Visão geral de Rust

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Este trabalho está licenciado com uma Licença Creative Commons - Atribuição-CompartilhaIgual 4.0 Internacional.
http://github.com/malbarbo/na-lp-copl

Conteúdo

Primeiros passos

Rust

Empowering everyone to build reliable and efficient software.

GET STARTED

Version 1.34.0

The Rust 2018 Edition is here!

Why Rust?

Performance

Rust is blazingly fast and memory-efficient: with no runtime or garbage collector, it can power performance-critical services, run on embedded devices, and easily integrate with other languages.

Reliability

Rust's rich type system and ownership model guarantee memory-safety and thread-safety — and enable you to eliminate many classes of bugs at compile-time.

Productivity

Rust has great documentation, a friendly compiler with useful error messages, and top-notch tooling — an integrated package manager and build tool, smart multi-editor support with auto-completion and type inspections, an auto-formatter, and more.

Principais referências

- Página do Rust
- Livro “The Rust Programming Language”
- FAQ
- API
- Reddit
- StackOverflow

Olá mundo

```rust
fn main() {
    println!("Olá mundo!");
}
```

Compilação

```
> rustc ola.rs
```

Execução

```
> ./arquivo
Olá mundo!
```

Olá mundo 2.0

```rust
use std::env;
fn main() {
    if let Some(nome) = env::args().skip(1).next() {
        println!("Olá {}!", nome);
    } else {
        println!("Olá desconhecido!");
    }
}
```

Execução

```
> ./ola2
Olá desconhecido!
> ./ola2 João
Olá João!
```

`cargo`

- Além do compilador, uma instalação Rust vem com o gerenciador de pacote e sistema de construção `cargo`

cargo

- Criação de um projeto

```
> cargo new exemplo
     Created binary (application) `exemplo` package
> cd exemplo
> tree
.
+-- Cargo.toml
+-- src
    |-- main.rs
1 directory, 2 files
> cat src/main.rs
fn main() {
    println!("Hello, world!");
}
```

cargo

- Compilação

```
# gera o arquivo target/debug/exemplo
> cargo build
# execução
> target/debug/exemplo
Hello world
```

- Compilação e execução direta

```
> cargo run
# informações de compilação ...
Hello world
```

- Por padrão o comando `cargo build` e `cargo run` geram binários menos eficiente e com informações para depuração
- A opção `--release` gera um binário mais eficiente mas demora mais tempo para compilar

```
# gera o arquivo target/release/exemplo
> cargo build --release
# execução
> target/release/exemplo
Hello world
```

Organização

- Vamos deixar o arquivo `src/main.rs` apenas com a função principal
- O restante do código vamos escrever no arquivo `src/lib.rs`
- Para projeto maiores, outros módulos podem ser criados em `src/`

Testes

- Cada função deve vir acompanhada de um testes automatizado
- Conteúdo do arquivo `src/lib.rs`

```
pub fn fat(n: u32) -> u32 {
    if n < 2 {
        1
    } else {
        n * fat(n - 1)
    }
}
```

Testes

- Continuação do arquivo `src/lib.rs`

```rust
#[cfg(test)]
mod tests {
    use super::fat;

    #[test]
    fn fat_test() {
        assert_eq!(1, fat(0));
        assert_eq!(1, fat(1));
        assert_eq!(2, fat(2));
        assert_eq!(6, fat(3));
    }
}
```

Testes

- Execução dos testes

```
> cargo test
...

running 1 test
test tests::fat_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; ...
```

Usando funções de src/lib.rs em src/main.rs

- Conteúdo do arquivo src/main.rs

```rust
use exemplo::fat;
use std::env;

fn main() {
    if let Some(n) = env::args().skip(1).next() {
        if let Ok(n) = n.parse() {
            println!("O fatorial de {} é {}", n, fat(n));
        } else {
            println!("Número inválido: {}", n);
        }
    } else {
        println!("Forneça um número...")
    }
}
```

Usando funções de `src/lib.rs` em `src/main.rs`

- Executando o programa e passando parâmetros

```
> cargo run
Forneça um número...
> cargo run -- abc
Número inválido: abc
> cargo run -- 10
O fatorial de 10 é 3628800
```

- O programa também pode ser executado diretamente

```
> target/debug/exemplo 10
O fatorial de 10 é 3628800
```

O básico

O básico

- Tipos primitivos
- Constantes e variáveis
- Estruturas de controle
- Funções

Tipos primitivos

Tipos primitivos

- Inteiros sem sinal
  - `u8`, `u16`, `u32`, `u64`, `u128`, `usize`
- Inteiros com sinal
  - `i8`, `i16`, `i32`, `i64`, `i128`, `isize`
- O tamanho de `usize` e `isize` é a quantidade de bytes necessários para referenciar qualquer local da memória. Devem ser usados para valores relacionados a memória, como índices de arranjos e operações com ponteiros.

