Gerenciando Projetos em Crescimento com Pacotes, Crates e Módulos

Diogo Silveira Mendonça

Slides Baseados no Capítulo 7 do Rust Book
https://doc.rust-lang.org/book/ch07-00-managing-growing-projects-with-packages-crates-and-modules.html

Introdução

- Neste capítulo veremos como podemos organizar um projeto em Rust, pensando no escalonamento do projeto.

- Vermos os conceitos de:
  - Pacotes
  - Crates
  - Módulos
  - Caminhos

- Os exemplos de código podem ser encontrados nesse link.

Pacotes e Crates

- Uma crate é a menor quantidade de código que o compilador Rust considera de uma vez.

- Crate pode ter duas formas:
  - **Crate binária**: São programas que você pode compilar para um executável, tendo uma função main. Exemplo: Todos os programas que fizemos anteriormente.

  - **Crate de biblioteca**: Não são compiladas para um executável e não possui uma função main. Sendo compartilhadas por diversos programas. Exemplo: A crate rand.

Pacotes e Crates

- Pacote é um conjunto de uma ou mais crates que fornece um conjunto de funcionalidades. Devendo possuir no mínimo uma crate.

- Um pacote pode possuir quantas crates binárias você quiser, mas apenas uma crate de biblioteca.

Pacotes e Crates

- Executando cargo new, teremos o arquivo Cargo.toml, que fornece um pacote.
- Se abrirmos o arquivo Cargo.toml, vamos perceber que ele não faz menção a src/main.rs.
- src/main.rs é a raiz da crate de uma crate binária com o mesmo nome do pacote.
- Se o diretório src/lib.rs existir, o pacote contém uma crate de biblioteca com o mesmo nome do pacote, sendo a raiz da crate.

$ cargo new my-project

    Created binary (application) `my-project` package

$ ls my-project

Cargo.toml

src

$ ls my-project/src

main.rs

Pacotes e Crates

- O Cargo sabe que src/main.rs e src/lib.rs são raízes da crate por padrão.
- O Cargo passa os arquivos raiz da crate para o rustc para construir a biblioteca ou o binário.
- Um pacote pode ter diversas crates binárias, colocando os arquivos no diretório src/bin, cada arquivo sendo uma crate separada.

$ cargo new my-project
    Created binary (application) `my-project` package

$ ls my-project
Cargo.toml
src

$ ls my-project/src
main.rs

Definindo módulos para controlar escopo e privacidade

- Referência de módulos e caminhos
  - Palavras chaves use e pub.
- Começa pela raiz da crate:
  - Ao compilar uma crate, o compilador procura o arquivo na raiz da crate.
    - src/lib.rs
    - src/main.rs
- Declaração de módulos:
  - Podemos declarar novos módulos, no arquivo raiz da crate.
    - Declara um módulo "garden" com “mod garden;”
    - O código do módulo poderá estar em:
      - Inline, dentro de chaves após “mod garden”.
      - src/garden.rs
      - src/garden/mod.rs

Definindo módulos para controlar escopo e privacidade

- Declaração de submódulos
  - Em arquivos que não são o raiz da crate.
  - Declarar um submódulo vegetable com “mod vegetables;” dentro de src/garden.rs.
    - Inline, dentro de chaves após “mod vegetables”.
    - src/garden/vegetables.rs
    - src/garden/vegetables/mod.rs

- Caminhos para um código em módulos:
  - Quando o módulo estiver declarado na crate.
  - Se a privacidade for pública, podemos chamar o código de um módulo, utilizando o caminho do código.
    - Um tipo Asparagus no submódulo vegetables do módulo garden seria encontrado utilizando: crate::garden::vegetables::Asparagus

Definindo módulos para controlar escopo e privacidade

- Privado x Público
  - Por default o módulo é privado.
  - Para deixar público é preciso utilizar “pub mod” em vez de mod.
  - Para tornar os itens dentro de um mod public, utilize pub antes da declaração.

- Palavra chave use
  - Use serve para criar atalhos de caminhos.
  - Utilizando “use crate::garden::vegetables::Asparagus;” poderemos depois escrever somente “Asparagus”, em vez do caminho completo.

