Rusting with style - Curso básico de linguagem Rust

Cleuton Sampaio

Veja no GitHub

Menu do curso

VÍDEO DESTA AULA

Match e outras coisas

A lição passada foi *punk* não? Bom, espero que essa seja mais suave para você...

Esta lição tem `exercícios`!* Procure no repo a pasta `exercicio`, leia e faça o exercício. A pasta `correcao` tem a correção deles. Clone o repositório para ficar mais fácil.

Como temos muitos scripts na pasta `codigo`, e não são projetos completos, vou te ensinar uma maneira fácil de rodá-los sem usar **myCompiler** e sem ficar criando projetos:

```
cargo install cargo-script
...
cargo script <nome do script>.rs
```

Você já ouviu falar em **pattern matching** (correspondência de padrões)? Em Rust, o `match` é uma construção de controle de fluxo poderosa que permite comparar um valor contra uma série de padrões e executar código com base no primeiro padrão que corresponder. É semelhante a um `switch` em outras linguagens, mas com muito mais flexibilidade e segurança.

Vamos a um exemplo:

```
#[allow(dead_code)]
enum Dia {
    Segunda,
    Terca,
```

```
    Quarta,
    Quinta,
    Sexta,
    Sabado,
    Domingo,
}

fn verificar_dia(dia: Dia) {
    match dia {
        Dia::Sabado | Dia::Domingo => {
            println!("É fim de semana!");
        },
        Dia::Segunda | Dia::Terca | Dia::Quarta | Dia::Quinta | Dia::Sexta
=> {
            println!("É um dia útil.");
        },
    }
}

fn main() {
    verificar_dia(Dia::Terca);
}
```

> `#[allow(dead_code)]` serve para desabilitar os avisos de código morto, ou seja, código que não foi usado. Esse exemplo vai encher o saco dizendo que os outros dias do `enum` não foram utilizados. Com isso, desabilitamos esse aviso. Só para esse exemplo! Não faça isso em produção.

Vamos explicar o exemplo... Para começar, temos um `enum` que é exatamente como em qualquer outra linguagem. E fazemos um `match` em uma variável desse tipo, buscando por dias específicos. Se der `match` em **Sábado OU Domingo** ele avisa que é fim de semana, do contrário, avisa que é dia útil.

Note que estamos usando um só pipe ("|") em vez dos costumeiros dois pipes para OR ("||"). Por que? Porque em pattern matching é assim. Em condicionais usamos dois pipes.

Caso a variável que estamos utilizando no `match` seja caractere ou numérica, podemos utilizar intervalos (como em um `for`):

```
    match idade {
        0 => println!("Recém nascida!"),
        1 | 2 | 3 => println!("Bebê"),
        4..=12 => println!("Criança"),
        13..=19 => println!("Adolescente"),
        _ => println!("Adulta"),
    }
```

Mas por enquanto só intervalos "inclusive" são permitidos.

Desestruturação

O `match` serve também para **desestruturar** arrays, slices, tupas, structs, ponteiros e referências.

> Em Rust, "desestruturar" (ou fazer "destructuring") refere-se ao ato de decompor um tipo composto em suas partes constituintes, geralmente por meio de padrões de correspondência (pattern matching). Isso permite extrair valores internos de tuplas, structs, enums, ou referências a tais tipos, de forma clara e direta, sem precisar acessar cada campo manualmente. Em outras palavras, ao desestruturar um objeto, o desenvolvedor pode simultaneamente validar a forma esperada daquele dado e, ao mesmo tempo, obter acesso imediato aos valores internos, tornando o código mais simples, expressivo e seguro.

```
    // arrays
    let v = [1, 4, 6];
    match v {
        [0, 4, 6] => println!("0,4,6!"),
        [1, x, y] => println!("1, {}, {}", x, y), // Vai mostrar esse!
        [2, _, _] => println!("Começa com 2"),
        _ => println!("Nenhum"),
    }

    // tuplas
    let t = (8, 12, 20);
        match t {
        (8, 12, 6) => println!("8,12,6!"),
        (_, 12, z) => println!("Tem 12 no segundo e o terceiro é {}!",z),
// Esse!
        _ => println!("Nenhum"),
    }
```

E podemos desestruturar structs:

```
    struct Funcionario {
        matricula: u32,
        idade: u8,
    }
    let f = Funcionario{matricula: 1007, idade: 32};
    match f {
        Funcionario{matricula: 1007, idade: z} => println!("Fulano tem {}
anos", z),
        _ => println!("Não é"),
    }
```

Só que não funciona com literais string!

guards

Um "guard" em um `match` é uma condição booleana opcional adicionada após um padrão que, se avaliada como verdadeira, faz com que aquele braço do `match` seja escolhido. Caso o padrão corresponda, mas o guard resulte em falso, o Rust seguirá tentando corresponder os padrões seguintes. Isso é útil para filtrar casos mais específicos sem precisar criar novos padrões complexos. A sintaxe geral é:

```
match valor {
    padrao if condicao => { /* código se padrao corresponder e condicao for
verdadeira */ },
    _ => { /* outros casos */ },
}
```

Exemplo simples:

```rust
fn main() {
    let numero = 5;

    match numero {
        x if x % 2 == 0 => println!("O número {} é par", x),
        x => println!("O número {} é ímpar", x),
    }
}
```

Nesse exemplo, o primeiro braço do match verifica se `numero` é par por meio do guard `if x % 2 == 0`. Se for verdade, executa esse braço; caso contrário, passa para o próximo, que imprime que o número é ímpar.