Tipos primitivos

- Pontos flutuante
  - `f32` (+/- 7 casas de precisão)
  - `f64` (+/- 15 casas de precisão)

Tipos primitivos

- Literais numéricos poder ter um sufixo que indica seu tipo
  - `10i8`
  - `10u32`
  - `10f32`
  - `10f64`
- Em geral, os sufixos não são necessários pois os tipos são inferidos pelo contexto

Tipos primitivos

- Booleano (1 byte)
  - `bool` (valores `true` e `false`)
- Caractere (4 bytes)
  - `char` (valor escalar Unicode)

Operadores

- Numéricos
  - `+`, `-`, `*`, `/`, `%`
- Relacionais
  - `==`, `!=`, `>`, `>=`, `<`, `<=`
- Lógicos
  - `||`, `&&`, `!`
- Bit a bit
  - `|`, `&`, `^`, `<<`, `>>`
  - `!` (inverte os bits)

Operadores

- Atribuição
  - `=`, `+=`, `*=`, etc
- Não tem os operadores de incremento e decremento (++ e −−)

Variáveis e constantes

Constantes

- Exemplos

```
const PI: f64 = 3.14;
const MSG_ALERTA: &'static str = "Esqueci qual era!";
const NOME_PROG: &str = "ola";
const NOTAS: [u64; 6] = [2, 5, 10, 20, 50, 100];
const MOEDAS: &[u64] = &[1, 5, 10, 25, 50, 100];
```

- O tipo é sempre necessário
- O tempo de vida é opcional
- É convenção utilizar palavras com letras maiúsculas separas por traço baixo (_)

Variáveis estáticas

- Exemplos

```
static NEXT_ID: u64 = 10;
```

- Mesma sintaxe e convenção de constantes
- O uso de variáveis estáticas é incomum em Rust
- Modificação de variáveis estáticas é inseguro

Variáveis

- Por padrão, variáveis são imutáveis

```
fn f(a: u32) {
    let b = 10;
    ...
    a = 10; // cannot assign to immutable argument
    b = 20; // cannot assign twice to immutable variabl
}
```

Variáveis

- O modificador `mut` é usado para tornar variáveis mutáveis

```
fn f(mut a: u32) {
    let mut x = 10;
    ...
    a = 10;
    b = 20;
}
```

Estruturas de controle

if

```
if condicao {
    consequente
}

if codicao {
    consequente
} else {
    alternativa
}
```

while

```
while condicao {
    instrucoes
}
```

break e **continue** podem ser usados como em C ou Java

loop

```
loop {
    instrucoes
}
```

break e **continue** podem ser usados como em C ou Java

for

```
// versão simplificada
for var in iterator {
    instrucoes
}
```

break e **continue** podem ser usados como em C ou Java

Funções

Máximo entre dois valores

```rust
pub fn maximo(a: i32, b: i32) -> i32 {
    if a > b {
        return a;
    } else {
        return b;
    }
}

#[test]
fn maximo_test() {
    assert_eq!(-1, maximo(-1, -2));
    assert_eq!(6, maximo(4, 6));
    assert_eq!(5, maximo(5, 5));
}
```

Máximo entre dois valores

```rust
pub fn maximo(a: i32, b: i32) -> i32 {
    // if é uma expressão em Rust, isto é,
    // produz um valor
    let max = if a > b {
        a
    } else {
        b
    };
    return max;
}
```

Máximo entre dois valores

```
pub fn maximo(a: i32, b: i32) -> i32 {
    let max = if a > b {
        a
    } else {
        b
    };
    // funções devolvem o valor da última expressão
    // neste caso não é necessário return
    // note a ausência de ;
    max
}
```

Máximo entre dois valores

```rust
pub fn maximo(a: i32, b: i32) -> i32 {
    // simplificando...
    // devolve diretamente o valor produzido pelo if
    if a > b {
        a
    } else {
        b
    }
}
```

Posse, empréstimos e referências

Posse, empréstimos e referências

- Posse
- Empréstimos e referências
- Slices
- Funções que produzem referências
- Referências mutáveis

Posse

O que é posse?