Definindo módulos para controlar escopo e privacidade

- Seguindo os passos anteriores, vamos ter uma estrutura de arquivos igual ao do lado.
- A crate binária backyard
- O módulo garden
- O Submódulo vegetables
- No arquivo vegetables.rs, temos Asparagus.

```
backyard
├── Cargo.lock
├── Cargo.toml
└── src
    ├── garden
    │   └── vegetables.rs
    ├── garden.rs
    └── main.rs
```

Definindo módulos para controlar escopo e privacidade

- src/main.rs
  - pub mod garden, chama src/garden.rs
  - Link do código.

```
use crate::garden::vegetables::Asparagus;

pub mod garden;

fn main() {
    let plant = Asparagus {};
    println!("I'm growing {:?}!", plant);
}
```

Definindo módulos para controlar escopo e privacidade

- Em src/garden.rs
  - A linha “pub mod vegetables” diz ao compilador para incluir o código do módulo vegetables que está no arquivo src/garden/vegetables.rs;

- O arquivo src/garden/vegetables.rs tem o código de Asparagus

```
#[derive(Debug)]
pub struct Asparagus {}
```

Definindo módulos para controlar escopo e privacidade

- Vamos criar um crate de biblioteca, que aplica a funcionalidade de um restaurante. Focando primeiro na estrutura dos arquivos, depois na funcionalidade.

- Um restaurante normalmente é dividido em duas partes:
  - Back of House.
  - Front of House.

- cargo new restaurant --lib

Definindo módulos para controlar escopo e privacidade

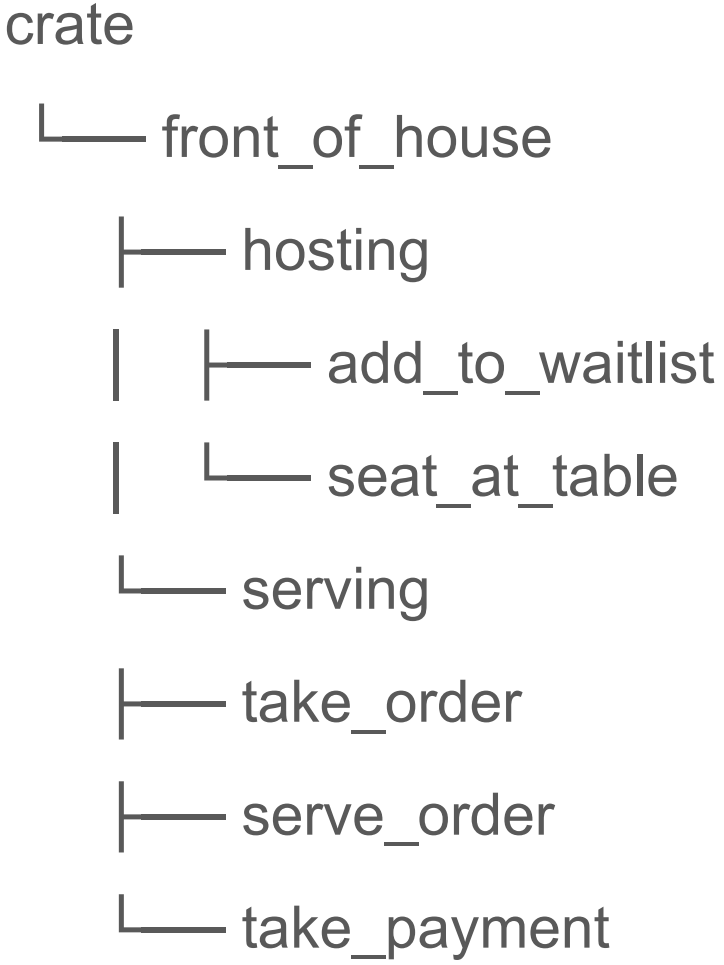
- Adicionando o módulo front_of_house em src/lib.rs
- Adicionamos os submódulos hosting e serving.
- Os submódulos, tem itens definidos como funções. Também poderiam ser definidos como structs, enums, constants e traits.

src/lib.rs:

