Enums e padrões de Matching

Diogo Silveira Mendonça

Slides Baseados no Capítulo 6 do Rust Book
https://doc.rust-lang.org/book/ch06-00-enums.html

Introdução

- Neste capítulo vamos conhecer as enumerações, conhecido também como enums.

- Enums permitem que você defina um tipo enumerando suas possíveis variantes.
  - Vamos verificar sua definição e uso.
  - Além de um tipo especial de enum que é o Option, que serve para dizer se o valor é algo ou se é nulo.

- Conhecer alguns padrões da expressão match.

- Vamos também conhecer o controle de fluxo conciso com if let.

- Os exemplos de código podem ser encontrados nesse link.

Definindo um Enum

- Podemos definir utilizando “enum nomeDoEnum {}”
- Nesse exemplo, temos as versões 4 e 6, para o tipo de endereço IP.
- Nesse caso, utilizar uma enumeração é mais apropriado que uma estrutura. Já que só pode ser um ou outro.
- IpAddrKind agora é um tipo de dado customizado.

```
enum IpAddrKind {
    V4,
    V6,
}
```

Valor do enum

- Podemos criar instâncias do enum utilizando:

```
let four = IpAddrKind::V4;
let six = IpAddrKind::V6;
```

- Os dois valores são do tipo IpAddrKind.

```
enum IpAddrKind {
    V4,
    V6,
}

fn main() {
    let four = IpAddrKind::V4;
    let six = IpAddrKind::V6;

    route(IpAddrKind::V4);
    route(IpAddrKind::V6);
}

fn route(ip_kind: IpAddrKind) {}
```

Valor do enum

- Esse código mostra como o ipAddr seria, caso fosse escrito como uma estrutura.
- O campo kind é do tipo IpAddrKind, definido no enum.
- O valor IpAddrKind::V4 está associado a “127.0.0.1”
- O valor IpAddrKind::V6 está associado a “::1”
- Link do código.

```
fn main() {
    enum IpAddrKind {
        V4,
        V6,
    }

    struct IpAddr {
        kind: IpAddrKind,
        address: String,
    }

    let home = IpAddr {
        kind: IpAddrKind::V4,
        address: String::from("127.0.0.1"),
    };

    let loopback = IpAddr {
        kind: IpAddrKind::V6,
        address: String::from("::1"),
    };
}
```

Valor do enum

- Aqui temos o mesmo formato da estrutura, porém utilizando enum.
- O código fica mais resumido.

```
fn main() {
        enum IpAddr {
                V4(String),
                V6(String),
        }

        let home = IpAddr::V4(String::from("127.0.0.1"));

        let loopback = IpAddr::V6(String::from("::1"));
}
```

Valor do enum

- Outra vantagem é a possibilidade de utilizar tipos diferentes de dados para V4 e V6.
- Isso não seria possível em um estrutura.

```
fn main() {
        enum IpAddr {
                V4(u8, u8, u8, u8),
                V6(String),
        }

        let home = IpAddr::V4(127, 0, 0, 1);

        let loopback = IpAddr::V6(String::from("::1"));
}
```

Valor do enum

- Quit não tem um dado associado.
- Move tem o nome dos campos, assim como uma estrutura.
- Write tem uma única string.
- ChangeColor tem 3 i32.

Exemplo:

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

Valor do enum

- Esses structs contém os mesmos dados do emun mostrado anteriormente.

```
struct QuitMessage; // unit struct
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String); // tuple struct
struct ChangeColorMessage(i32, i32, i32); // tuple struct
```

Valor do enum

- Assim como estruturas, podemos chamar métodos no enum.
- No exemplo temos o método call.

```
fn main() {
    enum Message {
        Quit,
        Move { x: i32, y: i32 },
        Write(String),
        ChangeColor(i32, i32, i32),
    }

    impl Message {
        fn call(&self) {
            // method body would be defined here
        }
    }

    let m = Message::Write(String::from("hello"));
    m.call();
}
```

A opção enum e suas vantagens sobre valores nulos

- No rust não existe valores nulos.
- No rust existe enum que representam os valores estarem presentes ou ausentes.
- O enum que faz esse papel é o enum:

```
enum Option<T> {
    None,
    Some(T),
}
```

- Esse enum é tão útil que está presente no prelúdio, então você pode utilizar Some e None, sem o prefixo Option::
- A sintaxe <T> é um tipo genérico de parâmetro, que será visto no capítulo 10.