- Posse (*ownership*) é um conceito central em Rust
  - A gerência de memória é feita através do sistema de posse e um conjunto de regras verificadas em tempo de compilação
- Regras
  - Cada valor tem uma variável que é chamada de dono
  - Só pode haver um dono para cada valor
  - Quando o dono sai do escopo, o valor é descartado

Transferência posse

- Para alguns tipos, quando uma variável é atribuída para outra, passada como parâmetro ou retornada de uma função, uma cópia do valor da variável é criada
- Estes tipos são chamados de tipos cópia (implementam o *trait* `Copy`). Todos os tipos primitivos são tipos cópia

Transferência posse

```rust
fn max(a: i32, b: i32) -> i32 {
    if a > b {
        a // cópia
    } else {
        b // cópia
    }
}
fn main() {
    // i32 é um tipo cópia
    let x = 10i32;
    let y = x; // cópia
    println!("{}", max(x, y + 1)); // cópia do x
}
```

Transferência posse

- Para os tipos não cópia, quando uma variável é atribuída para outra, passada como parâmetro ou retornada de uma função, a posse do valor é transferida
- Após a transferência de posse, o valor não pode mais ser acessado através da variável que tinha a posse do valor anteriormente

Transferência posse

```rust
fn max(a: Box<i32>, b: Box<i32>) -> Box<i32> {
    // transfere a posse do valor de a ou b no retorno
    // e descarta (desaloca) o outro valor
    if a > b { a } else { b }
}
fn main() {
    // Box armazena um valor do tipo especificado no heap
    // quando uma variável que tem a posse de um valor do tipo
    // Box sai do escopo, o valor é desalocado
    let x = Box::new(10);
    let y = Box::new(20);
    // a posse dos valores de x e y são transferidas na chamada de max
    // depois que o resultado é exibido, ele é desalocado
    println!("{}", max(x, y));
    // o valor não pode mais ser acessado através de x
    println!("{}", x); // erro: value used here after move ...
}
```

Referências e empréstimos

Referências e empréstimos

- Ao invés de transferir a posse do valor de uma variável para outra, podemos fazer um empréstimo (*borrowing*) do valor
- A forma usada pelo Rust para fazer empréstimos é através de referências
- Uma referência é criado com o operador &
- Referências permitem que variáveis referenciem valores sem ter a posse deles

Referências e empréstimos

```rust
fn maximo(valores: &Vec<i32>) -> i32 {
    // obs: um programador Rust escreveria este código
    //      de outra forma...
    let mut max = valores[0];
    for i in 1..valores.len() {
        if valores[i] > max {
            max = valores[i];
        }
    }
    max
}
fn main() {
    let valores = vec![1, 4, 6, 3];
    println!("{}", maximo(&valores));
    println!("{}", valores[1]); // funciona
}
```

Referências e empréstimos

- Alterar um valor que pode ser acessado por mais de uma variável no mesmo contexto pode gerar erros de memória
- Por isso o Rust não permite que um valor seja alterado nestes casos

Referências e empréstimos

- O código

```rust
let mut x = vec![1, 2, 3];
let y = &x[0];
x[2] = 10;
println!("y = {}", y);
```

- Gera o erro

```
3 |     let y = &x[0];
  |             - immutable borrow occurs here
4 |     x[2] = 10;
  |     ^ mutable borrow occurs here
5 |     println!("y = {}", y);
  |                        - immutable borrow later used her
```

Slices

Referências parciais

- A função `maximo` parece muito restrita, ela sempre requer uma referência para um vetor inteiro
- E se quisermos o máximo dos três primeiros elementos, ou dos 10 últimos, como faríamos?
- Podemos criar uma referência para um pedaço do vetor
- Referências para uma sequência contígua de elementos é chamada de *slice* em Rust

Slices

- Uma slice do tipo `T` é declarado como:

