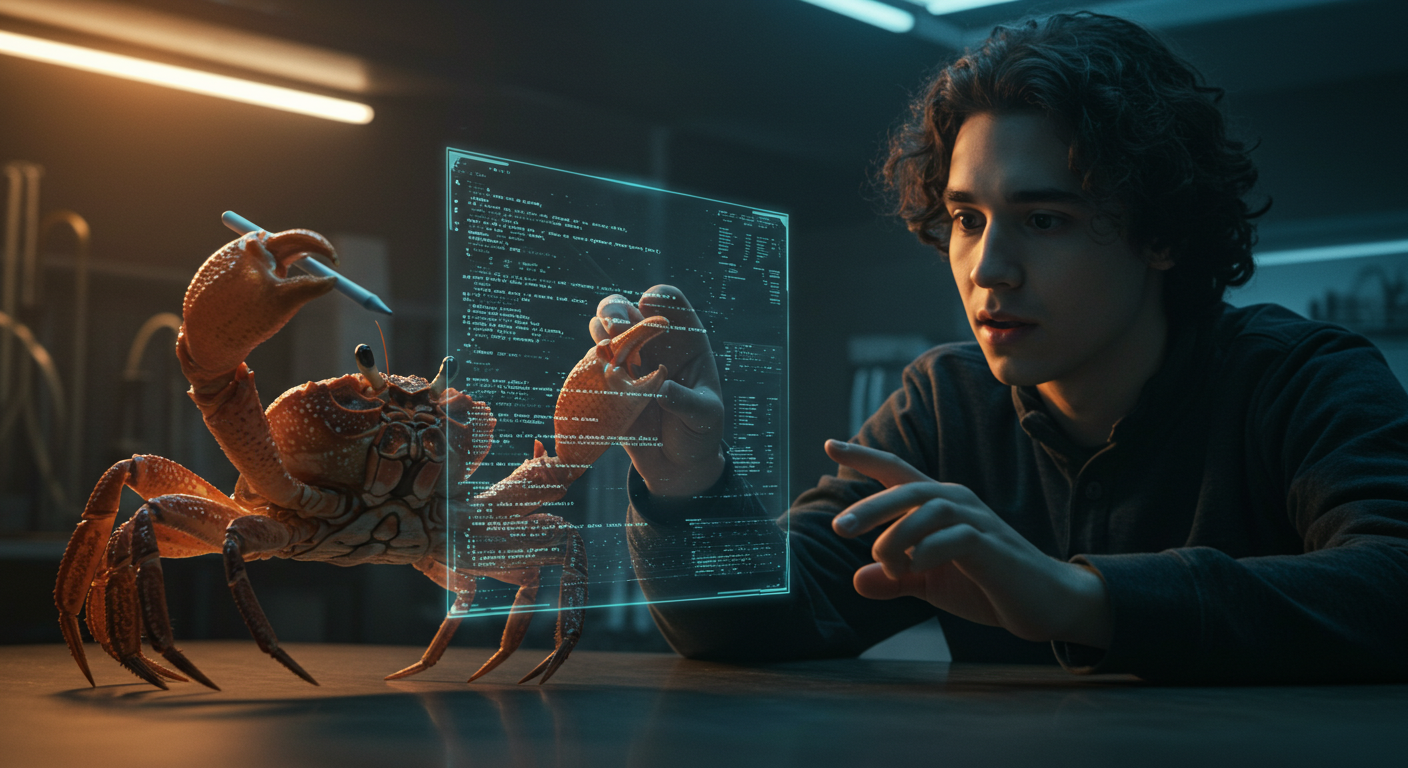
# Guia Rápido

Rust, Paradigma Funcional e Computação de Alto Desempenho

Feito para programadores

Sumário

[Fundamentos de Rust com Paradigma Funcional 4](#__RefHeading___Toc330_2765180531)

[1. Variáveis e Imutabilidade 4](#__RefHeading___Toc332_2765180531)

[2. Entrada e Saída de Dados 4](#__RefHeading___Toc334_2765180531)

[3. Estruturas de Decisão 4](#__RefHeading___Toc336_2765180531)

[4. Estruturas de Repetição 5](#__RefHeading___Toc338_2765180531)

[Iteradores com for 5](#__RefHeading___Toc340_2765180531)

[Iteradores com while 5](#__RefHeading___Toc342_2765180531)

[5. Listas (Vetores) 5](#__RefHeading___Toc344_2765180531)

[6. Dicionários (HashMap) 5](#__RefHeading___Toc346_2765180531)

[7. Funções e Closures 6](#__RefHeading___Toc348_2765180531)

[8. Manipulação de Strings 6](#__RefHeading___Toc350_2765180531)

[9. Trabalhando com Result e Option 6](#__RefHeading___Toc352_2765180531)

[Conclusão 7](#__RefHeading___Toc354_2765180531)

[Tipos de Dados em Rust 7](#__RefHeading___Toc356_2765180531)

[1. Tipos Escalares 7](#__RefHeading___Toc358_2765180531)

[2. Tipos Compostos 7](#__RefHeading___Toc360_2765180531)

[3. Tipos de Coleção (Dinamicamente Alocados) 7](#__RefHeading___Toc362_2765180531)

[4. Tipos Especiais 8](#__RefHeading___Toc364_2765180531)

