Cleuton Sampaio - Me siga!

RustingCrab.com.

Path no GitHub

Option, Result, "unwrap", "expect", "?"

Rust tem um sistema de tratamento de exceções bem diferente das outras linguagens e, para entendê-lo, é preciso estudar um pouco.

Exceção

O que é uma exceção? Pode ser um erro ou pode ser uma condição não satisfeita, como um valor vir "nulo" quando deveria vir preenchido. Algumas linguagens utilizam **SEH** ou "Structured Exception Handling" para lidar com exeções e os comandos mais comuns são o "***try/catch**".

Rust funciona de maneira diferente e, para isto, é preciso entender **enum**.

Agebraic Data Type e Enum

Uma das coisas interessantes de Rust são as **enums**. As enums do Rust são associadas ao conceito de "Tipos de Dados Algebricamente" (**ADT** - Algebraic Data Types) porque elas permitem modelar dados de maneira flexível, expressiva e estrutural, utilizando a combinação de dois conceitos fundamentais: tipos soma e tipos produto.

Eis um exemplo da característica dos **enums** em Rust que nos permite adicionar propriedades às suas variantes.

```
enum FormaPagamento {
    AVista,
    Parcelado { numero_parcelas: i64 },
}
```

Existem duas enums especiais em Rust: `Option<T>` e `Result<T, E>`. Eis seus valores:

```
enum Option<T> {
    Some(T), // Contém um valor do tipo T
    None,    // Representa a ausência de um valor
}

enum Result<T, E> {
    Ok(T),  // Representa sucesso e contém o valor do tipo T
    Err(E), // Representa falha e contém o erro do tipo E
}
```

Podemos usar `Option` e `Result` para declarar variáveis, parâmetros e retorno de funções:

```
fn calcular_parcela(montante: i32, numero_parcelas: Option<i32>) ->
Option<i32> {
    if let Some(parcelas) = numero_parcelas {
        if parcelas == 0 {
            None // Divisão por zero não é possível
        } else {
            Some(montante / parcelas)
        }
    } else {
        None // numero_parcelas é None
    }
}

fn main() {
    let montante = 100;
    let numero_parcelas = Some(0);
    let parcela = calcular_parcela(montante, numero_parcelas);
    match parcela {
        Some(valor) => println!("Valor da parcela: {}", valor),
```

```
        None => println!("Pagamento à vista"),
    }

    // Várias outras maneiras de lidar com um resultado "Option":

    if let Some(valor) = calcular_parcela(montante, numero_parcelas) {
        println!("Valor da parcela: {}", valor);
    } else {
        println!("Pagamento à vista");
    }

    let parcela = calcular_parcela(montante, numero_parcelas).unwrap_or(0);
    println!("Valor da parcela: {}", parcela);

    let parcela = calcular_parcela(montante,
numero_parcelas).unwrap_or_else(|| 0);
    println!("Valor da parcela: {}", parcela);

    let parcela = calcular_parcela(montante, numero_parcelas).expect("Erro
ao calcular a parcela"); // Vai dar "panic"

    // let parcela = calcular_parcela(montante, numero_parcelas).unwrap();
// Vai dar "panic"

    //let parcela = calcular_parcela(montante, numero_parcelas)? // Vai
repassar o problema para cima. Neste caso vai dar erro.


}
```

Para usar essa função precisamos "desembrulhar" o resultado "Option" o que pode ser feito de algumas maneiras:

- Atribuindo e depois utilizando `match`;
- Utilizando `if-let`;
- Utilizando `unwrap-or` ou `unwrap_or_else` com uma `closure` em caso de valor inexistente;
- Utilizando só o `unwrap` ou o `expect`, que darão **Panic** em caso de valor inexistente;
- Repassar para quem chamou, o que daria erro no caso da função `main`;

Você só pode acessar o valor se `desembrulhar` o `Option`.