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}

        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}

        fn serve_order() {}

        fn take_payment() {}
    }
}
```

Definindo módulos para controlar escopo e privacidade

- Ao lado temos a organização do módulo front_of_house.



Caminhos para referenciar o um item na árvore de módulos

- Um caminho pode ter duas formas
  - Absoluto, é o caminho completo a partir da raiz da crate. Começa com a palavra chave “crate”
  - Relativo, começa o caminho pelo módulo atual. Utiliza “self”, “super” ou um identificador no módulo atual.
- Cada parte do caminho é separado por ::
- eat_at_restaurant está no mesmo nível que front_of_house.

Esse código gera um erro! **src/lib.rs:**

```
mod front_of_house {
        mod hosting {
                fn add_to_waitlist() {}
        }
}

pub fn eat_at_restaurant() {
        // Absolute path
        crate::front_of_house::hosting::add_to_waitlist();

        // Relative path
        front_of_house::hosting::add_to_waitlist();
}
```

Caminhos para referenciar o um item na árvore de módulos

- Tentando rodar o código, recebemos o erro:

  error[E0603]: module `hosting` is private

- A mensagem de erro mostra que hosting é privado.

- Um módulo pai não pode acessar o item de um módulo filho, caso ele seja privado.

- Um módulo filho pode acessar os itens de seus ancestrais, mesmo se eles forem privados.

Expondo caminhos com a palavra-chave pub

- Deixando o módulo hosting público.
- Rodando dessa forma, teremos o erro:

  error[E0603]: function `add_to_waitlist` is private
- Apesar do módulo hosting estar público, o seu conteúdo ainda é privado.
- Deixar um módulo público. Só permite que o código em seus módulos ancestrais se refira a ele, não permite acessar seu código interno

src/lib.rs:

```
mod front_of_house {
        pub mod hosting {
                fn add_to_waitlist() {}
        }
}

pub fn eat_at_restaurant() {
        // Absolute path
        crate::front_of_house::hosting::add_to_waitlist();

        // Relative path
        front_of_house::hosting::add_to_waitlist();
}
```

Expondo caminhos com a palavra-chave pub

- Deixando a função add_to_waitlist() pública.
- Agora a função eat_at_restaurant() vai conseguir acessar o item add_to_waitlist().
- eat_at_restaurant pode acessar front_of_house, que é privado, pois estão no mesmo nível, ou seja, são irmãos.
- Pode acessar hosting e add_to_waitlist, por conta que eles estão públicos.

src/lib.rs:

```
mod front_of_house {
        pub mod hosting {
                pub fn add_to_waitlist() {}
        }
}

pub fn eat_at_restaurant() {
        // Absolute path
        crate::front_of_house::hosting::add_to_waitlist();

        // Relative path
        front_of_house::hosting::add_to_waitlist();
}
```

Começando caminhos relativos com super

- Podemos utilizar super para começar o caminho do módulo pai, em vez do módulo atual.
- No código ao lado, implementa a situação onde o chef corrige um pedido incorreto e o entrega pessoalmente ao cliente.
- Usamos super para referenciar o pai de back_of_house, que é a crate. Depois acessamos deliver_order.
- A utilização do super pode ser útil para evitar trabalhos futuros com correções, caso o código seja movido para um módulo diferente.

src/lib.rs:

```
fn deliver_order() {}

mod back_of_house {
    fn fix_incorrect_order() {
        cook_order();
        super::deliver_order();
    }

    fn cook_order() {}
}
```

Tornando estruturas e enumerações públicas.

- Podemos utilizar pub para tornar estruturas e enumerações públicas.
- Utilizar pub antes da definição de uma estrutura, tornará a estrutura pública, mas seus campos continuarão privados.
- Podemos escolher quais campos serão privados e quais serão públicos.
- O código a seguir mostra o caso, onde o cliente pode escolher o pão que acompanha sua refeição, mas o chefe que escolhe a fruta que acompanha a refeição. O cliente não pode escolher nem ver qual fruta vai receber.

src/lib.rs:

