Rust: uma visão geral

Carlos Augusto Motta de Lima 7991228
José Ernesto Young Rodrigues 7991083

junho de 2017

Resumo

Rust é uma lingugagem de programação multiparadigma, projetada para o desenvolvimento de sistemas seguros e eficientes. Suas características possibilitam segurança de memória e confiança para o programador lidar com paralelismo e concorrência, sem sacrifícios de performance. De criação recente, possui uma comunidade crescente e ativa, e uma documentação de qualidade que busca facilitar a superação das dificuldades de aprendizado da linguagem.

Palavras-chaves: Rust. lingugagem de programação. paralelismo e concorrência.

Introdução

Rust é uma linguagem de programação nascida em 2006 como um projeto pessoal de Graydon Hoare, empregado da Mozilla - que, em 2009, começou patrocinar o desenvolvimento da linguagem como projeto de pesquisa, anunciando-o em 2010.

Desde o princípio, os desenvolvedores de Rust tinham em mente as qualidades que queriam que a linguagem tivesse. Muita atenção foi dada às falhas das demais linguagens existentes (pouca segurança de memória; suporte à concorrência precário; controle limitado de recursos etc). Dessa forma, Rust foi proposta como uma alternativa de alta performance, segurança de memória e maior segurança em programação paralela e concorrente.

Porém, a equipe não tinha clara a forma como atingiria esses objetivos. Assim, Rust passou por diversas mudanças bastante significativas ao longo de sua história - descritas pelos desenvolvedores (KLABNIK, 2016) como experimentos necessários para se chegar ao resultado final.

A versão 1.0 da linguagem foi lançada em 15 de maio de 2015, e em 8 de junho de 2017 chegou à versão 1.18.0.

Rust foi construída de tal forma a passar ao programador total confiança ao desenvolvedor sistemas, para que ele possa lidar com gerenciamento de memória e multithreading de maneira prática e com confiança.

1 Conceitos

Nesta seção se encontram breves explicações dos principais conceitos de Rust. O leitor não familiarizado com a linguagem poderá ter um primeiro contato, e é possível que ache interessantes as particularidades de Rust, que misturam programação funcional, procedural e orientada a objetos.

Num primeiro momento, alguns conceitos podem parecer complexos. De fato, a curva de aprendizado pode ser uma barreira para interessados em Rust. Porém, como será explicado mais adiante, os conceitos de Rust que fogem um pouco do comumente visto nas linguagens de alto desempenho clássicas são os que garantem segurança de memória com eficiência.

1.1 Tipos e variáveis

Rust é uma linguagem de baixo nível e fortemente tipada. Provisiona uma ampla variedade de tipos primitivos, que incluem inteiros com sinal (`i8`, `i16`, `i32`, `i64` e `isize`), inteiros sem sinal (`u8`, `u16`, `u32`, `u64` e `usize`), números de ponto flutuante (`f32`, `f64`), caracteres (`char`), que são codificados em unicode (4 bytes), booleanos (`bool`), arrays (conjuntos de variáveis de um mesmo tipo), tuplas (conjuntos de variáveis de tipos possivelmente distintos), entre outros.

Variáveis têm seu tipo definido em tempo de compilação, o que quer dizer que uma variável não pode receber um valor de um tipo diferente. No entanto, a declaração explícita do tipo das variáveis não é obrigatória - o compilador infere tipos quando omitidos.

Por padrão, variáveis em Rust são imutáveis. No entanto, é possível definir variáveis mutáveis através da palavra-chave `mut`.

Abaixo, um exemplo dos tipos primitivos extraído do *Rust by Example*:

```
// Variáveis pode ter anotações de tipo.
let logical: bool = true;

let a_float: f64 = 1.0;  // Anotação comum
let an_integer   = 5i32; // Anotação em sufixo

// Ou um tipo padrão será atribuído.
let default_float   = 3.0; // `f64`
let default_integer = 7;   // `i32`

let mut mutable = 12; // `i32` mutável.

// Erro! O tipo da variável não pode ser trocado.
mutable = true;
```

Também é possível definir tipos personalizados, que incluem tuplas nomeadas (`tuple struct`), além das `structs` clássicas do C, e `unit structs`, que são tipos genéricos e sem campos, como podemos ver no exemplo abaixo:

```rust
// A tuple struct
struct Pair(i32, f32);

// A classic struct with two fields
struct Point {
    x: f32,
    y: f32,
}

// A unit struct
struct Nil;

// Structs can be reused as fields of another struct
struct Rectangle {
    p1: Point,
    p2: Point,
}
```

1.2 Ownership

Todas as linguagens devem gerenciar como a memória é utilizada por um programa: algumas - como Java e Ruby, por exemplo - possuem um componente (o *Garbage Collector*) que procura constantemente, em tempo de execução, espaços de memória que não são mais utilizados; outras, como C, requerem que o programador aloque e libere a memória explicitamente (usando por exemplo comandos como `malloc` e `free`).

