Rusting with style - Curso básico de linguagem Rust

Cleuton Sampaio

Veja no GitHub

Menu do curso

VÍDEO DESTA AULA

Arquivos

Exemplo simples de leitura e gravação de arquivos em Rust

A seguir, apresento um exemplo minimalista de como ler o conteúdo de um arquivo e, em seguida, gravar o conteúdo em outro arquivo. Vou explicar passo a passo como funciona cada parte do código.

```
use std::fs::File;
use std::io::{self, Read, Write};

fn main() -> io::Result<()> {
    // 1. Abra o arquivo "input.txt" para leitura
    let mut file = File::open("input.txt")?;

    // 2. Crie uma variável para armazenar o conteúdo lido
    let mut contents = String::new();

    // 3. Leia todo o conteúdo do arquivo para a variável 'contents'
    file.read_to_string(&mut contents)?;

    // 4. Exiba o conteúdo lido no terminal
    println!("Conteúdo do arquivo de entrada: {}", contents);

    // 5. Crie (ou sobrescreva) o arquivo "output.txt" para gravação
    let mut out_file = File::create("output.txt")?;

    // 6. Escreva algo no arquivo de saída; aqui, usamos o conteúdo que
lemos
```

```
    writeln!(out_file, "Conteúdo copiado: {}", contents)?;

    // 7. Retorne Ok(()) caso tudo dê certo
    Ok(())
}
```

Explicação passo a passo

1. **Importações necessárias**
   - `std::fs::File`: Métodos para lidar com arquivos (abrir, criar, etc.).
   - `std::io::{self, Read, Write}`: Para operações de entrada/saída, como leitura (`Read`) e escrita (`Write`).

2. **Assinatura da função `main`**

```
fn main() -> io::Result<()> {
    // ...
}
```

   - Declaramos que `main` retorna um `io::Result<()>`. Isso significa que qualquer erro de I/O será propagado de forma adequada, em vez de usarmos `panic!`.

3. **Abrindo o arquivo para leitura**

```
let mut file = File::open("input.txt")?;
```

   - `File::open("input.txt")` tenta abrir o arquivo para leitura.
   - O `?` propaga o erro se acontecer falha ao abrir o arquivo.

4. **Lendo o conteúdo do arquivo**

```
let mut contents = String::new();
file.read_to_string(&mut contents)?;
```

   - Cria-se uma `String` vazia chamada `contents`.
   - `read_to_string` lê todo o conteúdo do arquivo e joga na variável `contents`.
   - O `?` novamente propaga qualquer erro de leitura.

5. **Exibindo o conteúdo lido**

```
println!("Conteúdo do arquivo de entrada: {}", contents);
```

- Apenas mostramos no terminal o que foi lido do arquivo.

6. **Criando/abrindo arquivo para escrita**

```
let mut out_file = File::create("output.txt")?;
```

- `File::create("output.txt")` cria o arquivo caso não exista, ou sobrescreve caso já exista.

7. **Gravando no arquivo de saída**

```
writeln!(out_file, "Conteúdo copiado: {}", contents)?;
```

- Escreve no arquivo de saída.
- `writeln!` funciona como `println!`, porém direcionado para um *writer* (no caso, `out_file`).

8. **Retornando `Ok(())`**

```
Ok(())
```

- Indica que tudo correu bem e finaliza a função `main`.

Observações importantes

- **Tratamento de erros**: Em Rust, é comum usar o operador `?` para propagar erros de forma conveniente, evitando `unwrap()` que pode causar panic se algo falhar.
- **Performance**: Para arquivos grandes, pode-se usar métodos de leitura e escrita em blocos (buffers) para melhor performance, mas para um exemplo simples, `read_to_string` já é suficiente.
- **Criação de arquivo**: `File::create` apaga o conteúdo do arquivo se ele já existir. Se quiser abrir para acrescentar (append), usaria algo como `OpenOptions`.

Otimizando e flexibilizando a leitura de arquivos

Pode ser mais flexível usar uma assinatura genérica que receba qualquer tipo que implemente `std::io::Read`, em vez de "prender" a função a ler diretamente de arquivo e retornar uma `String`.** Dessa forma, você consegue reutilizar o mesmo código para ler tanto de arquivos quanto de outros tipos de "fontes" de dados (por exemplo, um buffer de memória, `stdin`, ou até mesmo um stream de rede).

Um exemplo de como ficaria esse design mais genérico, seguido de uma explicação:

```
use std::fs::File;
use std::io::{self, BufReader, Read};

/// Lê todo o conteúdo de um objeto que implementa `Read` e retorna uma
`String`.
```

```rust
fn read_all<R: Read>(mut reader: R) -> io::Result<String> {
    let mut contents = String::new();
    reader.read_to_string(&mut contents)?;
    Ok(contents)
}

fn main() -> io::Result<()> {
    // Exemplo com um arquivo
    let file = File::open("input.txt")?;
    let buf_reader = BufReader::new(file);

    let contents = read_all(buf_reader)?;
    println!("Conteúdo:\n{}", contents);

    Ok(())
}
```

Por que isso é mais desacoplado?