  `&[T]`

- Note que o símbolo `&` ainda é usado, pois uma slice é uma referência para um valor emprestado, isto é, que tem outro dono

Vector e slices

```rust
fn maximo(valores: &[i32]) -> i32 {
    let mut max = valores[0];
    for i in 1..valores.len() {
        if valores[i] > max {
            max = valores[i];
        }
    }
    max
}
fn main() {
    let valores = vec![1, 4, 6, 3];
    println!("{}", maximo(&valores[..3])); // &[1, 4, 6]
    println!("{}", maximo(&valores[2..])); // &[6, 3]
    println!("{}", maximo(&valores[1..4])); // &[4, 6, 3]
}
```

Arranjos e slices

- Também podemos criar slices de arranjos de tamanho fixo (alocados na pilha ou no heap)

```
// a declaração explícita dos tipos não é necessária
let valores: [i32; 6] = [1, 7, 3, 9, 2, 4];
let primeiros: &[i32] = &valores[..3]; // &[1, 7, 3]
let ultimos: &[i32] = &valores[4..]; // &[2, 4]
```

Strings e slices

- Literais de string são do tipo `&'static str`, ou seja, slices do tipo `str` com tempo de vida estática
- Valores do tipo `str` são alocadas estaticamente pelo compilador, por isto não é possível ter uma variável que seja dona de uma variável do tipo `str`
- O tipo `String` deve ser usado quando for necessário criar ou modificar strings em tempo de execução

Strings e slices

- É possível criar uma `String` a partir de uma `&str`

```
let mut s = String::from("casa");
s.push_str(" da sogra");
assert_eq!("casa da sogra", s);
// obtém uma slice para a primeira palavra da string
let c: &str = s.split_whitespace().next().unwrap();
assert_eq!("casa", c);
// obtém uma slice de uma slice
assert_eq!("c", &c[..1]);
```

Strings e slices

- `String` é como `Vec` e `&str` é como `&[]`
- Valores do tipo `String` assim como `Vec` são alocados do heap
- Da mesma forma que é preferível receber parâmetros do tipo `&[]` ao invés de `&Vec`, também é preferível receber parâmetros do tipo `&str` ao invés de `&String`

Funções que produzem referências

Funções que produzem referências

- Defina uma função que receba como entrada uma slice de valores e um valor chave e devolva uma referência para a primeira ocorrência do valor chave na slice

Funções que produzem referências

```rust
pub fn primeiro(valores: &[i32], chave: i32) -> &i32 {
    for i in 0..valores.len() {
        if valores[i] == chave {
            return &valores[i];
        }
    }
    panic!("Não encontrado!");
}

#[test]
fn primeiro_test() {
    let valores = [6, 3, 1, 7, 1];
    let mut r = primeiro(&valores, 6);
    // compara os ponteiros, não os valores
    assert_eq!(&valores[0] as *const _, r as *const _);
    r = primeiro(&valores, 1);
    assert_eq!(&valores[2] as *const _, r as *const _);
    assert_ne!(&valores[4] as *const _, r as *const _);
}
```

Funções que produzem referências

- Como pensar sobre a referência produzida pela função `primeiro`?
- Referências não são donas dos valores que elas referenciam
- Sempre que existe uma referência, existe alguém que é dono do valor referenciado por ela. O compilador garante isso em tempo de compilação!
- Como consequência, uma referência não pode ser criada do nada. Ou uma referência é criada a partir de uma variável que é dona de uma valor, ou a partir de outra referência

Funções que produzem referências

- No caso da referência produzida pela função `primeiro` existe apenas duas opções
  - Retornar uma referência para uma valor que tem uma variável estática como dona (o compilador não precisa se preocupar com o tempo de vida, pois a variável existe durante toda a execução do programa)
  - Retornar uma referência derivada de `valores` (o compilador precisa garantir que a referência seja descartada antes do valor que ela referencia)

Funções que produzem referências

- Não é possível distinguir os dois casos a partir da definição de `primeiro`, então o compilador assume o caso `valores`, que é mais restrito (podemos denotar a referência por `&'static` para informar o compilador do primeiro caso)

Funções que produzem referências

- Observe que não é possível retornar uma referência para uma variável local não estática. Isto porque a variável local deixa de existir quanto a função retorna, mas a referência continua existindo, neste caso estaríamos referenciando um valor que não tem mais dono, e o compilador não permite isso

