Usando struct para estruturar dados relacionados

Diogo Silveira Mendonça

Slides Baseados no Capítulo 5 do Rust Book
https://doc.rust-lang.org/book/ch05-00-structs.html

Introdução

- Será visto neste capítulo:
  - Structs ou Estruturas.
  - Exemplo de programa utilizando Structs.
  - Métodos.

- Os códigos dos exemplo podem ser encontrados nesse link.

Definindo e Instanciando Structs

- Podemos definir uma estrutura utilizando struct.

- Struct é semelhante a tupla.
  - podem armazenar diferentes tipos.
  - A diferença é que em uma struct, podemos nomear cada dado.

- Struct são mais flexíveis e fáceis de utilizar.
  - Não dependemos da ordem dos dados para especificar ou acessar os valores de uma instância.

Exemplo:

```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}
```

Definindo e Instanciando Structs

- Depois de definir a estrutura, podemos utilizar, criando uma instância dela e especificando os valores de cada campo.
- Os valores da estrutura, não precisam estar na mesma ordem da definição.
- Podemos acessar um valor específico, com NomeDaInstância.NomeCampo
- Se a instância for mutável, poderemos modificar os valores dos campos.

Exemplo:

```
fn main() {
        let mut user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
        };

user1.email = String::from("anotheremail@example.com");
}
```

Definindo e Instanciando Structs

- Podemos definir uma função para criar a instância para da estrutura.
- Nota que a função recebe como parâmetro o email e o username.
- No exemplo 1, repetimos username e email, essa repetição pode ser simplificada, como mostra o exemplo 2.
- Se o parâmetro tiver o mesmo nome do campo, podemos fazer a simplificação.

Exemplo 1:

```
fn build_user(email: String, username: String) -> User {
        User {
                active: true,
                username: username,
                email: email,
                sign_in_count: 1,
        }
}
```

Exemplo 2:

```
fn build_user(email: String, username: String) -> User {
        User {
                active: true,
                username,
                email,
                sign_in_count: 1,
        }
}
```

Definindo e Instanciando Structs

- Se quisermos criar uma nova instância user2, com os mesmo valores de user1, menos o email.
- O exemplo 1, pega os valores de user1 de forma explícita.
- No exemplo 2, é definido o email diferente de user1, depois ..user1 passa o restante dos campos.

```
let user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};
```

Exemplo 1:

```
let user2 = User {
    active: user1.active,
    username: user1.username,
    email: String::from("another@example.com"),
    sign_in_count: user1.sign_in_count,
};
```

Exemplo 2:

```
let user2 = User {
    email: String::from("another@example.com"),
    ..user1
};
```

Definindo e Instanciando Structs

- Se tentarmos acessar o username de user1, teremos um erro.
- Esse erro se dá por conta que o dado não está mais disponível para user1, porque foi movido para user2.
- Se utilizarmos username: user1.username.clone(), poderemos acessar sem problemas.
- Não precisamos clonar active e sign_in_count, pois eles por default já são clonados.

```
let user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};
```

Exemplo 1:

```
let user2 = User {
    active: user1.active,
    username: user1.username,
    email: String::from("another@example.com"),
    sign_in_count: user1.sign_in_count,
};
```

Exemplo 2:

```
let user2 = User {
    email: String::from("another@example.com"),
    ..user1
};
```

Structs-Tuplas sem Campos Nomeados para Criar Tipos Diferentes

- Rust suporta estruturas parecidas com Tuplas, conhecida como tuple struct.
- Podemos definir com "struct nome(tipos dos dados)"
- São úteis, para nomear a tupla como um todo, diferenciando das outras tuplas.

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);
}
```

Unit-like struct sem campos

- Unit-like são estruturas sem campo.
- São chamadas de unit-like, por conta da similaridade da tupla especial chamada unit.
- Podem ser úteis quando você precisa implementar um trait em algum tipo, mas não possui dados que deseja armazenar no próprio tipo.
- Será abordado mais para frente, no capítulo 10.

```
struct AlwaysEqual;

fn main() {
    let subject = AlwaysEqual;
}
```

Exemplo de um programa usando Structs

- Esse é um exemplo de um programa que calcula a área de um retângulo.
- Vamos começar com variáveis únicas e ir modificando para encaixar as estruturas.
- cargo new rectangles
- cargo run
- Retorna 1500 square pixels.
- Link do código.

```
fn main() {
    let width1 = 30;
    let height1 = 50;

    println!(
        "The area of the rectangle is {} square pixels.",
        area(width1, height1)
    );
}

fn area(width: u32, height: u32) -> u32 {
    width * height
}
```

Refatorando com tuplas

- Agora a função area recebe um unico parametro, sendo uma tupla.
- rect1 armazena a altura e largura.
- Porém agora precisamos lembrar que o index 0 é a largura e 1 a altura.
- Isso pode dificultar o entendimento do código, deixando mais suscetível a erros.
- Link do código.

```
fn main() {
        let rect1 = (30, 50);

        println!(
        "The area of the rectangle is {} square pixels.",
        area(rect1)
        );
}

fn area(dimensions: (u32, u32)) -> u32 {
        dimensions.0 * dimensions.1
}
```

Refatorando com Estruturas

- Definimos a estrutura Rectangle, com os campos width e height.
- Na main, instanciamos rect1 com width = 30 e height = 50.
- A função area agora recebe como parâmetro &Rectangle.
- A função area pega emprestado rect1 e devolve para main.
- Link do código.

