    printf("Valor de x: %d\n", x);
    printf("Valor apontado por p: %d\n", *p); // *p faz a derreferência: lê
o valor de x

    // Alterando x via ponteiro
    *p = 100;  // Atribui 100 no lugar onde p aponta (x)

    printf("Novo valor de x: %d\n", x);
```

```
    return 0;
}
```

Em Rust, referências comuns (`&T` ou `&mut T`) **quase sempre** podem ser usadas sem precisar do operador `*`, pois o compilador faz a derreferência de forma automática (por exemplo, ao chamar métodos ou acessar campos). Porém, é possível derreferenciar explicitamente usando `*` caso você queira acessar ou modificar o valor ao qual a referência aponta.

```
// Exemplo com referência imutável
fn main() {
    let x = 10;
    let x_ref = &x;

    // Acesso automático (mesmo sem usar `*`)
    println!("x_ref: {}", x_ref); // Imprime 10

    // Derreferência explícita
    println!("*x_ref: {}", *x_ref); // Imprime 10
}
```

```
// Exemplo com referência mutável
fn main() {
    let mut y = 20;
    let y_ref = &mut y;

    // Precisamos usar `*` para modificar o valor!!!!!
    *y_ref += 1;
    println!("y: {}", y); // Imprime 21
}
```

Ponteiros brutos

Um **ponteiro bruto** (em inglês, *raw pointer*) em Rust é um tipo de ponteiro que não está sujeito às verificações de segurança de memória (ownership, borrow checking) que se aplicam às referências comuns (`&T` e `&mut T`). Eles são declarados como `*const T` ou `*mut T` e podem apontar para qualquer endereço.

Um exemplo para explicar melhor:

```
fn main() {
    let p: *mut String;
    {
        let nome = String::from("Fulano");
        p = &nome as *const String as *mut String;
    }

    // `nome` saiu de escopo, a memória foi liberada!
```

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

O que acha que acontecerá nesse exemplo? Vai compilar? Não! Rust não vai deixar barato! Você tem um `raw pointer` (ponteiro bruto) apontando para uma variável que saiu de escopo, portanto, a área de memória que contém seu valor foi liberada. Para derreferenciar ponteiros brutos, você precisa criar um bloco com `unsafe`, avisando ao compilador que você sabe o que está fazendo:

```
fn main() {
    let p: *mut String;
    {
        let nome = String::from("Fulano");
        p = &nome as *const String as *mut String;
    }
    // `nome` saiu de escopo, a memória foi liberada!
    unsafe {
        // Comportamento indefinido!!!!!!!!
        println!("{}", *p);
    }
}
```

Funcionou? Não! Compilou. Só isso. Ao executar, o valor mostrado na console pela macro `println!` pode ser qualquer coisa, pois a memória foi liberada.

> Uso de `raw pointer` é uma violação das regras do **Rust**? Não chega a ser uma "violação" total dos princípios de Rust, mas sim um **escape hatch** (“escotilha de emergência”). Rust permite o uso de ponteiros brutos em blocos unsafe para cenários onde precisamos de mais controle (por exemplo, FFI ou manipulação de memória em baixo nível). Nesse contexto, o programador assume a responsabilidade de garantir a segurança manualmente, já que o compilador não faz as mesmas verificações que faz para referências seguras (&T e &mut T).

Contagem de referências

`Rc` (Reference Counted) é um tipo especial em Rust que permite várias partes do código compartilharem o mesmo valor sem precisar transferir a propriedade (ownership) toda vez. Ele mantém um contador interno do número de referências, desalocando o valor somente quando o contador chega a zero. Em geral, é usado para compartilhamento **somente de leitura** entre vários “donos” numa única thread (se precisar de thread-safety, use `Arc`). Um exemplo simples:

```
use std::rc::Rc;

fn main() {
    let dado = Rc::new(String::from("Olá, mundo!"));
    let r1 = Rc::clone(&dado);
    let r2 = Rc::clone(&dado);

    println!("dado: {}", dado);
```

```
    println!("r1: {}", r1);
    println!("r2: {}", r2);
    println!("Número de donos: {}", Rc::strong_count(&dado));
}
```

Nesse caso, `dado`, `r1` e `r2` apontam para o **mesmo** conteúdo `"Olá, mundo!"`, e o valor só é desalocado quando nenhum deles existir mais.

> **Atenção:** `Rc` não é **thread safe!**

A `Arc` (*Atomic Reference Counting*) funciona de forma parecida com `Rc`, mas pode ser compartilhada **entre múltiplas threads** com segurança. Segue um exemplo simples:

```
use std::sync::Arc;
use std::thread;

fn main() {
    // Cria um Arc para compartilhar o valor entre threads
    let dado = Arc::new(String::from("Olá, mundo!"));

    // Clonamos o Arc para cada thread (isso não copia o dado,
    // apenas incrementa o contador de referência atômico)
    let r1 = Arc::clone(&dado);
    let r2 = Arc::clone(&dado);

    // Criamos duas threads, cada uma usando seu clone do Arc
    let t1 = thread::spawn(move || {
        println!("Thread 1: {}", r1);
    });

    let t2 = thread::spawn(move || {
        println!("Thread 2: {}", r2);
    });