[Introdução ao Paradigma Funcional em Rust 8](#__RefHeading___Toc366_2765180531)

[O que é o Paradigma Funcional? 8](#__RefHeading___Toc368_2765180531)

[Como o Paradigma Funcional se Aplica em Rust 8](#__RefHeading___Toc370_2765180531)

[1. Imutabilidade 8](#__RefHeading___Toc372_2765180531)

[2. Funções Puras 9](#__RefHeading___Toc374_2765180531)

[3. Funções de Alta Ordem 9](#__RefHeading___Toc376_2765180531)

[4. Closures 9](#__RefHeading___Toc378_2765180531)

[5. Iteradores e Composição de Funções 9](#__RefHeading___Toc380_2765180531)

[6. Recursão 9](#__RefHeading___Toc382_2765180531)

[Conclusão 10](#__RefHeading___Toc384_2765180531)

[Rust e Computação de Alto Desempenho utilizando o Paradigma Funcional 10](#__RefHeading___Toc386_2765180531)

[1. O que é Computação de Alto Desempenho? 10](#__RefHeading___Toc388_2765180531)

[2. Como Rust Trabalha para Melhor Computação de Alto Desempenho? 10](#__RefHeading___Toc390_2765180531)

[3. Exemplos Práticos 10](#__RefHeading___Toc392_2765180531)

[3.1 Processamento Paralelo com Rayon 10](#__RefHeading___Toc394_2765180531)

[3.2 Uso de Map-Reduce para Processamento Funcional 11](#__RefHeading___Toc396_2765180531)

[3.3 Uso de ndarray para Cálculos Numéricos 11](#__RefHeading___Toc398_2765180531)

[Conclusão 11](#__RefHeading___Toc400_2765180531)

[Implementando uma API em Rust com Exemplo Prático 11](#__RefHeading___Toc402_2765180531)

[Introdução 11](#__RefHeading___Toc404_2765180531)

[1. Configuração do Ambiente 12](#__RefHeading___Toc406_2765180531)

[2. Criando a API 12](#__RefHeading___Toc408_2765180531)

[2.1 Importando Módulos Necessários 12](#__RefHeading___Toc410_2765180531)

[3. Testando a API 13](#__RefHeading___Toc412_2765180531)

[3.1 Listar Itens 13](#__RefHeading___Toc414_2765180531)

[3.2 Adicionar um Item 13](#__RefHeading___Toc416_2765180531)

[Conclusão 13](#__RefHeading___Toc418_2765180531)

[Desenvolvendo Aplicações Desktop com Rust e Interface Gráfica 13](#__RefHeading___Toc420_2765180531)

[Introdução 13](#__RefHeading___Toc422_2765180531)

[1. Escolhendo um Framework GUI para Rust 13](#__RefHeading___Toc424_2765180531)

[2. Configuração do Ambiente 14](#__RefHeading___Toc426_2765180531)

[3. Criando uma Aplicação GUI Simples 14](#__RefHeading___Toc428_2765180531)

[4. Executando a Aplicação 15](#__RefHeading___Toc430_2765180531)

[5. Expansão e Próximos Passos 15](#__RefHeading___Toc432_2765180531)

[Conclusão 15](#__RefHeading___Toc434_2765180531)

[Desenvolvendo Aplicações Desktop com Rust e Interface Gráfica – parte 2 15](#__RefHeading___Toc436_2765180531)

[Introdução 15](#__RefHeading___Toc438_2765180531)

[1. Escolhendo um Framework GUI para Rust 15](#__RefHeading___Toc440_2765180531)

[2. Configuração do Ambiente 15](#__RefHeading___Toc442_2765180531)

[3. Criando uma Aplicação GUI Simples 16](#__RefHeading___Toc444_2765180531)

[4. Executando a Aplicação 17](#__RefHeading___Toc446_2765180531)

[Persistência de Dados em Rust com SQLite 17](#__RefHeading___Toc448_2765180531)

[Introdução 17](#__RefHeading___Toc450_2765180531)

[1. Configurando o Ambiente 17](#__RefHeading___Toc452_2765180531)

[2. Criando e Conectando ao Banco de Dados 17](#__RefHeading___Toc454_2765180531)

[3. Criando Tabelas 18](#__RefHeading___Toc456_2765180531)

[4. Inserindo Dados 18](#__RefHeading___Toc458_2765180531)

[5. Lendo Dados do Banco 18](#__RefHeading___Toc460_2765180531)

[6. Atualizando Dados 18](#__RefHeading___Toc462_2765180531)

[7. Deletando Dados 19](#__RefHeading___Toc464_2765180531)

[Conclusão 19](#__RefHeading___Toc466_2765180531)

[Persistência de Dados em Rust com Paradigma Funcional e SQLite 19](#__RefHeading___Toc468_2765180531)

[Introdução 19](#__RefHeading___Toc470_2765180531)

[1. Configuração do Ambiente 19](#__RefHeading___Toc472_2765180531)

[2. Criando a Conexão com o Banco 20](#__RefHeading___Toc474_2765180531)

[3. Criando a Tabela 20](#__RefHeading___Toc476_2765180531)

[4. Inserindo Dados com Função Pura 20](#__RefHeading___Toc478_2765180531)

[5. Obtendo Usuários com Iteradores Funcionais 20](#__RefHeading___Toc480_2765180531)

[6. Atualizando Usuários com Função Pura 21](#__RefHeading___Toc482_2765180531)

[7. Removendo Usuário Funcionalmente 21](#__RefHeading___Toc484_2765180531)

[8. Executando Tudo no main 21](#__RefHeading___Toc486_2765180531)

[Conclusão 21](#__RefHeading___Toc488_2765180531)

# Fundamentos de Rust com Paradigma Funcional

Rust é uma linguagem segura e eficiente, com suporte ao paradigma funcional. Este material cobre os principais fundamentos da linguagem, enfatizando boas práticas funcionais.