```
struct Rectangle {
        width: u32,
        height: u32,
}

fn main() {
        let rect1 = Rectangle {
                width: 30,
                height: 50,
        };

        println!(
        "The area of the rectangle is {} square pixels.",
        area(&rect1)
        );
}

fn area(rectangle: &Rectangle) -> u32 {
        rectangle.width * rectangle.height
}
```

Adicionando funcionalidade úteis com Características derivadas

- Vamos adicionar a funcionalidade de imprimir as dimensões do retângulo.
- Ao compilar esse código, recebemos: error[E0277]: `Rectangle` doesn't implement `std::fmt::Display`
- Isso é causado por conta que println! macro possui algumas formatações.
- Por default println! usa display. Já que para tipos primitivos, só tem uma maneira de imprimir.
- Link do código.

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!("rect1 is {}", rect1);
}
```

Adicionando funcionalidade úteis com Características derivadas

- Para estruturas println! tem diversas formas de imprimir.
  - Com ;
  - Com {}
  - Mostrando todos os campos

- Rust não tenta adivinhar a forma que deve mostrar.

- Continuando lendo o erro, temos:

```
= help: the trait `std::fmt::Display` is not implemented for `Rectangle`
= note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print) instead
```

Adicionando funcionalidade úteis com Características derivadas

- Usando println!("rect1 is {:?}", rect1); como recomendado, temos que {:?} indica para utilizar o formato debug. Permitindo imprimir a estrutura.

- Porém iremos receber esse erro:

```
error[E0277]: `Rectangle` doesn't implement `Debug`
  = help: the trait `Debug` is not implemented for `Rectangle`
  = note: add `#[derive(Debug)]` to `Rectangle` or manually `impl Debug for Rectangle`
```

Adicionando funcionalidade úteis com Características derivadas

- Adicionando #[derive(Debug)] como sugerido.
- Agora conseguimos compilar e ter como resultado:

  rect1 is Rectangle { width: 30, height: 50 }
- Se mudarmos {:?} para {:#?}, teremos como resultado:

```
rect1 is Rectangle {
        width: 30,
        height: 50,
}
```

Exemplo:

```
#[derive(Debug)]
struct Rectangle {
        width: u32,
        height: u32,
}

fn main() {
        let rect1 = Rectangle {
                width: 30,
                height: 50,
        };

        println!("rect1 is {:?}", rect1);
}
```

Adicionando funcionalidade úteis com Características derivadas

- dbg! é uma macro, que pega ownership de uma expressão, diferentemente de println! que utiliza uma referência.

- dbg! imprime o número da linha que a chamada ocorre e o resultado da expressão.

- Usando dbg!, em vez de println!, temos como resultado:

```
[src/main.rs:10] 30 * scale = 60
[src/main.rs:14] &rect1 = Rectangle {
width: 60,
height: 50,
}
```

```
#[derive(Debug)]
struct Rectangle {
        width: u32,
        height: u32,
}

fn main() {
        let scale = 2;
        let rect1 = Rectangle {
                width: dbg!(30 * scale),
                height: 50,
        };

        dbg!(&rect1);
}
```

Sintaxe do Método

- Métodos são semelhantes a funções, definimos com fn e um nome, pode ter parâmetros e um valor de retorno.
- Métodos são definidos no contexto de uma struct (ou enum ou trait object)
- Seu primeiro parâmetro é sempre self. Que representa a instância da struct na qual o método está sendo chamado.
- Podemos chamar o método, utilizando rect1.area().
- Link do código.

```
#[derive(Debug)]
struct Rectangle {
        width: u32,
        height: u32,
}

impl Rectangle {
        fn area(&self) -> u32 {
                self.width * self.height
        }
}

fn main() {
        let rect1 = Rectangle {
                width: 30,
                height: 50,
        };

        println!(
                "The area of the rectangle is {} square pixels.",
                rect1.area()
        );
}
```

Sintaxe do Método

- Podemos nomear o método, com o mesmo nome do campo.
- Se utilizarmos rect1.width, será retornado o valor do campo.
- Se utilizarmos rect1.width(), será retornado o valor do método.

```
impl Rectangle {
        fn width(&self) -> bool {
                self.width > 0
        }
}

fn main() {
        let rect1 = Rectangle {
                width: 30,
                height: 50,
        };

        if rect1.width() {
                println!("The rectangle has a nonzero width;
                it is {}", rect1.width);

        }
}
```

Métodos com diversos parâmetros

- Podemos adicionar mais de um método ao bloco impl.
- Um método pode ter diversos parâmetros, desde que o primeiro seja self. Como mostrado no método can_hold.
- Link do código.

```
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    let rect2 = Rectangle {
        width: 10,
        height: 40,
    };
    let rect3 = Rectangle {
        width: 60,
        height: 45,
    };

    println!(
        "Can rect1 hold rect2? {}",
         rect1.can_hold(&rect2)
    );
    println!(
        "Can rect1 hold rect3? {}",
        rect1.can_hold(&rect3)
    );
}
```

Múltiplos blocos de impl

- Podemos adicionar diversos métodos em um único bloco de impl, como no exemplo 1.
- Também podemos separar os métodos em diversos blocos de impl, como no exemplo 2.

Exemplo 1:

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

Exemplo 2:

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

Funções associadas que não são métodos

- Funções associadas, são as funções que são definidas dentro de um bloco de impl.
- Seu primeiro parâmetro é diferente de self, portanto não é um método.
- Geralmente chamadas para criar novas instâncias do tipo (new)
- Similar a “métodos estáticos”
- Veremos mais sobre elas quando estudarmos módulos

```
#[derive(Debug)]
struct Rectangle {
        width: u32,
        height: u32,
}

impl Rectangle {
        fn square(size: u32) -> Self {
                Self {
                                width: size,
                                height: size,
                }
        }
}

fn main() {
        let sq = Rectangle::square(3);
}
```

Referências

- https://doc.rust-lang.org/book/ch05-00-structs.html