Rusting with style - Curso básico de linguagem Rust

Cleuton Sampaio

Veja no GitHub

Menu do curso

VÍDEO DESTA AULA

Programação funcional

Rust tem uma abordagem simplificada para a programação funcional, baseada em **closures** e **iterators**.

Closure

As **closures** são funções anônimas que você pode salvar em uma variável ou passar como argumentos para outras funções. Você pode criar uma closure em um lugar e então chamá-lo em outro lugar para avaliá-lo em um contexto diferente. Diferentemente das funções, as closures podem capturar valores do escopo no qual elas são definidas. Veja um exemplo:

```
fn main() {
    let delta = |a: f64, b: f64, c: f64| b * b - 4.0 * a * c;
    let x1 = |a: f64, b: f64, c: f64| (-b + delta(a, b, c)) / (2.0 * a);
    let x2 = |a: f64, b: f64, c: f64| (-b - delta(a, b, c)) / (2.0 * a);
    let a = 1.0;
    let b = 3.0;
    let c = 2.0;
    println!("Delta: {}", delta(a, b, c));
    println!("x1: {}, x2: {}", x1(a, b, c), x2(a, b, c));
}
```

Entendeu? Não? Vamos lá... A declaração das **closures** está nesses comandos:

```
let delta = |a: f64, b: f64, c: f64| b * b - 4.0 * a * c;
let x1 = |a: f64, b: f64, c: f64| (-b + delta(a, b, c)) / (2.0 * a);
let x2 = |a: f64, b: f64, c: f64| (-b - delta(a, b, c)) / (2.0 * a);
```

Aqui, declaramos três funções **closure**, ambas recebendo três parametros e retornando um valor. A sintaxe para criar uma **closure** é simples:

```
| <parâmetro> | <corpo>;

| x | x * 10;
```

O tipo dos parâmetros de uma closure pode ou não ser anotado. Isso depende da necessidade. Se houver mais de uma instrução no corpo de comando, então é necessário envolvê-las entre chaves ("{}").

Um exemplo mais completo de closure seria:

```
fn main() {
    let x = 10;
    let y = 5;

    // Definindo uma closure de múltiplas linhas:

    let closure_multilinhas = |a: i32, b: i32| {
        let soma = a + b;
        let produto = a * b;
        soma + produto + x + y // Soma do resultado com as variáveis do
escopo externo
    };

    let resultado = closure_multilinhas(3, 4); // Chamando a closure com os
valores 3 e 4
    println!("O resultado é: {}", resultado);
}
```

Um fator notável nesse exemplo de **closure** é a sensibilidade ao escopo onde a closure está sendo utilizada, pois ela usa os valores de `x` e `y` declaradas externamente. E esta possui anotações de tipos em seus parâmetros e mais de uma linha em seu corpo.

Closure como parâmetro de entrada

Funções e closures são como variáveis. Podemos utilizá-las como qualquer outra variável, inclusive como parâmetro para invocar uma função:

```
//! ```cargo
//! [package]
//! edition = "2021"
```

```
//! ```

fn teste<F : Fn(i32) -> i32>(f: F) -> i32 {
    f(10)
}

fn outro(x: i32) -> i32 {
    x * 100
}

fn main() {
    let x = teste(|y| y + 5);
    println!("{}", x);
    let y = teste(outro);
    println!("{}", y);
}
```

Calma que é forte! Para começar, a função `teste` é genérica, esperando um argumento do tipo `function`, que recebe um `i32` e retorna um `i32`. Ela simplesmente invoca a função que recebe com o argumento `10`.

No primeiro caso, invocamos `teste` com a **closure**: `|y| y + 5`. E ela retorna `15`. No segundo caso, invocamos a mesma função `teste` com a função `outro`, resultando no valor `1000`.

Anotando o tipo da closure com traits

Como você viu no exemplo anterior, quando passamos **closures** como parâmetros é necessário anotar na função o tipo da closure (ou `function`) que estamos esperando:

```
fn teste<F : Fn(i32) -> i32> ...
```

Aqui, anotei o tipo da closure com o `trait Fn`, que especifica uma função, neste caso recebendo `i32` e retornando `i32`. Isso é necessário pois as **closures** geralmente não trazem esse tipo de anotação. Isso poderia ser reescrito com o `trait where`:

```
fn teste<F>(f: F) -> i32
where
    F: Fn(i32) -> i32,
{
    f(10)
}
```

Assim como o `Fn`, o `where` serve para delimitar o domínio das **closures** e **functions** que podem ser passadas para a função `teste`.

Outro exemplo:

```rust
struct Calculadora {
    valor: i32,
}

impl Calculadora {
    fn new(valor: i32) -> Calculadora {
        Calculadora { valor }
    }

    fn aplicar<F>(&self, operacao: F) -> i32
    where
        F: Fn(i32) -> i32,
    {
        operacao(self.valor)
    }
}

fn main() {
    let calc = Calculadora::new(10);

    let adicionar = |x| x + 5;
    let multiplicar = |x| x * 2;

    println!("Resultado da adição: {}", calc.aplicar(adicionar));
    println!("Resultado da multiplicação: {}", calc.aplicar(multiplicar));
}
```

Existem mais `traits` que podemos utilizar como: `FnMut`, and `FnOnce`, mas não falaremos neste curso sobre eles.