## 1. Variáveis e Imutabilidade

Por padrão, as variáveis em Rust são imutáveis. Isso evita efeitos colaterais e torna o código mais previsível.

fn main() {

let x = 10; // Define uma variável imutável

println!("{}", x); // Exibe o valor de x no console

}

Para permitir alteração, use mut:

fn main() {

let mut x = 10; // Define uma variável mutável

x += 5; // Modifica o valor da variável

println!("{}", x); // Exibe o novo valor de x

}

## 2. Entrada e Saída de Dados

A leitura de entrada do usuário pode ser feita com stdin():

use std::io;

fn main() {

let mut entrada = String::new(); // Cria uma string vazia

io::stdin().read\_line(&mut entrada).expect("Erro ao ler linha"); // Lê entrada do teclado

println!("Voce digitou: {}", entrada.trim()); // Exibe a entrada removendo espaços extras

}

## 3. Estruturas de Decisão

O if em Rust funciona de forma similar a outras linguagens:

fn main() {

let numero = 10;

if numero > 5 {

println!("Maior que 5");

} else {

println!("Menor ou igual a 5");

}

}

O if pode ser usado como expressão:

fn main() {

let numero = 10;

let mensagem = if numero > 5 { "Maior" } else { "Menor" };

println!("{}", mensagem); // Exibe "Maior"

}

## 4. Estruturas de Repetição

### Iteradores com for

Percorre um intervalo de valores:

fn main() {

for x in 1..=5 {

println!("{}", x); // Exibe valores de 1 a 5

}

}

### Iteradores com while

Executa enquanto a condição for verdadeira:

fn main() {

let mut x = 0;

while x < 5 {

println!("{}", x);

x += 1;

}

}

## 5. Listas (Vetores)

Vetores são coleções de elementos do mesmo tipo:

fn main() {

let numeros = vec![1, 2, 3, 4, 5]; // Cria um vetor

for n in &numeros {

println!("{}", n); // Itera e exibe os elementos

}

}

Utilizando map para transformar elementos:

fn main() {

let numeros = vec![1, 2, 3, 4, 5];

let dobrados: Vec<i32> = numeros.iter().map(|x| x \* 2).collect();

println!("{:?}", dobrados); // Exibe [2, 4, 6, 8, 10]

}

## 6. Dicionários (HashMap)

O HashMap permite associar chaves a valores:

use std::collections::HashMap;

fn main() {

let mut dicionario = HashMap::new();

dicionario.insert("chave1", "valor1");

dicionario.insert("chave2", "valor2");

for (chave, valor) in &dicionario {

println!("{}: {}", chave, valor); // Exibe as chaves e valores

}

}

## 7. Funções e Closures

Funções são definidas com fn:

fn soma(a: i32, b: i32) -> i32 {

a + b // Retorna a soma dos números

}

fn main() {

println!("{}", soma(2, 3)); // Exibe 5

}

Closures são funções anônimas:

fn main() {

let quadrado = |x: i32| x \* x;

println!("{}", quadrado(4)); // Exibe 16

}

## 8. Manipulação de Strings

O String é uma versão dinamicamente alocada de string:

fn main() {

let texto = String::from("Ola, Rust!");

let tamanho = texto.len();

println!("Tamanho: {}", tamanho); // Exibe o tamanho da string

}

## 9. Trabalhando com Result e Option

O Option<T> representa valores opcionais:

fn dividir(a: i32, b: i32) -> Option<i32> {

if b == 0 {

None

} else {

Some(a / b)

}

}

fn main() {

match dividir(10, 2) {

Some(resultado) => println!("Resultado: {}", resultado),

None => println!("Divisão por zero"),

}

}

O Result<T, E> trata erros explicitamente:

fn dividir(a: i32, b: i32) -> Result<i32, &'static str> {

if b == 0 {

Err("Divisão por zero")

} else {

Ok(a / b)

}

}

fn main() {

match dividir(10, 0) {

Ok(resultado) => println!("Resultado: {}", resultado),

Err(erro) => println!("Erro: {}", erro),

}

}

## Conclusão

Rust permite programar de forma funcional ao priorizar imutabilidade, uso de iteradores e funções puras. Esses conceitos ajudam a escrever código mais seguro e eficiente.

# Tipos de Dados em Rust

Rust possui um sistema de tipos forte e estático, suportando os seguintes tipos de variáveis:

### ****1. Tipos Escalares****

São valores únicos armazenados em uma variável.

* **Números inteiros (**i8**,** i16**,** i32**,** i64**,** i128**,** isize**)** – Representam números inteiros com sinal.
* **Números inteiros sem sinal (**u8**,** u16**,** u32**,** u64**,** u128**,** usize**)** – Apenas números positivos.
* **Números de ponto flutuante (**f32**,** f64**)** – Representam números decimais.
* **Booleanos (**bool**)** – true ou false.
* **Caracteres (**char**)** – Representam um único caractere Unicode (por exemplo, 'A', 'é', '🚀').