1. **Generalização**
   - A função `read_all` agora não sabe nem se está lendo de um arquivo, do `stdin` ou de um vetor de bytes em memória (`&[u8]`). Tudo que ela precisa é de algo que implemente a trait `Read`.
   - Por exemplo, se amanhã você decidir ler de um `TcpStream`, não precisa alterar `read_all`; basta passar o `TcpStream` como parâmetro.
2. **Testabilidade**
   - Em testes, você pode passar um buffer de memória (p. ex., `std::io::Cursor<&[u8]>`) simulando o conteúdo de um "arquivo falso" ou dados de entrada. Isso facilita o teste de forma independente, sem precisar criar arquivos físicos.
3. **Responsabilidade única**
   - A função `read_all` tem a única responsabilidade de ler *algo* que seja `Read` e devolver o conteúdo em `String`.
   - A lógica de abrir arquivos, tratar caminhos, bufferizar com `BufReader`, etc., fica em outro lugar (no `main`, por exemplo), mantendo as funções mais coesas.
4. **Reuso**
   - O mesmo padrão serve para muitas aplicações onde você quer ler texto de diferentes fontes sem duplicar código.

Comparando com a abordagem inicial

Na abordagem inicial, nós amarramos a função diretamente a um caminho de arquivo (`path: &str`) e já fazíamos `File::open`, `BufReader::new` e `read_to_string` no mesmo local. Isso é simples de entender, mas fica menos flexível, pois a função só serve para aquele caso específico (um caminho de arquivo).

Já na abordagem genérica, nós temos:

- Uma função que lida unicamente com a leitura (de qualquer fonte que implemente `Read`).
- O código que lida com a abertura de arquivos, uso de `BufReader` e afins fica onde for apropriado (p. ex., na `main`).

Assim, se a sua intenção for deixar o código mais modular e reutilizável, **usar `impl Read` (ou `R: Read`) é de fato uma prática mais flexível e desacoplada**.

Arquivos CSV

Arquivos CSV são comuns em aplicações e podemos fazer isso facilmente em Rust, utilizando o crate **csv** em conjunto com o Serde para ler e gravar arquivos CSV. Vou mostrar **dois códigos**: um para **leitura** de um arquivo CSV (dados de "entrada"), e outro para **gravação** de um arquivo CSV (dados de "saída").

Configurando o Cargo.toml

No seu arquivo **Cargo.toml**, inclua as seguintes dependências:

```
[dependencies]
csv = "1"
serde = { version = "1.0", features = ["derive"] }
serde_derive = "1.0"
```

> Se estiver utilizando o **cargo-script** coloque isso dentro de um comentário no código:

```
//! ```cargo
//! [package]
//! edition = "2021"
//! [dependencies]
//! csv = "1"
//! serde = { version = "1.0", features = ["derive"] }
//! serde_derive = "1.0"
//! ```
```

Isso permitirá usar:

- O crate **csv** para manipular arquivos CSV.
- **Serde** e **Serde Derive** para (de)serializar structs de/para CSV de maneira automática.

Exemplo de leitura de CSV em Português

Arquivo CSV de exemplo (dados.csv)

Suponha que temos um arquivo chamado **dados.csv** com o seguinte conteúdo (repare que a **primeira linha** são os cabeçalhos):

```
id,nome,cidade
1,Ana,São Paulo
2,Bruno,Rio de Janeiro
3,Carla,Belo Horizonte
```

Código em Rust para ler o arquivo

```rust
use std::error::Error;
use serde::Deserialize;
use csv::ReaderBuilder;

#[derive(Debug, Deserialize)]
struct Pessoa {
    id: u32,
    nome: String,
    cidade: String,
}

fn main() -> Result<(), Box<dyn Error>> {
    // Cria um reader (leitor) para abrir e ler o arquivo CSV
    let mut leitor = ReaderBuilder::new()
        .has_headers(true)   // Indica que a primeira linha do CSV são
cabeçalhos
        .from_path("dados.csv")?;

    // Lê cada registro (linha) do arquivo, desserializando em uma struct
Pessoa
    for resultado in leitor.deserialize::<Pessoa>() {
        // Se ocorrer algum erro, o ? propaga o erro automaticamente
        let registro = resultado?;

        // Mostra o registro lido
        println!("{:?}", registro);
    }

    Ok(())
}
```

Explicação

1. `#[derive(Debug, Deserialize)]`: Faz com que a struct `Pessoa` possa ser desserializada diretamente de uma linha CSV, além de permitir o uso de `{:?}` para debug (impressão mais "crua").
2. **Cabeçalhos no CSV**: Como definimos `has_headers(true)`, a primeira linha (`id,nome,cidade`) não é convertida em `Pessoa`; ela é usada para mapear qual coluna vai para qual campo na struct.
3. **Laço de leitura**: `leitor.deserialize()` retorna um iterador de `Result<Pessoa, csv::Error>`. O `?` propaga o erro caso aconteça algo inesperado (por exemplo, tipo de dado inválido

ou problema no arquivo).