A opção enum e suas vantagens sobre valores nulos

- <T> significa que a variante Some da enumeração Option pode armazenar uma peça de dados de qualquer tipo.
- Cada tipo concreto usado em vez de T torna o tipo geral Option<T> um tipo diferente.
- Exemplo de uso:

```
let some_number = Some(5);
let some_char = Some('e');

let absent_number: Option<i32> = None;
```

- some_number é um Option<i32> e some_char é um Option<char>
- Esses dois são tipos diferentes que o Rust consegue inferir porque foram expecificados em Some.

A opção enum e suas vantagens sobre valores nulos

- Para absent_number, o Rust nos exige anotar o tipo geral Option, pois o compilador não consegue inferir o tipo que a correspondente variante Some terá, olhando apenas para um valor None. Aqui, dizemos ao Rust que pretendemos que absent_number seja do tipo Option<i32>.

- Quando temos um valor em Some, quer dizer que o valor está presente e ele está contido dentro de Some.

- Quando temos um valor em None, quer dizer que o valor é nulo.

A opção enum e suas vantagens sobre valores nulos

- Ter um Option<T> é melhor que ter null, por conta que Option<T> e T são diferentes tipos.
- O compilador não nos permitirá usar um valor Option<T> como se fosse definitivamente um valor válido.

```
fn main() {
    let x: i8 = 5;
    let y: Option<i8> = Some(5);

    let sum = x + y;
}
```

- No caso de exemplo, o compilador não permite somar i8 com Option<i8>, pois são diferentes tipos
- Isso previne um dos erros mais comuns que é achar que um valor não é nulo, mas na verdade ele é nulo.

Operador match de controle de fluxo

- Permite comparar um valor com uma série de padrões e, em seguida, executar código com base no padrão correspondente.

- Os padrões podem ser compostos por:
  - Valores literais
  - Nomes de variáveis
  - Curingas
  - Entre outros.

- Veremos um exemplo do uso do match, para uma máquina que identifica o valor de uma moeda.

Operador match de controle de fluxo

- Temos o enum Coin, com a classificação das moedas.
- Temos match listando as moedas e seus valores.
- O match pode ter diversos braços:
  Coin::Penny => 1,
- O braço do match tem duas partes:
  - Um padrão que é o valor Coin::Penny
  - O código a ser executado. Nesse caso retorna o valor da moeda.
  - O operador => separa o padrão e o código a ser executado.
- O match vai comparar o valor com o padrão, caso seja válido, o código é executado, se não for, ele compara com o próximo padrão.

Exemplo:

```
enum Coin {
        Penny,
        Nickel,
        Dime,
        Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
        match coin {
                Coin::Penny => 1,
                Coin::Nickel => 5,
                Coin::Dime => 10,
                Coin::Quarter => 25,
        }
}
```

Operador match de controle de fluxo

- Podemos utilizar chaves para delimitar o escopo do código.

- Normalmente não utilizamos chaves, quando o código a ser executado é pequeno.

```
fn value_in_cents(coin: Coin) -> u8 {
        match coin {
                Coin::Penny => {
                        println!("Lucky penny!");
                        1
                }
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
        }
}
```

Padrões atrelados a valores

- Durante os anos de 1999 e 2008 era cunhado Quarter com design diferente para os 50 estados.
- Podemos adicionar essa informação à nossa enumeração alterando a variante Quarter para incluir um valor UsState armazenado dentro dela

Exemplo:

```
#[derive(Debug)] // so we can inspect the
                 state in a minute

enum UsState {
    Alabama,
    Alaska,
    // --snip–
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
```

Padrões atrelados a valores

- Na chamada value_in_cents(Coin::Quarter(UsState::Alaska)), coin seria Coin::Quarter(UsState::Alaska)
- Nenhum padrão é válido, até chegar a Coin::Quarter(state)
- Nesse ponto state seria UsState::Alaska
- Que é utilizado no println!, tendo como resultado “State quarter from Alaska!”

```
fn value_in_cents(coin: Coin) -> u8 {
        match coin {
                Coin::Penny => 1,
                Coin::Nickel => 5,
                Coin::Dime => 10,
                Coin::Quarter(state) => {
                        println!("State quarter
                        from {:?}!", state);
                        25
                }
        }
}

fn main() {

value_in_cents(Coin::Quarter(UsState::
Alaska));
}
```

Matching com Option<T>

- Podemos lidar com Option<T> usando match

- A forma que a função match continua funcionando da mesma forma.

- A função recebe um Option<i32>
  - Se for Some, adiciona 1 a esse valor.
  - Se for None, a função deve retornar o valor None e não realizar nenhuma operação.

- O i se vincula ao valor contido em Some, então i recebe o valor 5. Então Some(5) corresponde a Some(i).