Para gerenciar memória, Rust possui um sistema denominado `ownership`, que é um conjunto de regras a serem seguidas pelo programador e que são verificadas pelo compilador. Assim, o gerenciamento de memória é feito em tempo de compilação, com "custo zero" durante a execução do programa.

Basicamente, são três as regras que regem `ownership`:

- Todo valor possui uma variável que é sua `owner`;
- Cada valor (na memória) só pertence a **uma** owner por vez;
- Quando uma `owner` sai de escopo, a memória é liberada.

O fato de cada variável ser responsável por liberar a própria memória garante que nenhum espaço de memória será liberado duas vezes.

Quando é feita uma atribuição (`let foo = x`) ou uma chamada de função (`foo(x)`), a `ownership` é transferida (processo esse chamado de `move`).

Em ambos exemplos, `foo` passa a ser a `owner` do espaço de memória antes pertencente a `x`. Também em ambos os casos, `x` deixa de ter acesso à memória que antes possuía. Essa regra evita acesso a "ponteiros soltos" (`dangling pointers`).

No exemplo abaixo, a última linha da função `main` não compilaria. Como a `ownership` da variável `the_bravest` foi movida na chamada da função `go_to_die`, o compilador identifica que a variável `the_bravest` na última linha de `main` já não é `owner` de nenhum espaço de memória.

```rust
struct BraveStruct {
    courage: u32,
    cowardness: u32,
}

fn go_to_die(some_struct : BraveStruct) {
    println!("Last words: {:?}...", some_struct);
    // Do nothing else and goes out of scope
}

pub fn main() {
    let the_bravest;
    println!("A hero is born!");

    the_bravest = BraveStruct {
        courage: 9001 as u32,
        cowardness: 2 as u32,
    };

    // ownership of the_bravest is moved to go_to_die
    go_to_die(the_bravest);

    // the line below wouldn't compile
    // println!("{:?} didn't die!", the_bravest);
}
```

1.3 Borrowing

O sistema de `ownership` pode parecer ser complicado. Por exemplo: como fazer quando se quer acessar o valor de uma variável após passá-la como parâmetro na chamada de uma função?

A resposta para esse questionamento vem com o sistema de `borrowing`. Através dele, é possível "emprestar" uma variável sem transferir sua `ownership`. Esse "empréstimo" se dá através de referências, de forma parecida com a usada na linguagem C.

O sistema de `borrowing` também possui regras específicas verificadas em tempo de compilação. De forma simples, são duas:

- Podem existir diversas **referências imutáveis** de uma variável ao mesmo tempo;
- Se existe uma **referência mutável** de uma variável mutável, ela é a **única** referência da variável enquanto ela viver.

Veja a seguir um exemplo que demonstra o uso de referências:

```rust
#[derive(Debug)]
struct Tutor {
    name: &'static str,
    position: &'static str,
```

```
}

fn get_phd(some_tutor : &mut Tutor) -> bool {
    some_tutor.position = "Professor";
    // Got his Phd -> return true :)
    true
}

pub fn main() {
    let alfredo = Tutor { name: "Goldman", position: "Professor" };
    let mut pedro = Tutor { name: "Bruel", position: "TA" };

    let our_professor = &alfredo;
    let his_supervisor = &alfredo;
    let our_TA = &pedro;

    // Impossible to be a Professor while being a TA :(
    get_phd(&mut pedro);
}
```

Devido à última linha, o código acima é inválido e não compila.

Isso acontece porque, na expressão `let our_TA = &pedro`, é criada uma referência **imutável** para a variável `pedro`. Mais abaixo, a expressão `&mut pedro`, passada como parâmetro na chamada da função `get_phd`, cria uma referência **mutável** para a mesma variável. Dessa forma, a segunda regra para criação de referências é quebrada, invalidando o código.

1.4 Lifetimes

Para garantir segurança no gerenciamento de memória ainda resta um sistema fundamental: `lifetimes`. Elas são o meio de comunicar ao compilador o tempo de vida de uma referência, evitando, principalmente, ponteiros soltos.

Mas como este problema é evitado? Suponha que a função a seguir calcule qual coluna de uma matriz é o vetor de maior norma:

```
fn max_column_norm<'a>(matrix: &'a Vec<Vec<i32>>) -> &'a Vec<i32> {
    ...
}
```

Esta função recebe como argumento uma referência para uma matriz e retorna uma referência para o vetor da coluna de norma máxima. Se após a execução desta função a matriz original for descartada e o espaço de memória utilizado por ela for liberado, a referência retornada pela função (a coluna da matriz de norma máxima) se torna inválida.

As anotações de `lifetimes` presentes na assinatura da função evitam isto. Neste contexto, uma palavra precedida de uma aspas simples, como `'a`, representa o tempo de vida associado àquela referência, o qual pode ser indiretamente associado a outras referências como, no caso, o valor de retorno da função.