### ****2. Tipos Compostos****

Podem armazenar múltiplos valores.

* **Tuplas (**(T1, T2, ...)**)** – Contêm múltiplos valores de diferentes tipos.

let pessoa: (&str, i32, f64) = ("João", 30, 1.75);

* **Arrays (**[T; N]**)** – Contêm múltiplos valores do mesmo tipo.

let numeros: [i32; 3] = [1, 2, 3];

### ****3. Tipos de Coleção (Dinamicamente Alocados)****

* **Vetores (**Vec<T>**)** – Arrays redimensionáveis.

let mut v = vec![1, 2, 3];

v.push(4);

* **String (**String**)** – Cadeias de caracteres dinâmicas.

let texto = String::from("Olá, Rust!");

* **HashMap (**HashMap<K, V>**)** – Dicionários para mapear chaves a valores.

use std::collections::HashMap;

let mut mapa = HashMap::new();

mapa.insert("chave", 42);

### ****4. Tipos Especiais****

* Option<T> – Representa um valor opcional (Some(valor) ou None).

let talvez\_numero: Option<i32> = Some(10);

* Result<T, E> – Representa operações que podem falhar (Ok(valor) ou Err(erro)).

fn dividir(a: i32, b: i32) -> Result<i32, &'static str> {

if b == 0 {

Err("Divisão por zero")

} else {

Ok(a / b)

}

}

# Introdução ao Paradigma Funcional em Rust

Rust é uma linguagem moderna que combina aspectos do paradigma funcional, imperativo e orientado a objetos. Se você nunca estudou o paradigma funcional antes, este material o guiará pelos conceitos fundamentais e como eles são aplicados em Rust.

## O que é o Paradigma Funcional?

O paradigma funcional enfatiza a imutabilidade, funções puras e a composição de funções. Alguns conceitos-chave incluem:

1. **Funções Puras**: Uma função pura retorna o mesmo resultado para os mesmos argumentos e não causa efeitos colaterais.
2. **Imutabilidade**: Variáveis não são alteradas depois de definidas.
3. **Funções de Alta Ordem**: Funções que aceitam outras funções como argumento ou retornam funções.
4. **Closures**: Funções anônimas que capturam variáveis do escopo.
5. **Recursão**: Substitui laços por chamadas recursivas.

## Como o Paradigma Funcional se Aplica em Rust

### 1. Imutabilidade

Por padrão, Rust define variáveis como imutáveis. Isso impede efeitos colaterais inesperados.

fn main() {

let x = 5;

// x = 6; // Erro: não é possível modificar x

println!("{}", x);

}

Para permitir alteração, usamos mut:

fn main() {

let mut x = 5;

x = 6;

println!("{}", x);

}

### 2. Funções Puras

Uma função pura não modifica o estado global.

fn soma(a: i32, b: i32) -> i32 {

a + b

}

fn main() {

println!("{}", soma(2, 3));

}

### 3. Funções de Alta Ordem

Podemos passar funções como argumentos:

fn aplica<F>(f: F, x: i32) -> i32

where

F: Fn(i32) -> i32,

{

f(x)

}

fn dobrar(x: i32) -> i32 {

x \* 2

}

fn main() {

let resultado = aplica(dobrar, 5);

println!("{}", resultado);

}

### 4. Closures

Rust permite criar funções anônimas:

fn main() {

let quadrado = |x: i32| x \* x;

println!("{}", quadrado(4));

}

### 5. Iteradores e Composição de Funções

Rust usa iteradores para substituir loops mutáveis:

fn main() {

let numeros = vec![1, 2, 3, 4, 5];

let dobro: Vec<i32> = numeros.iter().map(|x| x \* 2).collect();

println!("{:?}", dobro);

}

### 6. Recursão

Evita mutabilidade substituindo loops por chamadas recursivas:

fn fatorial(n: u32) -> u32 {

if n == 0 { 1 } else { n \* fatorial(n - 1) }

}

fn main() {

println!("{}", fatorial(5));

}

## Conclusão

Rust não é uma linguagem puramente funcional, mas suporta muitos conceitos do paradigma funcional, permitindo um código mais seguro e expressivo. Praticar esses conceitos ajudará você a escrever código mais eficiente e menos propenso a erros.

# Rust e Computação de Alto Desempenho utilizando o Paradigma Funcional

## 1. O que é Computação de Alto Desempenho?

Computação de Alto Desempenho (HPC - High-Performance Computing) refere-se ao uso de sistemas computacionais poderosos para resolver problemas complexos e exigentes em termos de processamento. Isso envolve:

* **Processamento Paralelo**: Execução simultânea de múltiplas tarefas.
* **Eficiência de Memória**: Uso otimizado de cache e alocação eficiente de recursos.
* **Baixa Latência**: Redução do tempo de resposta.
* **Execução Otimizada**: Cálculos altamente otimizados para obter o máximo desempenho.

Rust, sendo uma linguagem moderna e de baixo nível, oferece recursos que a tornam ideal para aplicações HPC, especialmente dentro do paradigma funcional.