O **Result** utilizamos para casos de erro. Vamos ver um exemplo:

```
/// Representa um Retângulo com base e altura positivas.
struct Retangulo {
    base: u32,
    altura: u32,
}

impl Retangulo {
    /// Cria um novo retângulo, garantindo que base e altura sejam
```

```
positivas.
    /// Retorna `Ok(Retangulo)` se as invariantes forem respeitadas, ou
`Err(String)` caso contrário.
    fn novo(base: u32, altura: u32) -> Result<Retangulo, String> {
        if base == 0 {
            return Err("A base deve ser maior que zero.".to_string());
        }
        if altura == 0 {
            return Err("A altura deve ser maior que zero.".to_string());
        }
        Ok(Retangulo { base, altura })
    }

    /// Calcula a área do retângulo.
    fn area(&self) -> u32 {
        self.base * self.altura
    }
}

/// Uma função que tenta criar um retângulo e propaga o erro para quem
chamou.
fn criar_retangulo(base: u32, altura: u32) -> Result<Retangulo, String> {
    // Propaga qualquer erro de `Retangulo::novo` para cima usando `?`
    let retangulo = Retangulo::novo(base, altura)?;
    Ok(retangulo)
}

fn main() -> Result<(), String> {

    // Tentativa de criar um retângulo válido e usar o operador `?` para
propagar erros
    let retangulo = criar_retangulo(5, 10)?;
    println!("Retângulo criado com sucesso: {} x {}", retangulo.base,
retangulo.altura);
    println!("Área do retângulo: {}", retangulo.area());

    // Tentativa de criar um retângulo inválido sem verificar o erro:

    let retangulo_errado = criar_retangulo(0, 10);
    println!("Retângulo criado, mas na verdade contém um erro!");

    // Tentativa de criar um retângulo inválido testando com if-let:
    if let Err(erro) = criar_retangulo(0, 10) {
        println!("Erro ao criar retângulo: {}", erro);
    } else {
        println!("Retângulo criado com sucesso!");
    }

    // Tentativa de criar um retangulo inválido testando com match:
    match criar_retangulo(0, 10) {
        Ok(retangulo) => println!("Retângulo criado com sucesso: {} x {}",
retangulo.base, retangulo.altura),
        Err(erro) => println!("Erro ao criar retângulo: {}", erro),
    }
```

```
    // Tentativa de criar um retângulo inválido (propaga o erro
automaticamente com `?`)
    let retangulo_errado = criar_retangulo(0, 10)?;
    println!("Retângulo criado: {} x {}", retangulo_errado.base,
retangulo_errado.altura);

    Ok(())
}
```

A implementação da função `novo` na struct `Retangulo` retorna `Result<Retangulo, String>`, ou seja, retorna uma instância de `Retangulo` ou uma mensagem String, caso um dos parâmetros esteja zerado.

A função `criar_retangulo` cria uma instância e retorna ou retorna um erro. O comando `Ok(retangulo)` vai retornar o novo retângulo ou propagar o erro para quem invocou essa função, que é a `main`.

Se a base ou a altura forem zero, é uma situação de erro pois uma **invariante** foi violada.

> Uma **invariante** é uma condição ou regra que deve ser sempre verdadeira para garantir a consistência de uma estrutura, sistema ou algoritmo. Por exemplo, em um retângulo, a base e a altura devem ser positivas. Invariantes ajudam a manter o programa correto, evitando estados inválidos ou inconsistentes.

Devemos tomar uma atitude caso dê algum erro. Se você nada fizer, o **Rust** não dará erro, a não ser que tente utilizar um retângulo inválido. Isso é diferente de lançar exceções em outras linguagens de programação.

Vamos ver como podemos tratar isso:

- Utilizando "?" para propagar o erro na `main` (main retorna: `Result<(), String>`);
- Utilizando `match` para testar o erro;
- Utilizando `if-let` para testar o erro;

Resumindo

Se uma variável pode ser nula, ou seja, vir sem valor, então `Option<T>` é a melhor opção de tipo de dado para ela. E precisaremos **desembrulhar** a variável para obter o seu valor (ou None). Agora, se o valor de algo puder violar uma **invariante**, então devemos embrulhar em um `Result<T,E>` para possibilitar o teste. E podemos propagar para quem chamou, em vez de dar **Panic**.