Generics

Genérics servem para generalizar de tipos e funcionalidades para casos mais amplos. Serve para reduzir a duplicação de código de muitas maneiras. Utilizamos parâmetros de tipo para especificar os tipos que podem ser utilizados.

Um parâmetro de tipo é declarado com colchetes angulares e inicial maiúscula. Parâmetros de tipo são normalmente representados como . Em Rust, "genérico" também descreve qualquer coisa que aceite um ou mais parâmetros de tipo genérico . Qualquer tipo especificado como um parâmetro de tipo genérico é genérico, e todo o resto é concreto (não genérico).

```rust
fn main() {
    // Função genérica que retorna o maior valor entre dois elementos
    fn maior<T: PartialOrd>(a: T, b: T) -> T {
        if a > b {
            a
        } else {
            b
        }
    }
```

```
    let num1 = 10;
    let num2 = 20;
    println!("O maior número é: {}", maior(num1, num2));

    let char1 = 'a';
    let char2 = 'b';
    println!("O maior caractere é: {}", maior(char1, char2));
}
```

Como pode ver, podemos declarar **funções** dentro de **funções**, e também podemos delimitar os tipos possíveis que a função genérica pode aceitar. Neste caso, qualquer tipo passado deve ser **ordenável**, implementando o `trait PartialOrd`. No exemplo utilizamos duas variáveis `i32` e duas `char` e a função genérica conseguiu saber qual era o maior.

Expressões funcionais

O grande **tchan** da programação funcional são as expressões sem efeitos colaterais, muito popularizadas em **Java** com a **Stream API**. Em Rust podemos fazer isso também. Temos várias funções que podem ser utilizadas para criar expressões funcionais:

- **iter**: Cria um iterador sobre referências aos elementos de uma coleção. É útil quando você deseja iterar sobre os elementos sem consumir a coleção.

  ```
  let iter = numeros.iter();
  ```

- **filter**: Filtra os elementos de um iterador com base em um predicado. Retorna um novo iterador contendo apenas os elementos que satisfazem a condição especificada.
- **map**: Aplica uma função a cada elemento de um iterador e retorna um novo iterador contendo os resultados. É útil para transformar os elementos de uma coleção.
- **collect**: Consome um iterador e coleta os elementos em uma coleção, como um vetor, uma string ou um hashmap. É frequentemente usado para converter iteradores em coleções.
- **fold**: Reduz um iterador a um único valor, aplicando uma função acumuladora a cada elemento, começando com um valor inicial.
- **for_each**: Aplica uma função a cada elemento de um iterador, mas não retorna um novo iterador. É útil para executar tarefas como imprimir valores.

```
fn main() {
    // Uso de closures com funções de alta ordem
    let numeros = vec![1, 2, 3, 4, 5];
    let numeros_dobrados: Vec<i32> = numeros.iter().map(|&x| x *
2).collect();
    println!("Números dobrados: {:?}", numeros_dobrados);
```

```
    // Uso de closures com filter
    let numeros_pares: Vec<i32> = numeros.iter().filter(|x| *x % 2 ==
0).map(|&x| x).collect();
    // Também funcionaria assim:
    // let numeros_pares: Vec<i32> = numeros.iter().filter(|x| *x % 2 ==
0).copied().collect();
    println!("Números pares: {:?}", numeros_pares);
}
```

Neste exemplo temos o uso de `iter`, `map`, `filter` e `collect`. Na primeira expressão, utilizamos o `map` para transformar cada valor da entrada, gerando um novo `Vec` com os valores dobrados. NO segundo caso, usamos o `filter` para selecionar os valores e o `map` para transformá-los em cópias e gerar um novo `Vec`.

Reduzindo a um só valor

O `fold` é muito interessante pois permite reduzir uma coleção a um só valor. Isso é muito útil quando queremos calcular algo sobre um vetor. Vamos fazer isso com o **desvio padrão amostral**:

```
fn main() {
    let dados = vec![1.0, 2.0, 3.0, 4.0, 5.0];
    let n = dados.len() as f64;

    let media = dados.iter().sum::<f64>() / n;

    let desvio_padrao = (dados.iter().fold(0.0, |acc, &x| acc + (x -
media).powi(2)) / (n - 1.0)).sqrt();

    println!("Desvio padrão amostral: {}", desvio_padrao);
}
```

Primeiro, calculamos a média dos valores, necessária para calcular o desvio padrão. Depois, usamos o `fold` para reduzir a coleção a um só valor. O `fold` recebe dois parâmetros:

- O valor inicial para o acumulador de valores
- Uma **closure** que acumule os valores

Neste caso, estamos acumulando a diferença entre cada valor e a média, já dividindo. Para maior acurácia, seria melhor calcular a variância e depois fechar o cálculo, mas eu quis dar um exemplo completo.

For-each

Aqui está um exemplo simples de `for_each`:

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    numbers.iter().for_each(|&x| {
        println!("O número é: {}", x);
```

```
    });
}
```