## 2. Como Rust Trabalha para Melhor Computação de Alto Desempenho?

Rust combina a segurança de memória com desempenho próximo ao de C e C++. Os principais fatores que contribuem para sua eficiência em HPC incluem:

* **Gerenciamento Seguro de Memória**: Sem necessidade de garbage collector, evitando pausas na execução.
* **Imutabilidade por Padrão**: Garante maior previsibilidade na execução do código.
* **Zero-Cost Abstractions**: Permite que as abstrações de alto nível sejam eliminadas pelo compilador em tempo de execução.
* **Uso de Iteradores e Map-Reduce**: Abordagem funcional que reduz a necessidade de loops mutáveis.
* **Compilador LLVM**: Garante otimizações avançadas para execução eficiente do código.

## 3. Exemplos Práticos

### 3.1 Processamento Paralelo com Rayon

Rayon é uma biblioteca que permite processamento paralelo de forma simples e segura. O uso de par\_iter permite distribuir tarefas automaticamente em múltiplos núcleos.

use rayon::prelude::\*;

fn main() {

let numeros: Vec<i32> = (1..=1\_000\_000).collect();

let soma: i32 = numeros.par\_iter().map(|x| x \* 2).sum();

println!("Soma dos números dobrados: {}", soma);

}

### 3.2 Uso de Map-Reduce para Processamento Funcional

O paradigma funcional enfatiza o uso de operações como map e reduce para manipulação eficiente de dados sem mutabilidade.

fn main() {

let numeros = vec![1, 2, 3, 4, 5];

let dobrados: Vec<i32> = numeros.iter().map(|x| x \* 2).collect();

let soma: i32 = dobrados.iter().sum();

println!("Dobrados: {:?}, Soma: {}", dobrados, soma);

}

### 3.3 Uso de ndarray para Cálculos Numéricos

A biblioteca ndarray facilita operações matriciais e vetoriais, essenciais para aplicações HPC como Machine Learning e simulações científicas.

use ndarray::Array2;

fn main() {

let a = Array2::from\_shape\_vec((2, 2), vec![1, 2, 3, 4]).unwrap();

let b = Array2::from\_shape\_vec((2, 2), vec![5, 6, 7, 8]).unwrap();

let c = a.dot(&b);

println!("Produto Matricial:\n{:?}", c);

}

## Conclusão

Rust se destaca para Computação de Alto Desempenho ao oferecer segurança, eficiência e suporte ao paradigma funcional. O uso de abstrações zero-cost, paralelismo seguro e otimizações do compilador permitem um código altamente eficiente, ideal para aplicações científicas, financeiras e de Big Data. A combinação de iteradores, paralelismo e estruturas como ndarray proporciona uma abordagem funcional poderosa para HPC.

# Implementando uma API em Rust com Exemplo Prático

## Introdução

Rust é uma linguagem segura, rápida e eficiente, ideal para desenvolvimento de APIs de alto desempenho. Neste artigo, vamos demonstrar como criar uma API REST em Rust utilizando o framework **Axum**, que é moderno e fácil de usar.

## 1. Configuração do Ambiente

Antes de começar, certifique-se de ter o Rust instalado. Se ainda não tiver, instale utilizando o Rustup:

curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

Agora, crie um novo projeto Rust:

cargo new api\_rust

cd api\_rust

Adicione as dependências no Cargo.toml:

[dependencies]

axum = "0.6"

tokio = { version = "1", features = ["full"] }

serde = { version = "1", features = ["derive"] }

serde\_json = "1"

## 2. Criando a API

Agora, vamos desenvolver nossa API em src/main.rs.

### 2.1 Importando Módulos Necessários

use axum::{Router, routing::{get, post}, Json};

use serde::{Deserialize, Serialize};

use std::net::SocketAddr;

use tokio::net::TcpListener;

#[derive(Serialize, Deserialize)]

struct Item {

id: u32,

name: String,

}

async fn get\_items() -> Json<Vec<Item>> {

let items = vec![

Item { id: 1, name: "Item 1".to\_string() },

Item { id: 2, name: "Item 2".to\_string() },

];

Json(items)

}

async fn add\_item(Json(payload): Json<Item>) -> Json<Item> {

Json(payload)

}

#[tokio::main]

async fn main() {

let app = Router::new()

.route("/items", get(get\_items))

.route("/items", post(add\_item));

let addr = SocketAddr::from(([127, 0, 0, 1], 3000));

println!("Servidor rodando em http://{}", addr);

axum::Server::bind(&addr)

.serve(app.into\_make\_service())

.await

.unwrap();

}

## 3. Testando a API

Execute a API com:

cargo run

Agora, faça requisições para testar os endpoints.

### 3.1 Listar Itens

curl http://127.0.0.1:3000/items

### 3.2 Adicionar um Item

curl -X POST http://127.0.0.1:3000/items -H "Content-Type: application/json" -d '{"id": 3, "name": "Item 3"}'

## Conclusão

Neste artigo, criamos uma API simples em Rust usando Axum. Com essa base, é possível expandir para incluir banco de dados, autenticação e mais funcionalidades para aplicações robustas.

A configuração do ambiente no artigo funciona tanto para Linux quanto para Windows e macOS, pois o **Rustup** é multiplataforma. Se você estiver no Windows, pode instalar o Rust pelo **Rustup-Init.exe** disponível no site oficial:

🔗 <https://rustup.rs/>

No Windows, o comando curl pode não estar disponível no terminal padrão. Se for o caso, basta baixar e rodar o instalador manualmente.