Funções que produzem referências

- O que acontece se passarmos o valor da chave como referência? (Se a chave é de um tipo grande, a passagem por valor é custosa)
  - Agora o valor produzido por `primeiro` pode tanto ser derivado de `valores` como derivado de `chave`
  - Neste caso o compilador não tem informações suficientes para o uso seguro das referências

Funções que produzem referências

- Sem ver o corpo da função `primeiro`, você diria que este código é seguro quanto ao uso da memória?

```
pub fn primeiro(valores: &[i32], chave: &i32) -> &i32 {
    // ...
}
#[test]
fn primeiro_test() {
    let valores = [6, 3, 1, 7, 1];
    let r = {
        let chave = 7;
        primeiro(&valores, &chave)
    };
    assert_eq!(&valores[3] as *const _, r as *const _);
}
```

Funções que produzem referências

O compilador também não sabe! Por isso ele diz que precisa de mais informações

```
1 | pub fn primeiro(valores: &[i32], chave: &i32) -> &i32 {
  |                                                  ^ expected lifetime
  |                                                    parameter
  |
  = help: this function's return type contains a borrowed value,
    but the signature does not say whether it is borrowed
    from `valores` or `chave`
```

Funções que produzem referências

Precisamos informar ao compilador que a referência produzia por `primeiro` é derivada de `valores`

```
pub fn primeiro<'a>(valores: &'a [i32], chave: &i32) -> &'a i32 {
    for i in 0..valores.len() {
        if valores[i] == *chave {
            return &valores[i];
        }
    }
    panic!("Não encontrado!");
}
```

Funções que produzem referências

- Com a anotação do tempo de vida (*lifetime*) o compilador consegue distinguir que este código é valido

```
let valores = [6, 3, 1, 7, 1];
let r = {
    let chave = 7;
    primeiro(&valores, &chave)
};
```

- E que este não é

```
let chave = 7;
let r = {
    let valores = [6, 3, 1, 7, 1];
    primeiro(&valores, &chave)
};
```

Referências mutáveis

Referências mutáveis

- A referências que vimos até agora são imutáveis, ou seja, não é possível modificar o valor referenciado

```
let x = 10;
let y = &x;
*y = 10;
// erro: `y` is a `&` reference, so the data it
// refers to cannot be written
```

- Observe que adicionar **mut** a variável permite que y referencie outro valor, mas não modificar o valor referenciado

```
let a = 5;
let x = 10;
let mut y = &x;
// ...
y = &a;
```

Referências mutáveis

- O que queremos é uma referência mutável (`&mut`)

```
let mut x = 10;
let y = &mut x;
*y = 20;
assert_eq!(20, x);
```

Referências mutáveis

- Note que não é possível ler o valor de `x` se uma referência mutável é usada posteriormente ao uso de `x`

```
let mut x = 10;
let y = &mut x;
*y = 20;
assert_eq!(20, x);
*y = 30;
```

```
3 |     let y = &mut x;
  |             ------ mutable borrow occurs here
4 |     *y = 20;
5 |     println!("{}", x);
  |                    ^ immutable borrow occurs here
6 |     *y = 30;
  |     ------- mutable borrow later used here
```

Referências mutáveis

- Enquanto é possível ter várias referências imutáveis para um valor, não é possível ter mais que uma referência mutável para o mesmo valor
- Por esta razão as referências imutáveis também são chamadas de referências compartilhadas e as referências mutáveis de referências exclusivas

Estruturas, enumerações e métodos

Estruturas

- Estruturas são definidas com **`struct`**

```
pub struct Ponto {
    x: f64,
    y: f64,
}

let p = Ponto { x: 1.0, y: 2.0 };
assert_eq!(1.0, origem.x);
assert_eq!(2.0, origem.y);
```

Estruturas

- Por padrão os tipos definidos pelo usuário não são do tipo cópia
- Se todos os campos de uma estrutura são do tipo cópia, ela pode ser marcada como tipo cópia usando o atributo `#[derive(Copy)]`

```
#[derive(Copy)]
pub struct Ponto {
    x: f64,
    y: f64,
}
```

