Rust

Daniel Sá e Gabriel Guimarães

Sumário

- História.
  - Quem criou ?
  - Objetivos?
  - Mercado de trabalho
- Vantagens e desvantagens.

- Exemplo de Código.

História

- Criador
  - Graydon Hoare (Mozilla research)
    - 2006->2012: projeto pessoal
    - 2010: anunciado
    - 2012->presente: produto

- Paradigma: **Multiparadigma**
  - funcional
  - imperativo
  - estruturado
  - genérico
  - concorrente

- # História

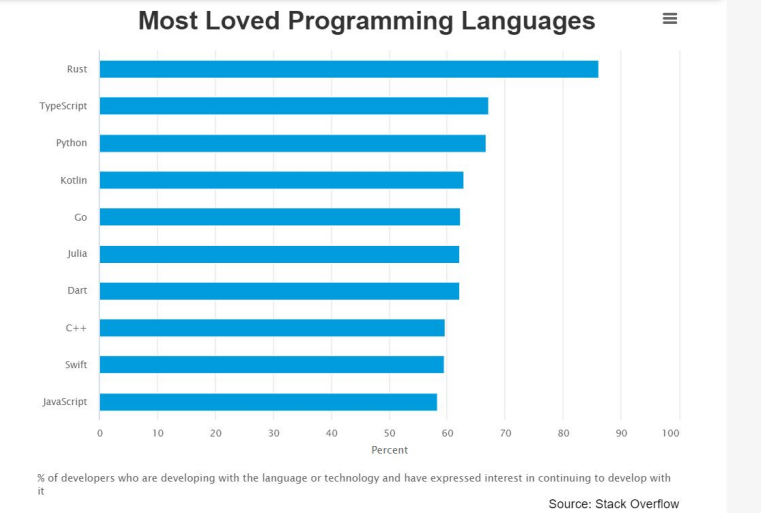
  - Objetivos
    - Segurança
    - Desempenho
    - Concorrência
    - Legibilidade

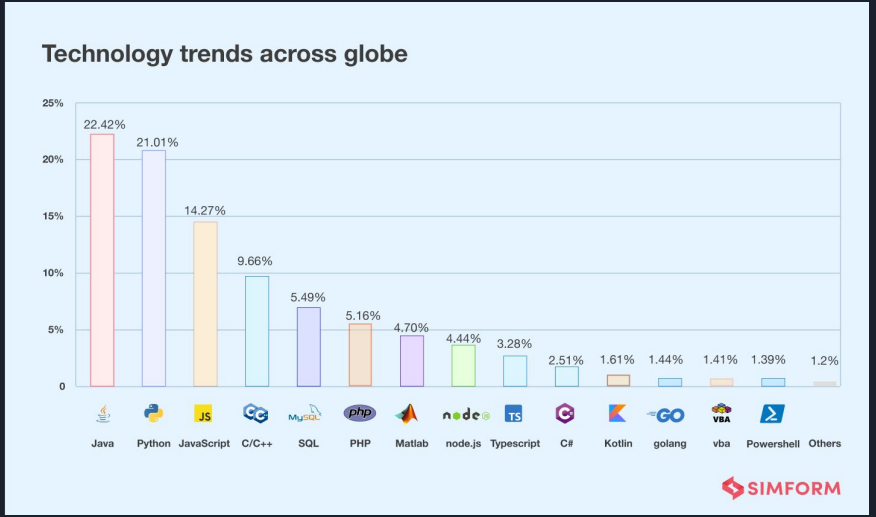
Linguagem moderna, segura para softwares de baixo nível

História

- Mercado de trabalho
  - Aplicabilidade
    - Sistemas Operacionais
    - Games (Amethyst, Piston, etc)
    - WEB
      - Back-end (Rocket, Actix-web, etc)
      - Front-end (Yaw, Druid, etc)
    - IA (Tch-rs, RustLearn)

- História
  - Mercado de trabalho
    - Popularidade





- História

  - Mercado de trabalho

    - Salário
      - **R$6000 - R$12000**

Vantagens e Desvantagens

Vantagens:

- Segurança de memória.
- Concorrência robusta.
- Alto desempenho.
- Confiabilidade e manutenibilidade.
- Ecossistema crescente.
- Portabilidade.

- Vantagens e Desvantagens

- Desvantagens:
  - Curva de aprendizado.
  - Complexidade.
  - Tempo de compilação.
  - Ecossistema em desenvolvimento.
  - Integração com outras linguagens.

Exemplo de Código

Tipos de dados:

- ❏ Inteiro :
  - Com sinal (i8, i16, i32, i64, i128, isize).
  - Sem sinal (u8, u16, u32, u64, u128, usize).
- ❏ Ponto flutuante: (f32, f64).
- ❏ Booleano: (bool)
- ❏ Caractere: (char)
- ❏ Dados de Propriedade: (String, &str)
- ❏ Dados compostos: (tuple, array, slice)
- ❏ Compostos estruturados: (struct, enum)
- ❏ Dados operacionais: (Option<T>, Result<T,E>).
- ❏ Referência: (&T, &mut T)

Exemplo de Código

- Tipos de dados:
- Função: (fn)
- Dados genéricos: (T)
- Dado unidade (`()`)
- Vetor: (vec<T>).
- Fatia mutável. (&mut [T])
- Intervalo. (Range, RangeInclusive)
- Iterator: (Iterator).