Além disso, se estiver usando Windows, recomendo utilizar o **PowerShell** ou **WSL (Windows Subsystem for Linux)** para uma experiência mais fluida ao rodar comandos.

# Desenvolvendo Aplicações Desktop com Rust e Interface Gráfica

## Introdução

Rust é uma linguagem conhecida por sua segurança e desempenho, sendo amplamente utilizada em sistemas de alto desempenho e back-end. No entanto, também é possível desenvolver aplicações desktop com interface gráfica (GUI). Este artigo explora como configurar um ambiente para desenvolvimento de aplicações desktop em Rust e implementa um exemplo prático utilizando o framework **Druid**.

## 1. Escolhendo um Framework GUI para Rust

Rust não possui uma biblioteca GUI nativa, mas há diversas opções de terceiros, incluindo:

* **Druid** – Simples, moderno e multiplataforma.
* **GTK-rs** – Ideal para aplicações que utilizam GTK.
* **Slint** – Focado em aplicações leves.
* **egui** – Para aplicações rápidas e interativas.
* **Tauri** – Para aplicações híbridas com frontend em JavaScript e backend em Rust.

Neste artigo, utilizaremos **Druid**, pois ele é fácil de configurar e mantém a filosofia segura do Rust.

## 2. Configuração do Ambiente

Antes de começar, certifique-se de ter Rust instalado. Se ainda não tiver, instale com o Rustup:

curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

Agora, crie um novo projeto:

cargo new rust\_desktop\_app

cd rust\_desktop\_app

Adicione a dependência do **Druid** no arquivo Cargo.toml:

[dependencies]

druid = "0.7"

## 3. Criando uma Aplicação GUI Simples

Agora, edite o arquivo src/main.rs e adicione o seguinte código:

use druid::widget::{Button, Flex, Label};

use druid::{AppLauncher, Data, Env, Lens, LocalizedString, Widget, WidgetExt, WindowDesc};

#[derive(Clone, Data, Lens)]

struct AppState {

count: i32,

}

fn build\_ui() -> impl Widget<AppState> {

let label = Label::new(|data: &AppState, \_: &Env| format!("Contador: {}", data.count));

let button = Button::new("Incrementar")

.on\_click(|\_, data: &mut AppState, \_| data.count += 1);

Flex::column()

.with\_child(label)

.with\_spacer(10.0)

.with\_child(button)

}

fn main() {

let main\_window = WindowDesc::new(build\_ui)

.title(LocalizedString::new("Rust GUI App"));

let initial\_state = AppState { count: 0 };

AppLauncher::with\_window(main\_window)

.launch(initial\_state)

.expect("Falha ao iniciar a aplicação");

}

## 4. Executando a Aplicação

Para rodar a aplicação, utilize o comando:

cargo run

Isso abrirá uma janela com um botão e um contador que se incrementa ao ser pressionado.

## 5. Expansão e Próximos Passos

Agora que você tem um projeto funcional, pode expandi-lo com:

* **Novos componentes UI** (como caixas de texto e menus);
* **Persistência de dados** (utilizando SQLite ou JSON);
* **Suporte multiplataforma** para Windows, Linux e macOS.

## Conclusão

Rust pode ser utilizado para desenvolver aplicações desktop com interface gráfica de maneira segura e eficiente. Com frameworks como Druid, é possível criar interfaces modernas e responsivas com código conciso e idiomático.

# Desenvolvendo Aplicações Desktop com Rust e Interface Gráfica – parte 2

## Introdução

Rust é uma linguagem conhecida por sua segurança e desempenho, sendo amplamente utilizada em sistemas de alto desempenho e back-end. No entanto, também é possível desenvolver aplicações desktop com interface gráfica (GUI). Este artigo explora como configurar um ambiente para desenvolvimento de aplicações desktop em Rust e implementa um exemplo prático utilizando o framework **Druid**.

## 1. Escolhendo um Framework GUI para Rust

Rust não possui uma biblioteca GUI nativa, mas há diversas opções de terceiros, incluindo:

* **Druid** – Simples, moderno e multiplataforma.
* **GTK-rs** – Ideal para aplicações que utilizam GTK.
* **Slint** – Focado em aplicações leves.
* **egui** – Para aplicações rápidas e interativas.
* **Tauri** – Para aplicações híbridas com frontend em JavaScript e backend em Rust.

Neste artigo, utilizaremos **Druid**, pois ele é fácil de configurar e mantém a filosofia segura do Rust.

