Roteiro de Estudos de Rust para Sistemas Distribuídos

Introdução

Este roteiro foi desenvolvido especificamente para preparar você para o curso de **Sistemas Distribuídos e Computação Paralela** usando a linguagem Rust. Rust foi escolhida por sua capacidade única de garantir segurança de memória e concorrência sem sacrificar performance, tornando-a ideal para sistemas de alto desempenho e distribuídos.

O roteiro está dividido em **fases progressivas** que correspondem às unidades do curso. Cada fase inclui conceitos essenciais de Rust, recursos de aprendizagem e exercícios práticos. Espera-se que você dedique aproximadamente **20-30 horas** ao longo de 3-4 semanas antes do início do curso para completar as fases fundamentais (Fases 0-2).

Fase 0: Preparação e Ambiente

Objetivos

- Instalar o ecossistema Rust
- Familiarizar-se com as ferramentas básicas
- Escrever e executar seu primeiro programa

Instalação

Linux/macOS:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source $HOME/.cargo/env
```

Windows: Baixe e execute o instalador em rustup.rs

Verificação:

```
rustc --version
cargo --version
```

Ferramentas Essenciais

Ferramenta	Propósito	Comando
`cargo`	Gerenciador de pacotes e build	`cargo new projeto`
`rustc`	Compilador Rust	`rustc main.rs`
`rustfmt`	Formatador de código	`cargo fmt`
`clippy`	Linter para boas práticas	`cargo clippy`
`rust-analyzer`	Language server (IDE)	Instalar no VS Code/ editor

Primeiro Programa

Crie um novo projeto e execute:

```
cargo new hello_rust
cd hello_rust
cargo run
```

Exercício: Modifique `src/main.rs` para ler seu nome da linha de comando e imprimir uma saudação personalizada.

Recursos

- Instalação Oficial
- Cargo Book

Fase 1: Fundamentos de Rust

Objetivos

- Compreender ownership, borrowing e lifetimes (conceitos únicos de Rust)
- Dominar tipos de dados básicos e controle de fluxo
- Trabalhar com structs, enums e pattern matching

Tópicos Essenciais

1.1 Ownership e Borrowing

Ownership é o conceito central de Rust. Cada valor tem um único "dono" e é liberado quando o dono sai de escopo.

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1; // s1 foi "movido" para s2
    // println!("{}", s1); // ERRO: s1 não é mais válido
    println!("{}", s2); // OK
}
```

Borrowing permite referências temporárias sem transferir ownership:

```
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1); // Empresta s1
    println!("'{}' tem {} caracteres", s1, len); // s1 ainda é
        válido
}

fn calculate_length(s: &String) -> usize {
    s.len()
} // s sai de escopo, mas não é liberada (apenas emprestada)
```

Regras de Borrowing: 1. Você pode ter **uma** referência mutável OU **várias** referências imutáveis 2. Referências devem sempre ser válidas (não podem apontar para memória liberada)

1.2 Structs e Enums

```
// Struct para representar um ponto
struct Point {
    x: f64,
    y: f64,
}

impl Point {
    fn new(x: f64, y: f64) -> Self {
        Point { x, y }
    }

    fn distance(&self, other: &Point) -> f64 {
```

```
        ((self.x - other.x).powi(2) + (self.y -
        other.y).powi(2)).sqrt()
    }
}

// Enum para representar estados
enum ConnectionState {
    Connected,
    Disconnected,
    Error(String),
}
```

1.3 Pattern Matching

```
fn handle_state(state: ConnectionState) {
    match state {
        ConnectionState::Connected => println!("Conectado!"),
        ConnectionState::Disconnected => println!("Desconectado"),
        ConnectionState::Error(msg) => println!("Erro: {}", msg),
    }
}
```

Exercícios Práticos

1. **Ownership:** Implemente uma função que recebe uma `String`, adiciona um sufixo e retorna a string modificada sem fazer cópias desnecessárias.

2. **Borrowing:** Crie uma struct `Rectangle` com campos `width` e `height`. Implemente métodos para calcular área e perímetro usando `&self`.

3. **Enums:** Crie um enum `Message` que pode representar diferentes tipos de mensagens (Text, Image, Video) com dados associados. Implemente uma função que processa cada tipo.