- Exemplo de Código

```rust
fn main() {
    let num1 = 10;
    let num2 = 5;
    let sum = add_numbers(num1, num2);
    println!("A soma de {} e {} é: {}", num1, num2, sum);
}

fn add_numbers(a: i32, b: i32) -> i32 {
    return a + b;
}
```

Exemplo de Código

1. Propriedades de propriedade (ownership):

   Rust possui um sistema de propriedade exclusivo que define como os recursos são alocados e liberados na memória. Esse sistema garante que apenas uma única entidade (variável, função etc.) possua acesso exclusivo a um recurso em um determinado momento, evitando problemas como vazamentos de memória e condições de corrida.

```rust
fn main() {
    let s = String::from("Hello"); // String criada e de propriedade da vari
    take_ownership(s); // 's' é transferida para a função 'take_ownership'

    // 's' não é mais válida aqui, pois a propriedade foi transferida

    let x = 5; // Variável 'x' de propriedade da função 'main'
    make_copy(x); // 'x' é passada por valor para a função 'make_copy'

    // 'x' ainda é válida aqui, pois i32 implementa a trait 'Copy'

} // As variáveis 's' e 'x' são liberadas aqui
  // A memória ocupada por 's' é desalocada automaticamente
  // 'x' não requer desalocação, pois é um tipo 'Copy'
```

```rust
} // As variáveis 's' e 'x' são liberadas aqui
  // A memória ocupada por 's' é desalocada automaticamente
  // 'x' não requer desalocação, pois é um tipo 'Copy'

fn take_ownership(some_string: String) {
    println!("Valor recebido: {}", some_string);
    // 'some_string' é válida aqui e é de propriedade da função 'take_owner
} // 'some_string' é liberada aqui e a memória ocupada por ela é desalocada

fn make_copy(some_integer: i32) {
    println!("Valor recebido: {}", some_integer);
    // 'some_integer' é válida aqui e é de propriedade da função 'make_copy
} // 'some_integer' é liberada aqui
```

Exemplo de Código

- Conceito de empréstimos (borrowing):

  Em Rust, é possível emprestar referências imutáveis ou mutáveis de um recurso para outras partes do código. Isso permite compartilhar dados sem perder a propriedade e ajuda a evitar cópias desnecessárias. O sistema de empréstimos é verificado em tempo de compilação, garantindo que as regras de mutabilidade sejam seguidas corretamente.

```rust
fn main() {
    let s = String::from("Hello"); // String criada e de propriedade da var
    let len = calculate_length(&s); // Passagem de empréstimo de referência

    println!("O comprimento da string '{}' é {}", s, len);
    // 's' continua sendo válida aqui, pois a referência foi emprestada par
}

fn calculate_length(s: &String) -> usize {
    s.len()
    // 's' é apenas uma referência à String original e não possui propriedad
    // O valor da String original não pode ser modificado através dessa refe
}
```

Exemplo de Código

- Tempo de vida (lifetime):

  Rust possui um sistema de tempo de vida que rastreia e verifica as dependências de referências para garantir que não haja referências inválidas. O sistema de tempo de vida permite que o compilador valide a validade e a duração de referências em tempo de compilação, prevenindo erros de acesso inválido à memória.

```rust
fn main() {
    let result;

    {
        let s1 = String::from("Hello"); // String criada e de propriedade da
        let s2 = String::from("World"); // String criada e de propriedade da

        result = longest_word(&s1, &s2); // Passagem de empréstimo de referê
        println!("A palavra mais longa é: {}", result);
        // As referências 's1' e 's2' continuam sendo válidas aqui, dentro d
    }

    // As referências 's1' e 's2' estão fora de escopo e não são mais válida
    // A variável 'result' ainda é válida aqui, pois tem uma vida maior

    // println!("A palavra mais longa é: {}", result); // Erro! 'result' est
```

```rust
    // As referências 's1' e 's2' estão fora de escopo e não são mais válid
    // A variável 'result' ainda é válida aqui, pois tem uma vida maior

    // println!("A palavra mais longa é: {}", result); // Erro! 'result' es
}

fn longest_word<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    if s1.len() > s2.len() {
        s1
    } else {
        s2
    }
    // A função retorna uma referência para uma das strings de entrada
    // A anotação <'a> indica que a referência retornada tem o mesmo tempo
}
```

Exemplo de Código

- Tratamento explícito de erros:

  Rust incentiva o tratamento explícito de erros, em vez de exceções, através do uso de tipos Result<T, E> e Option<T>. Isso obriga o desenvolvedor a lidar com possíveis erros de forma explícita, melhorando a robustez do código e tornando mais fácil identificar e tratar condições de erro.

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

fn main() {
    let file_name = "example.txt";

    match read_file_contents(file_name) {
        Ok(contents) => {
            println!("Conteúdo do arquivo '{}':\n{}", file_name, contents);
        }
        Err(error) => {
            eprintln!("Erro ao ler o arquivo '{}': {}", file_name, error);
        }
    }
}

fn read_file_contents(file_name: &str) -> Result<String, std::io::Error> {
    let mut file = File::open(file_name)?;

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

    Ok(contents)
}
```

Exemplo de Código

- Macros:

  Rust possui um poderoso sistema de macros que permite a geração de código durante a compilação. As macros permitem a escrita de código mais conciso e flexível, permitindo abstrações e automatização de tarefas repetitivas.

```rust
macro_rules! greet {
    ($name:expr) => {
        println!("Hello, {}!", $name);
    };
}

fn main() {
    greet!("John"); // Chama a macro 'greet' com o argumento "John"
    greet!("Alice"); // Chama a macro 'greet' com o argumento "Alice"
}
```