## 2. Configuração do Ambiente

Antes de começar, certifique-se de ter Rust instalado. Se ainda não tiver, instale com o Rustup:

curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

Agora, crie um novo projeto:

cargo new rust\_desktop\_app

cd rust\_desktop\_app

Adicione a dependência do **Druid** no arquivo Cargo.toml:

[dependencies]

druid = "0.7"

## 3. Criando uma Aplicação GUI Simples

Agora, edite o arquivo src/main.rs e adicione o seguinte código:

use druid::widget::{Button, Flex, Label, TextBox, Slider, Checkbox, List, Scroll};

use druid::{AppLauncher, Data, Env, Lens, LocalizedString, Widget, WidgetExt, WindowDesc};

#[derive(Clone, Data, Lens)]

struct AppState {

count: i32,

text: String,

slider\_value: f64,

checkbox\_value: bool,

}

fn build\_ui() -> impl Widget<AppState> {

let label = Label::new(|data: &AppState, \_: &Env| format!("Contador: {}", data.count));

let button = Button::new("Incrementar")

.on\_click(|\_, data: &mut AppState, \_| data.count += 1);

let textbox = TextBox::new().lens(AppState::text);

let slider = Slider::new().lens(AppState::slider\_value);

let checkbox = Checkbox::new("Opção selecionada").lens(AppState::checkbox\_value);

let list = List::new(|| Label::new(|item: &String, \_: &Env| format!("Item: {}", item)));

let scrollable\_list = Scroll::new(list);

Flex::column()

.with\_child(label)

.with\_spacer(10.0)

.with\_child(button)

.with\_spacer(10.0)

.with\_child(textbox)

.with\_spacer(10.0)

.with\_child(slider)

.with\_spacer(10.0)

.with\_child(checkbox)

.with\_spacer(10.0)

.with\_child(scrollable\_list)

}

fn main() {

let main\_window = WindowDesc::new(build\_ui)

.title(LocalizedString::new("Rust GUI App"));

let initial\_state = AppState {

count: 0,

text: "Digite algo".to\_string(),

slider\_value: 0.5,

checkbox\_value: false,

};

AppLauncher::with\_window(main\_window)

.launch(initial\_state)

.expect("Falha ao iniciar a aplicação");

}

## 4. Executando a Aplicação

Para rodar a aplicação, utilize o comando:

cargo run

Isso abrirá uma janela com um botão, um campo de texto, um controle deslizante, um checkbox e uma lista rolável.

# Persistência de Dados em Rust com SQLite

## Introdução

A persistência de dados é essencial para muitas aplicações, permitindo armazenar informações permanentemente. Em Rust, podemos usar o SQLite, um banco de dados leve e eficiente. Neste artigo, vamos aprender como configurar um ambiente de persistência de dados em Rust utilizando a biblioteca rusqlite.

## 1. Configurando o Ambiente

Antes de começar, certifique-se de ter Rust instalado. Se ainda não tiver, instale-o com Rustup:

curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

Agora, crie um novo projeto Rust:

cargo new rust\_sqlite

cd rust\_sqlite

Adicione a dependência do SQLite no arquivo Cargo.toml:

[dependencies]

rusqlite = { version = "0.29", features = ["bundled"] }

## 2. Criando e Conectando ao Banco de Dados

Vamos criar um banco de dados SQLite e conectar a ele:

use rusqlite::{Connection, Result};

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

let conn = Connection::open("meu\_banco.db")?;

println!("Conectado ao banco de dados!");

Ok(())

}

Esse código cria um arquivo meu\_banco.db no diretório do projeto se ele não existir.

## 3. Criando Tabelas

Agora, vamos criar uma tabela chamada usuarios:

use rusqlite::{Connection, Result};

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

let conn = Connection::open("meu\_banco.db")?;

conn.execute(

"CREATE TABLE IF NOT EXISTS usuarios (

id INTEGER PRIMARY KEY,

nome TEXT NOT NULL,

email TEXT UNIQUE NOT NULL

)",

[],

)?;

println!("Tabela criada com sucesso!");

Ok(())

}

## 4. Inserindo Dados

Vamos inserir um novo usuário na tabela:

use rusqlite::{params, Connection, Result};

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

let conn = Connection::open("meu\_banco.db")?;

conn.execute(

"INSERT INTO usuarios (nome, email) VALUES (?1, ?2)",

params!["João Silva", "joao@email.com"],

)?;

println!("Usuário inserido com sucesso!");

Ok(())

}

## 5. Lendo Dados do Banco

Agora, vamos buscar e exibir os dados da tabela usuarios:

use rusqlite::{params, Connection, Result};

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

let conn = Connection::open("meu\_banco.db")?;

let mut stmt = conn.prepare("SELECT id, nome, email FROM usuarios")?;

let usuarios = stmt.query\_map([], |row| {

Ok((row.get::<\_, i32>(0)?, row.get::<\_, String>(1)?, row.get::<\_, String>(2)?))

})?;

for usuario in usuarios {

let (id, nome, email) = usuario?;

println!("ID: {}, Nome: {}, Email: {}", id, nome, email);

}

Ok(())

}

## 6. Atualizando Dados

Podemos atualizar um usuário existente:

use rusqlite::{params, Connection, Result};

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

let conn = Connection::open("meu\_banco.db")?;

conn.execute(

"UPDATE usuarios SET email = ?1 WHERE nome = ?2",

params!["novoemail@email.com", "João Silva"],

)?;

println!("Usuário atualizado com sucesso!");

Ok(())

}

## 7. Deletando Dados

Podemos remover um usuário pelo ID:

use rusqlite::{params, Connection, Result};

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

let conn = Connection::open("meu\_banco.db")?;

conn.execute("DELETE FROM usuarios WHERE id = ?1", params![1])?;

println!("Usuário removido com sucesso!");

Ok(())

}

## Conclusão

Rust oferece uma integração robusta com SQLite usando rusqlite, permitindo persistência de dados eficiente e segura. Com este conhecimento, você pode expandir para projetos maiores e integrar SQLite em suas aplicações Rust.