Recursos

- The Rust Book - Capítulos 3, 4, 5, 6
- Rust by Example - Ownership
- Visualizando Ownership (vídeo)

Fase 2: Tratamento de Erros e Coleções

Objetivos

- Trabalhar com `Result` e `Option` para tratamento de erros
- Usar coleções (Vec, HashMap, HashSet)
- Compreender iteradores e closures

Tópicos Essenciais

2.1 Result e Option

Rust não tem exceções. Erros são valores:

```rust
use std::fs::File;
use std::io::Read;

fn read_file(path: &str) -> Result<String, std::io::Error> {
    let mut file = File::open(path)?; // ? propaga erros
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    Ok(contents)
}

fn main() {
    match read_file("data.txt") {
        Ok(contents) => println!("Conteúdo: {}", contents),
        Err(e) => eprintln!("Erro ao ler arquivo: {}", e),
    }
}
```

2.2 Coleções

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

fn main() {
    // Vec (vetor dinâmico)
    let mut numbers = vec![1, 2, 3];
    numbers.push(4);

    // HashMap (dicionário)
    let mut scores = HashMap::new();
    scores.insert("Alice", 100);
```

```
    scores.insert("Bob", 85);

    // Iteração
    for (name, score) in &scores {
        println!("{}: {}", name, score);
    }
}
```

2.3 Iteradores e Closures

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    // Closure (função anônima)
    let sum: i32 = numbers.iter()
        .filter(|&x| x % 2 == 0)  // Apenas pares
        .map(|x| x * 2)            // Dobra o valor
        .sum();                    // Soma tudo

    println!("Soma: {}", sum); // 12
}
```

Exercícios Práticos

1. **Result:** Implemente uma calculadora que lê dois números e uma operação da linha de comando. Retorne `Result<f64, String>` para tratar divisão por zero e operações inválidas.

2. **Coleções:** Crie um programa que lê uma lista de palavras e conta a frequência de cada palavra usando `HashMap`.

3. **Iteradores:** Dada uma lista de números, use iteradores para encontrar a média dos números maiores que 10.

Recursos

- The Rust Book - Capítulos 8, 9, 13
- Rust by Example - Error Handling

Fase 3: Concorrência com Threads

Objetivos

- Criar e gerenciar threads
- Compartilhar dados entre threads com segurança usando `Arc` e `Mutex`
- Compreender `Send` e `Sync` traits

Tópicos Essenciais

3.1 Criando Threads

```rust
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("Thread: {}", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("Main: {}", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap(); // Espera a thread terminar
}
```

3.2 Compartilhando Dados: Arc e Mutex

Problema: Como compartilhar um contador entre threads?

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    // Arc = Atomic Reference Counted (ponteiro compartilhado
        thread-safe)
    // Mutex = Mutual Exclusion (lock para acesso exclusivo)
    let counter = Arc::new(Mutex::new(0));
```

```
    let mut handles = vec![];

    for _ in 0..10 {
        let counter_clone = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter_clone.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Resultado: {}", *counter.lock().unwrap()); // 10
}
```

Por que `Arc<Mutex<T>>`? - `Arc`: Permite múltiplas threads terem ownership compartilhada - `Mutex`: Garante que apenas uma thread acessa o dado por vez

3.3 Channels para Comunicação

```
use std::sync::mpsc; // multiple producer, single consumer
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("Olá da thread!");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Recebi: {}", received);
}
```

3.4 Send e Sync Traits

Rust garante segurança de concorrência através de dois traits:

- **`Send`:** Tipos que podem ser transferidos entre threads

- **`Sync`:** Tipos que podem ser compartilhados entre threads (via referências)

A maioria dos tipos implementa ambos automaticamente. O compilador previne uso incorreto:

```
use std::rc::Rc;
use std::thread;

fn main() {
    let rc = Rc::new(5);
    // thread::spawn(move || {
    //     println!("{}", rc); // ERRO: Rc não é Send!
    // });
    // Use Arc em vez de Rc para threads
}
```

Exercícios Práticos

1. **Threads Básicas:** Implemente o problema do Produtor-Consumidor usando channels. Um produtor gera números de 1 a 100, e um consumidor os imprime.

2. **Sincronização:** Crie 5 threads que incrementam um contador compartilhado 1000 vezes cada. Use `Arc<Mutex<i32>>` e verifique que o resultado final é 5000.

3. **Barrier:** Implemente uma simulação onde 4 threads executam tarefas e devem sincronizar em um ponto (use `std::sync::Barrier`).

Recursos

- The Rust Book - Capítulo 16
- Rust Atomics and Locks (Capítulos 1-4)
- Rust by Example - Threads

Fase 4: Paralelismo de Dados com Rayon

Objetivos

- Usar Rayon para paralelismo de dados (alternativa a OpenMP)
- Compreender iteradores paralelos
- Analisar desempenho de código paralelo

Tópicos Essenciais

4.1 Introdução ao Rayon

Adicione ao `Cargo.toml`:

```toml
[dependencies]
rayon = "1.8"
```

Exemplo básico:

```rust
use rayon::prelude::*;

fn main() {
    let numbers: Vec<i32> = (0..1_000_000).collect();

    // Sequencial
    let sum: i32 = numbers.iter().sum();

    // Paralelo (troque .iter() por .par_iter())
    let sum_parallel: i32 = numbers.par_iter().sum();

    println!("Soma: {}", sum_parallel);
}
```

4.2 Operações Paralelas Comuns

```rust
use rayon::prelude::*;

fn main() {
    let data = vec![1, 2, 3, 4, 5, 6, 7, 8];

    // Map paralelo
    let squared: Vec<i32> = data.par_iter()
        .map(|&x| x * x)
        .collect();

    // Filter paralelo
    let evens: Vec<i32> = data.par_iter()
        .filter(|&x| x % 2 == 0)
        .cloned()
        .collect();

    // Reduce paralelo
    let product: i32 = data.par_iter()
```

```
        .product();
}
```

4.3 Medindo Desempenho

```rust
use std::time::Instant;
use rayon::prelude::*;

fn is_prime(n: u64) -> bool {
    if n < 2 { return false; }
    (2..=(n as f64).sqrt() as u64).all(|i| n % i != 0)
}

fn main() {
    let numbers: Vec<u64> = (1..100_000).collect();

    // Sequencial
    let start = Instant::now();
    let count_seq = numbers.iter().filter(|&&n|
        is_prime(n)).count();
    let time_seq = start.elapsed();

    // Paralelo
    let start = Instant::now();
    let count_par = numbers.par_iter().filter(|&&n|
        is_prime(n)).count();
    let time_par = start.elapsed();

    println!("Sequencial: {} primos em {:?}", count_seq,
        time_seq);
    println!("Paralelo: {} primos em {:?}", count_par, time_par);
    println!("Speedup: {:.2}x", time_seq.as_secs_f64() /
        time_par.as_secs_f64());
}
```

Exercícios Práticos

1. **Multiplicação de Matriz-Vetor:** Implemente a multiplicação de uma matriz por um vetor usando Rayon. Compare o desempenho com a versão sequencial.

2. **Cálculo de π (Monte Carlo):** Use Rayon para paralelizar o método de Monte Carlo para estimar π. Gere milhões de pontos aleatórios e conte quantos caem dentro de um círculo unitário.

3. **Análise de Speedup:** Para o exercício anterior, meça o tempo de execução com 1, 2, 4, 8 threads e calcule o speedup. Compare com a Lei de Amdahl.

Recursos

- Rayon Documentation
- Rayon GitHub Examples

Fase 5: Networking e Sockets

Objetivos

- Criar servidores e clientes TCP/UDP
- Trabalhar com serialização de dados (JSON)
- Combinar threads com networking

Tópicos Essenciais

5.1 Cliente TCP Básico

```rust
use std::net::TcpStream;
use std::io::{Read, Write};

fn main() -> std::io::Result<()> {
    let mut stream = TcpStream::connect("127.0.0.1:8080")?;

    // Envia dados
    stream.write_all(b"Hello, server!")?;

    // Recebe resposta
    let mut buffer = [0; 512];
    let n = stream.read(&mut buffer)?;
    println!("Resposta: {}",
        String::from_utf8_lossy(&buffer[..n]));

    Ok(())
}
```

5.2 Servidor TCP Multi-threaded

```rust
use std::net::{TcpListener, TcpStream};
use std::io::{Read, Write};
use std::thread;

fn handle_client(mut stream: TcpStream) {
    let mut buffer = [0; 512];

    loop {
        match stream.read(&mut buffer) {
            Ok(0) => break, // Conexão fechada
            Ok(n) => {
                println!("Recebi: {}",
        String::from_utf8_lossy(&buffer[..n]));
                stream.write_all(&buffer[..n]).unwrap(); // Echo
            }
            Err(e) => {
                eprintln!("Erro: {}", e);
                break;
            }
        }
    }
}

fn main() -> std::io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:8080")?;
    println!("Servidor escutando na porta 8080");

    for stream in listener.incoming() {
        match stream {
            Ok(stream) => {
                thread::spawn(|| handle_client(stream));
            }
            Err(e) => eprintln!("Erro ao aceitar conexão: {}", e),
        }
    }

    Ok(())
}
```

5.3 Serialização com Serde (JSON)

Adicione ao `Cargo.toml`:

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