4. **Flexibilidade**: Se você quiser ler manualmente sem Serde, pode usar `leitor.records()` para trabalhar com cada linha como texto bruto; porém, usando `Deserialize`, o processo de leitura e conversão fica mais simples e seguro.

Exemplo de escrita de CSV em Português

Agora, vejamos um exemplo para **gravar** dados em um arquivo CSV, também usando Serde para serializar automaticamente structs em linhas CSV.

Código em Rust para gerar um arquivo CSV

```
use std::error::Error;
use serde::Serialize;
use csv::WriterBuilder;

#[derive(Debug, Serialize)]
struct Pessoa {
    id: u32,
    nome: String,
    cidade: String,
}

fn main() -> Result<(), Box<dyn Error>> {
    // Criamos um vetor com alguns dados
    let lista_de_pessoas = vec![
        Pessoa {
            id: 1,
            nome: "Ana".to_string(),
            cidade: "São Paulo".to_string(),
        },
        Pessoa {
            id: 2,
            nome: "Bruno".to_string(),
            cidade: "Rio de Janeiro".to_string(),
        },
        Pessoa {
            id: 3,
            nome: "Carla".to_string(),
            cidade: "Belo Horizonte".to_string(),
        },
    ];

    // Cria (ou sobrescreve) o arquivo "saida.csv" para escrita
    let mut escritor = WriterBuilder::new()
        .has_headers(true)  // Fará com que a primeira linha escrita seja o
cabeçalho (id,nome,cidade)
        .from_path("saida.csv")?;

    // Serializa cada "Pessoa" em uma linha CSV
```

```
    for pessoa in lista_de_pessoas {
        escritor.serialize(pessoa)?;
    }

    // Força a gravação de qualquer dado pendente no buffer
    escritor.flush()?;

    println!("Arquivo CSV 'saida.csv' gravado com sucesso!");

    Ok(())
}
```

Explicação

1. `#[derive(Debug, Serialize)]`: Permite que a struct `Pessoa` seja serializada automaticamente para CSV.
2. `WriterBuilder`: Assim como no leitor, podemos configurar detalhes. Com `has_headers(true)`, a primeira vez que chamamos `serialize`, o cabeçalho (`id,nome,cidade`) será gravado automaticamente.
3. **Laço de gravação**: Cada `Pessoa` é transformada em uma linha CSV com base nos campos da struct. Se usar `wtr.write_record(&["...", "...", "..."])` seria a forma manual (sem Serde).
4. `.flush()`: Assegura que tudo esteja efetivamente escrito no arquivo antes de encerrarmos o programa.

- O crate **csv** fornece uma API simples para manipulação de arquivos CSV.
- **Serde** integra-se muito bem com o crate **csv**, permitindo (des)serializar structs de/para CSV automaticamente, desde que os nomes dos campos na struct correspondam aos cabeçalhos do arquivo.
- Com essa abordagem, seu código fica mais enxuto, robusto (pois lida com tipos ao invés de apenas strings) e fácil de manter.

Arquivos JSON

Para manipular arquivos JSON em Rust, a combinação mais comum é usar as crates **serde** (para serialização e desserialização) e **serde_json** (para lidar especificamente com o formato JSON). A seguir apresento exemplos de **leitura** e **gravação** de arquivos JSON em Rust, com nomes de structs, campos e variáveis em Português.

Configurando o *Cargo.toml*

Inclua as seguintes dependências no seu `Cargo.toml`:

```
[dependencies]
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
```

Ou estes comentários, se estiver utilizando **cargo-script**:

```
//! ```cargo
//! [package]
//! edition = "2021"
//! [dependencies]
//! serde = { version = "1.0", features = ["derive"] }
//! serde_json = "1.0"
//! ```
```

- **serde**: Para (de)serialização de dados.
- **serde_json**: Para ler e escrever JSON especificamente.