Estruturas

- É possível gerar um construtor que inicializa cada campo com um valor padrão (se houver)

```
#[derive(Default)]
pub struct Ponto {
    x: f64,
    y: f64,
}

let p = Ponto::default();
assert_eq!(0.0, p.x);
assert_eq!(0.0, p.y);
```

Estruturas

- Também podemos criar nossos próprios construtores (com qualquer nome)

```
impl Ponto {
    pub fn new(x: f64, y: f64) -> Ponto {
        Ponto { x, y }
        // equivalente a Ponto { x: x, y: y}
    }
}

let p = Ponto::new(1.0, 2.0);
assert_eq!(1.0, p.x);
assert_eq!(2.0, p.y);
```

Estruturas

- É possível adicionar métodos a estrutura

```
impl Ponto {
    fn distancia_origem(&self) -> f64 {
        (self.x * self.x + self.y * self.y).sqrt()
    }
}

let p = Ponto::new(3.0, 4.0)
assert_eq!(5.0, p.distancia_origem());
```

Estruturas

- Note que o Rust cria automaticamente uma referência para `p` para passar como primeiro argumento para `distancia_origem`
  - `p.distancia_origem()` é equivalente, neste caso, a
  - `Ponto::distancia_origem(&p)`
- É possível ter mais que um bloco `impl` para cada estrutura

Estruturas

- O primeiro parâmetro de um método é sempre uma instância do tipo para o qual o método está sendo definido. Pode ser (entre outros)
  - `self`
  - `&self`
  - `&mut self`

Estruturas

- Quando definir métodos ou funções?
  - Os métodos são interessantes quanto a funcionalidade é inerente a uma estrutura particular
  - Funções caso contrário

Estruturas

- Também podemos informar ao compilador usando `derive` que queremos outras funcionalidades em uma estrutura
  - `Debug`: permite exibir a estrutura usando `"{:?}"`
  - `PartialEq`: permite comparar por igualdade parcial
  - `PartialOrd`: permite comparar por ordem parcial
- Obs: Ordenação de números de ponto flutuante é difícil!

Enums e casamento de padrões

Enums e casamento de padrões

- Tipos enumerados são criado com `enum`

```
enum Sinal {
    Verde,
    Amarelo,
    Vermelho,
}
```

Enums e casamento de padrões

- Para verificar o valor de um enum, podemos usar a construção `match`

```
impl Sinal {
    fn proximo(&self) -> Sinal {
        match self {
            Sinal::Verde => Sinal::Amarelo,
            Sinal::Amarelo => Sinal::Vermelho,
            Sinal::Vermelho => Sinal::Verde,
        }
    }
}
...
let mut sinal = Sinal::Verde;
sinal = sinal.proximo();
```

Enums e casamento de padrões

- Todos os possível casos precisam ser considerados em um `match`
- Podemos usar _ para os casos restantes sem precisar enumerá-los

```
impl Sinal {
    fn livre(&self) -> bool {
        match self {
            Sinal::Vermelho => false,
            _ => true,
        }
    }
}
```

Enums e casamento de padrões

- Por padrão, nenhuma operação é definida para os tipos enumerados
- Temos que implementar as operações ou pedir pro compilar usando `derive`

Enums e casamento de padrões

```rust
#[derive(PartialEq, Debug)]
enum Sinal {
    Verde,
    Amarelo,
    Vermelho,
}
let sinal = Sinal::Vermelho;
// PartialEq permite usar as operações == e !=
if sinal == Sinal::Vermelho {
    ...
}
// Debug permite mostrar o valor usando {:?}
println!("{:?}", sinal); // printa Vermelho
// assert_eq! requer que o tipo implemente PartialEq e Debug
assert_eq!(Sinal::Verde, sinal.proximo());
```

Enums e casamento de padrões

- Cada variante de um `enum` também pode ter campos
- Isso torna enums em Rust semelhante a tipos algébricos de dados de outros linguagens

Enums e casamento de padrões

```
enum Comando {
    Listar,
    Remover(usize),
    Incluir {
        item: String,
        preco: u64,
    }
}

let comando = ...;

match &comando {
    Comando::Lista => ...,
    Comando::Remover(x) => x ...,
    Comando::Incluir { item: i, preco: p } => i, p ...,
}
```