Exemplo:

```
fn main() {
        fn plus_one(x: Option<i32>) -> Option<i32> {
                match x {
                        None => None,
                        Some(i) => Some(i + 1),
                }
        }

        let five = Some(5);
        let six = plus_one(five);
        let none = plus_one(None);
}
```

Matching é exaustivo

- Se o match não cobrir o valor None, o compilador irá retornar um erro:

  error[E0004]: non-exhaustive patterns: `None` not covered

- O Rust sabe que não cobrimos todos os casos e indica qual caso ficou faltando.

- O match do Rust é exaustivo, ou seja, precisamos abranger todas as possibilidades para que o código seja válido.

Exemplo:

```
fn main() {
    fn plus_one(x: Option<i32>) -> Option<i32> {
        match x {
            None => None,
            Some(i) => Some(i + 1),
        }
    }

    let five = Some(5);
    let six = plus_one(five);
    let none = plus_one(None);
}
```

Padrões Abrangentes e o Espaço Reservado _

- No exemplo ao lado, temos um rolamento de dado. Caso o valor seja:
  - 3 a pessoa ganha um chapéu.
  - 7 a pessoa perde o chapéu
  - Outro valor a pessoa se move.

- Apesar de não cobrimos todos os valores de forma explícita o padrão other, se aplica a todos os outros valores diferentes de 3 e 7.

- Se deixássemos o other antes do 3 e 7, o valor nunca seria comparado com o padrão 3 e 7, por conta que other seria válido.

```
fn main() {
    let dice_roll = 9;
    match dice_roll {
        3 => add_fancy_hat(),
        7 => remove_fancy_hat(),
        other => move_player(other),
    }

    fn add_fancy_hat() {}
    fn remove_fancy_hat() {}
    fn move_player(num_spaces: u8) {}
}
```

Padrões Abrangentes e o Espaço Reservado _

- Caso não queiramos utilizar o valor do padrão, podemo utilizar o “_” em vez de other.
- “_” corresponde a qualquer valor, mas não se vincula a esse valor.
- Nesse exemplo mudamos a regras do jogo. Em vez do jogador se mover, ele irá jogar o dado novamente.
- A parte do código de execução não precisa do valor do dado, então utilizaremos “_”.

```
fn main() {
    let dice_roll = 9;
    match dice_roll {
        3 => add_fancy_hat(),
        7 => remove_fancy_hat(),
        _ => reroll(),
    }

    fn add_fancy_hat() {}
    fn remove_fancy_hat() {}
    fn reroll() {}
}
```

Padrões Abrangentes e o Espaço Reservado _

- Mudando novamente as regras do jogo. Agora em vez de jogar novamente os dados, nada acontece.

- Podemos expressar isso usando o valor unitário () que é uma tupla vazia.

```
fn main() {
    let dice_roll = 9;
    match dice_roll {
        3 => add_fancy_hat(),
        7 => remove_fancy_hat(),
        _ => (),
    }

    fn add_fancy_hat() {}
    fn remove_fancy_hat() {}
}
```

Controle de Fluxo Conciso com if let

- A sintaxe if let, permite combinar if e let de uma maneira menos verbosa para lidar com valores que correspondem a um padrão, ignorando os demais.
- Temos no exemplo 1, a utilização do match.
- No exemplo 2, temos o uso do if let.
- A sintaxe if let recebe um padrão e uma expressão separados por “=”.
- O if let não é executado, se o valor não corresponder ao padrão.
- Como if let, você perde a verificação exaustiva.

Exemplo 1:

```
let config_max = Some(3u8);
match config_max {
    Some(max) => println!("The
    maximum is configured to be {}", max),
    _ => (),
}
```

Exemplo 2:

```
let config_max = Some(3u8);
if let Some(max) = config_max {
    println!("The maximum is
    configured to be {}", max);
}
```

Controle de Fluxo Conciso com if let

- O exemplo 1 mostra a utilização do "_" no match, para os outros valores diferentes de Quarter.
- Podemos utilizar um else no if let, como no exemplo 2. Tendo o mesmo resultado do exemplo 1.

Exemplo 1:

```
let mut count = 0;
match coin {
        Coin::Quarter(state) =>
println!("Statequarter from {:?}!", state),
        _ => count += 1,
}
```

Exemplo 2:

```
let mut count = 0;
if let Coin::Quarter(state) = coin {
        println!("State quarter from {:?}!", state);
} else {
        count += 1;
}
```

Referências

Capítulo 6 - Rustbook - https://doc.rust-lang.org/book/ch06-00-enums.html