```
mod back_of_house {
        pub struct Breakfast {
                pub toast: String,
                seasonal_fruit: String,
        }

        impl Breakfast {
                pub fn summer(toast: &str) -> Breakfast {
                                Breakfast {
                                toast: String::from(toast),
                                seasonal_fruit: String::from("peaches"),
                                }
                }
        }
}
pub fn eat_at_restaurant() {
        // Order a breakfast in the summer with Rye toast
        let mut meal = back_of_house::Breakfast::summer("Rye");

        // Change our mind about what bread we'd like
        meal.toast = String::from("Wheat");
        println!("I'd like {} toast please", meal.toast);

        // The next line won't compile if we uncomment it; we're not allowed
        // to see or modify the seasonal fruit that comes with the meal
        // meal.seasonal_fruit = String::from("blueberries");
}
```

Tornando estruturas e enumerações públicas.

- A estrutura Breakfast é pública, assim como o campo toast.
- seasonal_fruit é privado.
- eat_at_restaurant pode acessar toast.
- Se descontarmos a linha que tentar acessar seasonal_fruit teremos um erro.

src/lib.rs:

```
pub struct Breakfast {
        pub toast: String,
        seasonal_fruit: String,
    }

pub fn eat_at_restaurant() {
```

Tornando estruturas e enumerações públicas.

- Quando tornamos uma enumeração pública, todas as suas variantes se tornam públicas.
- eat_at_restaurant pode acessar Soup e Salad, pois Appetizer é público.

src/lib.rs:

```
mod back_of_house {
    pub enum Appetizer {
        Soup,
        Salad,
    }
}

pub fn eat_at_restaurant() {
    let order1 =
        back_of_house::Appetizer::Soup;
    let order2 =
        back_of_house::Appetizer::Salad;
}
```

Trazendo o caminho para o escopo, utilizando “use”

- Podemos utilizar a palavra-chave use para criar um atalho, simplificando os caminhos.
- O atalho que o use cria, serve somente para aquele escopo.
- O use verifica a privacidade, assim como qualquer outro caminho.
- Depois de criar o atalho, podemos acessar hosting como se ele fosse a raiz da crate.

src/lib.rs:

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

Trazendo o caminho para o escopo, utilizando “use”

- No exemplo ao lado, o use está fora do escopo.
- Teremos o erro:

  error[E0433]: failed to resolve: use of undeclared crate or module `hosting`
- Nesse caso como o atalho de use não foi utilizado, teremos também o warning:

  warning: unused import: `crate::front_of_house::hosting`

src/lib.rs:

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

mod customer {
    pub fn eat_at_restaurant() {
        hosting::add_to_waitlist();
    }
}
```

Criando caminhos idiomáticos

- Poderíamos também ter utilizado o use para criar um atalho até add_to_waitlist, em vez de somente até hosting.
- Então acessar utilizando somente add_to_waitlist();.
- O atalho anterior até o hosting é um exemplo de caminho idiomático.
- Um caminho idiomático nos permite saber qual a origem daquele item.
- No exemplo ao lado não fica tão claro a origem de add_to_waitlist.

src/lib.rs:

```
mod front_of_house {
        pub mod hosting {
                pub fn add_to_waitlist() {}
        }
}

use crate::front_of_house::hosting::add_to_waitlist;

pub fn eat_at_restaurant() {
        add_to_waitlist();
}
```

Criando caminhos idiomáticos

- O exemplo 1, mostra a forma idiomática de trazer a estrutura HashMap da biblioteca padrão para o escopo de um crate binário.
- Não se tem tanta lógica pela adoção de caminhos idiomáticos, é somente uma forma que as pessoas acostumaram a utilizar.
- Com exceção ao caso onde o nome de um item é compartilhado por dois módulos diferentes.
- Podemos utilizar um caminho idiomático para deixar diferenciado, como no caso do exemplo 2.