```rust
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize, Debug)]
struct Message {
    id: u32,
    content: String,
    timestamp: u64,
}

fn main() {
    let msg = Message {
        id: 1,
        content: "Hello, Rust!".to_string(),
        timestamp: 1234567890,
    };

    // Serializar para JSON
    let json = serde_json::to_string(&msg).unwrap();
    println!("JSON: {}", json);

    // Deserializar de JSON
    let msg2: Message = serde_json::from_str(&json).unwrap();
    println!("Mensagem: {:?}", msg2);
}
```

Exercícios Práticos

1. **Echo Server:** Implemente um servidor TCP que aceita múltiplas conexões simultâneas (usando threads) e retorna qualquer mensagem recebida em maiúsculas.

2. **Chat Simples:** Crie um sistema de chat onde múltiplos clientes podem se conectar a um servidor e enviar mensagens que são broadcast para todos os clientes conectados. Use `Arc<Mutex<Vec<TcpStream>>>` para gerenciar a lista de clientes.

3. **Protocolo Customizado:** Defina uma struct `Request` e `Response` com Serde. Implemente um servidor que recebe requests em JSON, processa (ex: soma dois números) e retorna responses em JSON.

Recursos

- The Rust Book - Capítulo 20
- Serde Documentation
- Rust Network Programming

Fase 6: Tópicos Avançados

6.1 Programação Assíncrona com Tokio

Para sistemas distribuídos modernos, async/await é essencial:

```
// Cargo.toml
// [dependencies]
// tokio = { version = "1", features = ["full"] }

use tokio::net::TcpListener;
use tokio::io::{AsyncReadExt, AsyncWriteExt};

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let listener = TcpListener::bind("127.0.0.1:8080").await?;

    loop {
        let (mut socket, _) = listener.accept().await?;

        tokio::spawn(async move {
            let mut buf = [0; 1024];

            loop {
                let n = socket.read(&mut buf).await.unwrap();
                if n == 0 { break; }
                socket.write_all(&buf[0..n]).await.unwrap();
            }
        });
    }
}
```

6.2 Atomics e Sincronização de Baixo Nível

```
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;
```

```rust
use std::thread;

fn main() {
    let counter = Arc::new(AtomicUsize::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter_clone = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            for _ in 0..1000 {
                counter_clone.fetch_add(1, Ordering::SeqCst);
            }
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Resultado: {}", counter.load(Ordering::SeqCst));
}
```

Recursos

- Tokio Tutorial
- Rust Atomics and Locks (Capítulos 5-10)

Checklist de Preparação

Antes do início do curso, você deve ser capaz de:

- [ ] Criar e compilar projetos Rust com `cargo`
- [ ] Explicar ownership, borrowing e lifetimes
- [ ] Usar structs, enums e pattern matching
- [ ] Tratar erros com `Result` e `Option`
- [ ] Trabalhar com coleções (Vec, HashMap)
- [ ] Criar e sincronizar threads com `Arc<Mutex<T>>`
- [ ] Usar channels para comunicação entre threads
- [ ] Paralelizar loops com Rayon
- [ ] Implementar clientes e servidores TCP básicos

- [ ] Serializar/deserializar dados com Serde

Recursos Adicionais

Livros

- The Rust Programming Language (gratuito, oficial)
- Rust by Example (gratuito, prático)
- Rust Atomics and Locks (concorrência avançada)

Vídeos

- Rust Crash Course (Traversy Media)
- Crust of Rust (Jon Gjengset - avançado)

Comunidade

- Rust Users Forum
- r/rust
- Rust Discord

Prática

- Rustlings (exercícios interativos)
- Exercism Rust Track (problemas com mentoria)

Dicas Finais

1. **O compilador é seu amigo:** Mensagens de erro em Rust são extremamente detalhadas. Leia-as com atenção.
2. **Não lute contra o borrow checker:** Se o compilador reclama de borrowing, provavelmente há um problema real de design. Repense a estrutura.
3. **Comece simples:** Não tente escrever código genérico e super-otimizado logo de início. Faça funcionar primeiro, otimize depois.
4. **Use `clippy` e `rustfmt`:** Essas ferramentas ensinam boas práticas e mantêm o código consistente.

5. **Pratique todos os dias:** Mesmo que sejam 30 minutos, a consistência é mais importante que longas sessões esporádicas.

6. **Pergunte:** A comunidade Rust é extremamente acolhedora para iniciantes. Não hesite em pedir ajuda.

Boa sorte e bem-vindo ao mundo de Rust! 🦀