Option e Result

Option:
Imagine que você quer pegar um valor que pode ou não existir, como o resultado de uma busca em uma lista. Em Rust, o tipo `Option<T>` é uma forma segura de representar algo que pode ou não ter um valor do tipo `T`. Um `Option<T>` pode ser `Some(valor)`, se o valor existe, ou `None`, se não existe. Assim, antes de usar o valor interno, você verifica se é `Some` ou `None`, prevenindo erros de acessar algo inexistente.

Result:
O tipo `Result<T, E>` é usado para lidar com operações que podem dar certo ou falhar. Se a operação for bem-sucedida, você obtém um `Ok(valor)`, onde `valor` é o resultado esperado. Se algo der errado, você obtém um `Err(erro)`, informando o tipo de erro ocorrido. Assim, antes de usar o resultado, você verifica se é `Ok` ou `Err`, tratando os erros de forma clara e controlada.

Exemplos:

````
**Exemplo de Option:**
```rust
fn main() {
    let numeros = vec![10, 20, 30];
    // Tentando acessar o índice 3, que não existe
    let valor = numeros.get(3);

    match valor {
        Some(v) => println!("O valor no índice 3 é {}", v),
        None => println!("Não existe valor nesse índice"),
````
````

```
    }
}
```

Nesse exemplo, `get(3)` retorna um `Option<&i32>`. Como o índice não existe, retorna `None`, e tratamos esse caso de forma segura sem causar um erro.

Exemplo de Result:

```
fn main() {
    let resultado = "42".parse::<i32>(); // parse pode falhar se a string
não for um número

    match resultado {
        Ok(numero) => println!("Número convertido com sucesso: {}",
numero),
        Err(erro) => println!("Falha ao converter: {}", erro),
    }
}
```

Aqui, `parse` retorna um `Result<i32, _>`. Se der certo, obtemos `Ok(numero)`. Caso contrário, `Err(erro)`, permitindo tratar o problema.

Let com match combinados

Para tratar erros usando `let` e `match`, normalmente você associa a variável resultante da operação propensa a falhas a um padrão que extrai o valor em caso de sucesso (`Ok`) ou lida com a falha (`Err`). Isso é especialmente útil com o tipo `Result`, que é retornado por muitas funções que podem falhar. Segue um exemplo simples:

```
fn main() {
    // Tentamos converter uma string para um número inteiro
    let resultado = "42".parse::<i32>();

    let numero = match resultado {
        Ok(valor) => valor,                 // Se der certo, extraímos o
valor
        Err(erro) => {                     // Se der errado, tratamos o
erro
            println!("Erro ao converter: {}", erro);
            return;                         // Aqui poderíamos encerrar o
programa ou tomar outra ação
        }
    };

    println!("Número convertido: {}", numero);
}
```

Neste exemplo:

- A variável `resultado` é um `Result<i32, _>`.
- Ao fazer `let numero = match resultado { ... }`, utilizamos um `match` para verificar se `resultado` é `Ok(valor)` ou `Err(erro)`.
- Em caso de sucesso, `numero` recebe o valor inteiro convertido.
- Em caso de erro, tratamos o erro (imprimimos uma mensagem) e interrompemos a execução, sem atribuir nenhum valor a `numero`.

Dessa forma, usamos `let` em conjunto com `match` para extrair valores com segurança de um `Result` e tratar adequadamente eventuais erros.

Tratamento de erros

No Rust, o tratamento de erros é feito de forma explícita, usando principalmente o tipo `Result`. Em vez de lançar exceções, funções que podem falhar retornam um `Result<T, E>`, onde `T` é o tipo esperado em caso de sucesso e `E` é o tipo de erro.

A ideia é:

- Ao chamar uma função que retorna `Result`, você precisa lidar com o caso de sucesso (`Ok(...)`) e o caso de erro (`Err(...)`).
- Isso pode ser feito com `match`, `if let` ou usando o operador `?` para "propagar" o erro para o chamador. O operador `?` é bem útil porque, se o resultado for `Ok`, ele extrai o valor; se for `Err`, ele faz a função atual retornar imediatamente esse erro, sem precisar escrever código extra.

Exemplo sem `?`:

```
fn ler_numero() -> Result<i32, std::num::ParseIntError> {
    let texto = "42";
    let numero = texto.parse::<i32>();
    match numero {
        Ok(n) => Ok(n),
        Err(e) => Err(e),
    }
}
```

Exemplo com `?`:
A mesma lógica acima pode ser simplificada usando `?`:

```
fn ler_numero() -> Result<i32, std::num::ParseIntError> {
    let texto = "42";
    let numero = texto.parse::<i32>()?; // Se falhar, volta Err
automaticamente
    Ok(numero)
}
```

Desse modo, o erro é tratado diretamente no tipo de retorno da função. Quem chamar `ler_numero()` também receberá um `Result` e decidirá o que fazer em caso de falha, mantendo o código mais simples e claro.