# Persistência de Dados em Rust com Paradigma Funcional e SQLite

## Introdução

O paradigma funcional enfatiza **imutabilidade, funções puras e composição**. Em Rust, embora não seja uma linguagem puramente funcional, podemos adotar esse paradigma para tornar nosso código mais seguro e previsível. Neste artigo, exploramos como implementar persistência de dados utilizando **SQLite com rusqlite**, seguindo um estilo funcional.

## 1. Configuração do Ambiente

Primeiro, certifique-se de ter o **Rust** instalado. Caso não tenha, instale com:

curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

Agora, crie um novo projeto Rust:

cargo new rust\_sqlite\_funcional

cd rust\_sqlite\_funcional

Adicione a dependência no Cargo.toml:

[dependencies]

rusqlite = { version = "0.29", features = ["bundled"] }

## 2. Criando a Conexão com o Banco

Vamos definir uma função pura para abrir uma conexão com SQLite de forma imutável:

use rusqlite::{Connection, Result};

fn conectar\_banco() -> Result<Connection> {

Connection::open("banco\_funcional.db")

}

Essa função retorna uma conexão sem modificar estado global.

## 3. Criando a Tabela

Em vez de executar a criação da tabela diretamente, criamos uma **função pura** que recebe a conexão como parâmetro:

fn criar\_tabela(conn: &Connection) -> Result<()> {

conn.execute(

"CREATE TABLE IF NOT EXISTS usuarios (

id INTEGER PRIMARY KEY,

nome TEXT NOT NULL,

email TEXT UNIQUE NOT NULL

)",

[],

)?;

Ok(())

}

## 4. Inserindo Dados com Função Pura

Criamos uma função imutável para inserir usuários sem modificar variáveis globais:

use rusqlite::params;

fn inserir\_usuario(conn: &Connection, nome: &str, email: &str) -> Result<()> {

conn.execute(

"INSERT INTO usuarios (nome, email) VALUES (?1, ?2)",

params![nome, email],

)?;

Ok(())

}

## 5. Obtendo Usuários com Iteradores Funcionais

Utilizamos **iteradores** para tornar a busca mais funcional:

fn listar\_usuarios(conn: &Connection) -> Result<Vec<(i32, String, String)>> {

let mut stmt = conn.prepare("SELECT id, nome, email FROM usuarios")?;

let usuarios = stmt.query\_map([], |row| {

Ok((row.get(0)?, row.get(1)?, row.get(2)?))

})?;

Ok(usuarios.filter\_map(Result::ok).collect())

}

Aqui, filter\_map(Result::ok) remove erros silenciosamente e collect() transforma o iterador em um vetor.

## 6. Atualizando Usuários com Função Pura

fn atualizar\_email(conn: &Connection, id: i32, novo\_email: &str) -> Result<()> {

conn.execute(

"UPDATE usuarios SET email = ?1 WHERE id = ?2",

params![novo\_email, id],

)?;

Ok(())

}

## 7. Removendo Usuário Funcionalmente

fn remover\_usuario(conn: &Connection, id: i32) -> Result<()> {

conn.execute("DELETE FROM usuarios WHERE id = ?1", params![id])?;

Ok(())

}

## 8. Executando Tudo no main

Agora, combinamos todas as funções mantendo a imutabilidade e composição funcional:

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

let conn = conectar\_banco()?;

criar\_tabela(&conn)?;

inserir\_usuario(&conn, "Alice", "alice@email.com")?;

inserir\_usuario(&conn, "Bob", "bob@email.com")?;

let usuarios = listar\_usuarios(&conn)?;

for (id, nome, email) in usuarios {

println!("ID: {}, Nome: {}, Email: {}", id, nome, email);

}

atualizar\_email(&conn, 1, "alice@novoemail.com")?;

remover\_usuario(&conn, 2)?;

Ok(())

}

## Conclusão

Esse código adota **princípios funcionais** como **imutabilidade, uso de funções puras e iteradores**. Isso torna o código mais previsível e modular, facilitando testes e manutenção. Rust permite escrever código funcional sem sacrificar desempenho, tornando-o ideal para persistência de dados segura e eficiente.

# Sobre o Autor

Gilberto Aparecido Bernardo Junior

Profissional com 30 anos de atuação na área de tecnologia da informação, onde tive a oportunidade de desempenhar diferentes funções, como instrutor, desenvolvedor de sistemas, analista e gestor de infraestrutura. Ao longo da carreira, pude contribuir em empresas de diversos portes e segmentos, além de ter a chance de empreender, fundando minha própria empresa. Atualmente, tenho o prazer de compartilhar conhecimento ministrando cursos de formação profissional em uma instituição reconhecida.

Habilidades e experiências:

\* Vivência em desenvolvimento de sistemas, incluindo análise, programação e implementação.

\* Experiência em gestão de projetos e infraestrutura de TI.

\* Capacidade de colaborar em equipe e contribuir com liderança quando necessário.

\* Habilidades de comunicação e ensino, sempre buscando aprimorar a forma de transmitir conhecimento.

\* Passagem por diversos setores, como educação, saúde e odontologia, o que me permitiu aprender e crescer profissionalmente.

Acredito que o aprendizado contínuo e a troca de experiências são fundamentais para o desenvolvimento pessoal e profissional.