Exemplo de leitura de um arquivo JSON

Arquivo JSON de exemplo (pessoas.json)

Suponha que temos um arquivo **pessoas.json** assim:

```
[
  {
    "id": 1,
    "nome": "Ana",
    "cidade": "São Paulo"
  },
  {
    "id": 2,
    "nome": "Bruno",
    "cidade": "Rio de Janeiro"
  },
  {
    "id": 3,
    "nome": "Carla",
    "cidade": "Belo Horizonte"
  }
]
```

Código em Rust para ler o arquivo

```
use std::error::Error;
use std::fs::File;
use std::io::BufReader;
use serde::Deserialize;
use serde_json;

#[derive(Debug, Deserialize)]
struct Pessoa {
    id: u32,
    nome: String,
```

```
    cidade: String,
}

fn main() -> Result<(), Box<dyn Error>> {
    // Abre o arquivo
    let arquivo = File::open("pessoas.json")?;
    // Cria um BufReader para ler de forma eficiente
    let leitor = BufReader::new(arquivo);

    // Desserializa o conteúdo do JSON para um vetor de Pessoa
    let lista_de_pessoas: Vec<Pessoa> = serde_json::from_reader(leitor)?;

    // Exibe as pessoas lidas
    for pessoa in lista_de_pessoas {
        println!("{:?}", pessoa);
    }

    Ok(())
}
```

Passo a passo

1. **Struct com `#[derive(Deserialize)]`**
   Permite que cada objeto JSON seja convertido em uma instância de `Pessoa`.

2. **`BufReader`**
   Ler o arquivo através de um buffer é mais eficiente do que leituras pontuais;
   `serde_json::from_reader` exige um leitor que implemente `std::io::Read`.

3. **`serde_json::from_reader`**
   Faz a desserialização automática do JSON em `Vec<Pessoa>` (um vetor de `Pessoa`). Se o formato do JSON não bater com o da struct, um erro será retornado.

4. **Tratamento de Erro**
   Aqui usamos `Result<(), Box<dyn Error>>` e o operador `?` para simplificar a propagação de erros de I/O ou de parsing.

Exemplo de gravação de um arquivo JSON

Agora vamos criar um arquivo JSON a partir de um vetor de structs `Pessoa`.

Código em Rust para escrever o arquivo

```
use std::error::Error;
use std::fs::File;
use serde::Serialize;
use serde_json;

#[derive(Debug, Serialize)]
struct Pessoa {
```

```
    id: u32,
    nome: String,
    cidade: String,
}

fn main() -> Result<(), Box<dyn Error>> {
    // Cria alguns dados de exemplo
    let lista_de_pessoas = vec![
        Pessoa {
            id: 1,
            nome: "Ana".to_string(),
            cidade: "São Paulo".to_string(),
        },
        Pessoa {
            id: 2,
            nome: "Bruno".to_string(),
            cidade: "Rio de Janeiro".to_string(),
        },
        Pessoa {
            id: 3,
            nome: "Carla".to_string(),
            cidade: "Belo Horizonte".to_string(),
        },
    ];

    // Cria (ou sobrescreve) o arquivo saida.json
    let arquivo = File::create("saida.json")?;

    // Escreve em formato JSON "bonito" (com indentação)
    serde_json::to_writer_pretty(arquivo, &lista_de_pessoas)?;

    println!("Arquivo JSON 'saida.json' gravado com sucesso!");

    Ok(())
}
```

Passo a passo

1. **Struct com `#[derive(Serialize)]`**
   Permite serializar a struct `Pessoa` diretamente em JSON.

2. **Criação do arquivo**
   `File::create("saida.json")` vai criar o arquivo (ou sobrescrevê-lo se já existir).

3. **`serde_json::to_writer_pretty`**

   - Serializa e escreve os dados (no caso, `Vec<Pessoa>`) em formato JSON para dentro do `arquivo`.
   - A versão “pretty” adiciona quebras de linha e indentação para melhor legibilidade. Se preferir um formato mais compacto, use `serde_json::to_writer`.

4. **Tratamento de Erro**

    - Qualquer problema de I/O (por exemplo, falta de permissão) ou de serialização é retornado por meio de `Result`.

Observações finais

- **Formas alternativas**:
    - Ler para uma `String` e então chamar `serde_json::from_str(&sua_string)`.
    - Escrever para uma `String` com `serde_json::to_string` e depois salvar a `String` manualmente.
- **Validação**:
    - Caso o JSON recebido não corresponda ao formato de `Pessoa` (tipos incorretos, campos faltando, etc.), ocorrerá erro de desserialização.
- **Organização do código**:
    - Em aplicações reais, é comum ter funções separadas (por exemplo, `fn ler_pessoas_de_arquivo(...)` e `fn gravar_pessoas_em_arquivo(...)`) que façam apenas essa responsabilidade; a `main` então chamaria essas funções.

Dessa forma, utilizando **serde_json**, você consegue ler e gravar arquivos JSON em Rust de maneira fácil e idiomática!