Enums e casamento de padrões

- Ou podemos criar as vinculações usando apenas o nome do campo

```
match &comando {
    ...
    Comando::Incluir { item, preco } => item, preco ...
}
```

Enums e casamento de padrões

- Também podemos especificar o valor dos campos e adicionar condições

```
match &comando {
    Comando::Lista => ...,
    Comando::Remover(0) => ...,
    Comando::Remover(12) => ...,
    Comando::Remover(x) if x > 10 => x ...,
    Comando::Remover(x) => x ...,
    ...
}
```

Enums e casamento de padrões

- A condição de uma cláusula é chamada de guarda
- A especificação do caso (sem a guarda) é chamada de padrão
- O `match` e outras construções do Rust suportam casamento de padrão

Option

- Um tipo bastante utilizado em Rust é o tipo `Option`, definido da seguinte forma

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

- Ele é usado para representar um valor que está presente ou não

Option

- No exemplo para encontrar o valor máximo de uma slice, podemos retornar `None` ao invés de chamar `panic!()` se a slice for vazia

```
fn maximo(valores: &[i32]) -> Option<i32> {
    if valores.is_empty() {
        None
    } else {
        let mut max = valores[0];
        for i in 1..valores.len() {
            if valores[i] > max {
                max = valores[i];
            }
        }
        Some(max)
    }
}
```

Option

- Quem chama a função máximo deve decidir como tratar o caso de slice vazia

```
let valores = ...;
match maximo(valores) {
    Some(m) => ...
    None => ...
}
// ou
if let Some(m) = maximo(valores) {
    ...
} else {
    // sabemos que é None
}
```

- Se quem chama a função tem certeza que a slice não é vazia, então ele pode chamar `unwrap` ou `expect`

```
// unwrap extrai o valor de Some ou chama
// panic!() para None
let m = maximo(valores).unwrap();

// expect extrai o valor de Some ou chama panic!()
// com a msg para None
let m = maximo(valores).expected("Deu ruim!");
```

Option

- Implementação de unwrap e expect

```
pub fn unwrap(self) -> T {
    match self {
        Some(val) => val,
        None => panic!(
            "called `Option::unwrap()` on a `None` value"),
    }
}

pub fn expect(self, msg: &str) -> T {
    match self {
        Some(val) => val,
        None => panic!(msg),
    }
}
```

Result

- Outro enum bastante usado é o `Result`, definido como

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

- É usado para retornar e propagar erros. A variante `Ok` representa sucesso e `Err` representa erro

Result

- Por exemplo, a função `Stdin::read_line`, que lê uma linha da entrada padrão, retorna o número de bytes lidos em caso de sucesso ou `io::Error` em caso de erro

```
fn echo() -> Result<(), io::Error> {
    let mut input = String::new();

    match io::stdin().read_line(&mut input) {
        Ok(_) => println!("{}", input),
        Err(e) => return Err(e),
    }

    Ok(())
}
```

Result

- Erros podem ser propagados usando o operador ?

```
fn echo() -> Result<(), io::Error> {
    let mut input = String::new();
    io::stdin().read_line(&mut input)?;
    println!("{}", input);
    Ok(())
}
```

Result

- Os métodos `unwrap` e `expect` também são definidos para `Result`

```
let x: i32 = "123".parse().unwrap();
```

Outros tipos importantes

Iterator

- Tipos que implementam `IntoIterator` produzem iteradores e podem ser usados no for

```
let valores = vec![1, 2, 3];
for v in &valores {
    println!("{}", v);
}
```

- Veja a documentação do módulo `std::iter`

Tipos

- Coleções
  - `Vec<T>`
  - `HashMap<K, V>`
- Veja a documentação do módulo `std::collections`

Tipos

- Ponteiros inteligentes
  - `Box<T>`
  - `Rc<T>`
  - `Arc<T>`
- Veja a seção Smart pointers do livro

Tipos

- Células mutáveis e compartilháveis
  - `Cell<T>`
  - `RefCell<T>`
  - `UnsafeCell<T>`

Traits

- Geral
  - `Default`
  - `Copy`
  - `Debug`
  - `Display`

Traits

- Operadores relacionais
  - `PartialEq` e `Eq`
  - `PartialOrd` e `Ord`

Traits

- Conversão
  - `From<T>`
  - `Into<T>`
  - `FromStr`