    // Esperamos as threads terminarem
    t1.join().unwrap();
    t2.join().unwrap();

    // Mostra que o valor original ainda existe aqui
    println!("Main thread: {}", dado);

    // Quando nenhuma referência (Arc) existir mais,
    // o valor será desalocado automaticamente.
}
```

A diferença crucial é que `Arc` usa operações atômicas para manter o contador de referências seguro entre diferentes threads, enquanto `Rc` só funciona em contexto **single-thread**, pois não é seguro compartilhar o contador de referências dele em múltiplas threads.

Mutabilidade com Refcell

`RefCell<T>` é um tipo especial que permite "mutabilidade interna" em Rust, ou seja, possibilita modificar um valor mesmo que tenhamos apenas uma referência imutável a ele. Diferentemente das referências normais, o `RefCell` faz as checagens de empréstimo em **tempo de execução** (em vez de tempo de compilação). Se você tentar ter mais de um empréstimo mutável ou usar um empréstimo imutável ao mesmo tempo que um empréstimo mutável, o programa vai falhar em execução com um "panic" em vez de gerar erro no compilador. Isso é útil em cenários nos quais precisamos contornar as regras estáticas de borrowing, especialmente quando a mutabilidade necessária não é simples de descrever por meio das referências convencionais.

> **Atenção:** Assim como `raw pointers` `RefCell` é algo que precisa ser utilizado com muito cuidado, pois **entorta** as regras do Rust.

Rc + RefCell

Combinar `Rc` (referência contada) com `RefCell` (mutabilidade interna) permite que vários donos acessem e modifiquem o mesmo valor dinamicamente. Eis um exemplo simples:

```
use std::cell::RefCell;
use std::rc::Rc;

fn main() {
    let valor = Rc::new(RefCell::new(10));

    let v1 = Rc::clone(&valor);
    let v2 = Rc::clone(&valor);

    {
        // Borrow mutavelmente usando RefCell
        let mut ref_mut = v1.borrow_mut();
        *ref_mut += 5; // Modifica o valor de 10 para 15
    }

    // Todos veem a mesma mudança, já que compartilham o mesmo RefCell
    println!("v1: {}", v1.borrow());
    println!("v2: {}", v2.borrow());
    println!("valor: {}", valor.borrow());
}
```

No entanto, esse padrão é **desaconselhável** como prática geral porque:

1. Você perde as garantias de segurança em tempo de compilação para empréstimos: o `RefCell` faz essas checagens em **tempo de execução**, podendo resultar em *panic* se houverem conflitos de empréstimo.
2. Fica menos claro para quem lê o código onde e como ocorre a mutação, já que a ideia de "várias referências modificando simultaneamente" vai contra o modelo de *ownership* típico do Rust.
3. Geralmente, usar `Rc<RefCell<T>>` indica que você está forçando um padrão mais "estilo GC" (garbage-collected) dentro do Rust, abrindo mão de parte do controle oferecido pelo *borrow checker* em favor de maior flexibilidade — o que pode ser bom em casos pontuais, mas não deve ser a solução padrão.

Lifetimes explícitos

Lifetimes em Rust servem basicamente para garantir que referências não se tornem inválidas. Quando você usa uma referência dentro de uma `struct`, o compilador precisa saber por quanto tempo esse valor vai existir antes de ser destruído. Um exemplo simples:

```
struct Pessoa<'a> {
    nome: &'a str,
}

fn main() {
    let nome = String::from("Fulano");
    let pessoa = Pessoa { nome: &nome };

    println!("Nome da pessoa: {}", pessoa.nome);
}
```

Aqui, `Pessoa<'a>` indica que a struct guarda uma referência `&'a str`, e `'a` é o **lifetime**: O compilador assegura que o `String` de `nome` continue vivo enquanto a `Pessoa` estiver usando essa referência. Sem deixar isso explícito, o Rust não saberia garantir se `nome` ainda existiria (por exemplo, se fosse destruído antes de usarmos a struct).

Boas práticas

Em Rust, o ideal é deixar o compilador garantir a segurança de memória e o controle de empréstimos.

Quando você usa **raw pointers** (`*const T`/`*mut T`), está abrindo mão dessa segurança em tempo de compilação e assumindo manualmente a responsabilidade de não criar ponteiros inválidos ou acessos concorrentes proibidos.

Já **`Rc<RefCell<T>`** permite múltiplos donos de um valor mutável, mas faz as checagens de empréstimo só em tempo de execução, o que pode levar a *panic* se houver conflito e, além disso, confunde a noção de quem realmente está modificando quem.

Por fim, **lifetimes explícitos** indicam que seu código está exigindo relações de escopo muito intrincadas e que, talvez, haja um design mais simples que aproveite melhor o sistema de *ownership*. Em todos esses casos, você está assumindo mais responsabilidade manual do que o normal, o que pode ser necessário em alguns cenários específicos, mas geralmente é um sinal de que há soluções mais “idiomáticas” em Rust para resolver o problema.