Exemplo 1:

src/main.rs:

```
use std::collections::HashMap;

fn main() {
        let mut map = HashMap::new();
        map.insert(1, 2);
}
```

Exemplo 2:

src/lib.rs:

```
use std::fmt;
use std::io;

fn function1() -> fmt::Result {
        // --snip--
}

fn function2() -> io::Result<()> {
        // --snip--
}
```

Atribuindo novos nomes com “as”

- Mesmo com o problema anterior, é possível não adotar um caminho idiomático.
- No exemplo ao lado, std::fmt::Result é simplificado para Result e std::io::Result é simplificado para IoResult.

src/lib.rs:

```
use std::fmt::Result;
use std::io::Result as IoResult;

fn function1() -> Result {
    // --snip--
}

fn function2() -> IoResult<()> {
    // --snip--
}
```

Re-exportando nomes usando “pub use”

- Podemos utilizar pub use, para permitir o uso do atalho fora do escopo que foi criado.

src/lib.rs:

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

Usando pacotes externos

- Assim como mostrado no capítulo 2, adicionamos no Cargo.toml

  ```
  rand = "0.8.5"
  ```

- Isso fala para o Cargo fazer o Download do pacote rand.

- Utilizar o use, para podermos utilizar no escopo.

  ```
  use rand::Rng;
  fn main() {
      let secret_number = rand::thread_rng().gen_range(1..=100);
  }
  ```

- Para bibliotecas padrões, não precisamos adicionar no Cargo.toml, somente utilizar o use para podermos utilizar no escopo.

  ```
  use std::collections::HashMap;
  ```

Usando Caminhos Aninhados para Limpar Listas extensas de “use”

- Nos dois exemplos, estamos utilizando dois itens diferentes da biblioteca padrão.
- Podemos simplificar o import do exemplo 1, como mostra no exemplo 2.

src/main.rs

Exemplo 1:

```
// --snip--
use std::cmp::Ordering;
use std::io;
// --snip--
```

Exemplo 2:

```
// --snip--
use std::{cmp::Ordering, io};
// --snip--
```

Usando Caminhos Aninhados para Limpar Listas extensas de “use”

- Nesse caso temos o import de io e o import de Write que pertence a io.
- Podemos simplificar esse import como no exemplo 2.

src/lib.rs:

Exemplo 1:

```
use std::io;
use std::io::Write;
```

Exemplo 2:

```
use std::io::{self, Write};
```

O operador glob

- Se quisermos utilizar todos os itens que estão públicos. Podemos utilizar *, como mostra o exemplo abaixo.

  use std::collections::*;

- Tome cuidado, isso dificulta identificar quais itens são importados e quais são definidos no próprio programa.

Separando Módulos em diferentes arquivos

- Vimos o código ao lado, anteriormente. Vamos separar em outros arquivos src/front_of_house.rs.
- Extrair front_of_house para seu próprio arquivo.
- Remover o código dentro das chaves para o módulo. Deixando somente a declaração.

src/lib.rs:

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

Separando Módulos em diferentes arquivos

- Depois de realizar as modificações, teremos o código ao lado.
- Observe que você só precisa carregar um arquivo usando uma declaração mod uma vez na sua árvore de módulos.
- A seguir vamos deixar hosting em um próprio arquivo. O processo é um pouco diferente pois hosting é um módulo filho de front_of_house.
- Vamos mover o código de hosting para src/front_of_house/hosting.rs.
- Deixar somente a declaração de hosting no arquivo front_of_house.rs
- Link do código.

src/lib.rs:

```
mod front_of_house;

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

src/front_of_house.rs:

```
pub mod hosting {
    pub fn add_to_waitlist() {}
}
```

Separando Módulos em diferentes arquivos

- Depois de realizar as ações, teremos o código ao lado.
- Existe também uma forma diferente de organizar os módulos, utilizando um estilo mais antigo.
- Poderíamos organizar os arquivos dessa forma:
  - src/front_of_house/mod.rs
  - src/front_of_house/hosting/mod.rs
- Não podemos deixar as duas formas de organização para um mesmo módulo.
- Não é recomendado, mas é possível utilizar os dois modelos de organização para módulos diferentes.

src/lib.rs:

```
mod front_of_house;
pub use
crate::front_of_house::hosting;
pub fn eat_at_restaurant() {
        hosting::add_to_waitlist();
}
```

src/front_of_house.rs:

```
pub mod hosting;
```

src/front_of_house/hosting.rs:

```
pub fn add_to_waitlist() {}
```

Referências

Capítulo 7 do Rust Book
https://doc.rust-lang.org/book/ch07-00-managing-growing-projects-with-packages-crates-and-modules.html