Neste exemplo, efetivamente o que é informado ao compilador sobre o tempo de vida das referências em questão é o seguinte: a função `max_column_norm` lida com apenas um `lifetime` genérico `<'a>`, que será extraído da varíavel (que é uma referência) `matrix: &'a Vec<Vec<i32>>`, e será atribuido ao valor de retorno (que também é uma referência) `&'a Vec<i32>`.

Assim, o compilador garante que o tempo de vida restante da referência retornada pela função será exatamente o mesmo que aquele da referência da matriz original. Se em algum momento a matriz sair de escopo e sua memória for liberada, automaticamente o acesso à referência retornada pela função será invalidado pelo compilador.

Ademais, é importante ficarem claros dois pontos básicos sobre `lifetimes`. O primeiro é sobre as palavras que representam os tempos de vida das referências que são passadas como parâmetro para a funcão. No exemplo dado, a única `lifetime` presente é `'a`, mas nada impede que existam diversas: e.g. `fn another_function<'a, 'b, 'c, 'd>` . Além disso, se duas referências (que são argumentos de uma funcão) forem declaradas com o mesmo tempo de vida (e.g. `fn func<'a>(myref1: &'a Vec<u16>, myref2: &'a Vec<u32>)`) então a palavra (`'a`) será associada automaticamente ao tempo de vida mais restrito entre as duas opções, isto é, será associada àquela que deixará de existir primeiro.

Finalmente, o sistema de `lifetimes` existe para atrelar o tempo de vida de uma (ou mais) referência passada como argumento para uma função (o qual é conhecido em tempo de compilação) ao tempo de vida de uma (ou mais) referência que será retornada por esta função (o qual seria desconhecido caso não existissem `lifetimes`).

Como um adendo, vale ressaltar que em Rust não faz sentido pensar em uma função que não receba nenhum argumento e retorne uma referência, pois tal função retornaria uma referência a um espaço com `ownership` atrelada ao escopo dela mesma, ou seja, ao terminar a execução da função toda esta memória seria liberada e a referência estaria invalidada. Para contornar isso, pode-se retornar o valor em si e não uma referência, transferindo assim a `ownership` dele para alguma variável no escopo onde esta função executada.

2 Paralelismo e Concorrência

Rust procura não opinar sobre qual a melhor forma de construtir programas com paralelismo e concorrência e, pelo contrário, busca permitir que o programador lide com as abstrações de hardware provenientes do SO da forma que bem entender. Não obstante, a linguagem provê ferramentas para um manuseio claro e limpo sobre estas estruturas.

Existem duas formas de alcançar paralelismo e concorrência, ambas utilizando threads reais (isto é, quando uma thread no programa representa exatamente uma thread no hardware): principalmente por compartilhamento de memória ou principalmente por troca de mensagens.

Aqui serão expostas as estruturas básicas para seguir em ambos os caminhos (ou até mesmo em um híbrido dos dois).

2.1 Troca de mensagens: *channels*

Um slogan do *Effective Go* (série de textos que descrevem as boas práticas da linguagem Go, feita pelos próprios desenvolvedores) diz o seguinte: *Do not communicate by sharing memory; instead, share memory by communicating.* Esta ideia surgiu com a noção

de que manter um programa que compartilha memória entre threads correto e seguro é difícil.

Nessa mesma linha, Rust possui em sua biblioteca padrão diretivas para a criação de canais, pelos quais é possível transmitir `ownerships` de valores entre threads. Abaixo está um exemplo de um programa com esta característica.

```
use std::thread;
use std::sync::mpsc;

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

    thread::spawn(move || {
        let vals = vec![
            String::from("One"),
            String::from("message"),
            String::from("at"),
            String::from("a"),
            String::from("time"),
        ];

        for val in vals {
            tx.send(val).unwrap();
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }
}
```

Neste trecho de código a atenção deve ser voltada a dois elementos: `mpsc::channel()` e `thread::spawn`. O primeiro deles diz respeito à criação de um canal onde vários agentes (e.g. threads, funções, ...) poderão enviar conteúdo mas apenas um agente receberá todas elas (*mpsc* significa *multiple producers single consumer*).

Após a criação do canal, este é desestruturado em dois valores: `(tx, rx)`, que representam respectivamente o transmissor e o receptor das mensagens. Como se trata de um *mpsc* o transmissor `tx` poderia ser clonado diversas vezes para ser utilizado por diversas threads, mas o receptor `rx` não. Da forma que foi escrito este programa, as mensagens são enviadas assincronamente, ainda assim Rust possui outra diretiva para uma comunicação síncrona: `mpsc::sync_channel()`.

Vale ressaltar que todo canal criado possui um tipo. Por exemplo, a criação do canal neste código poderia ser substituida por `mpsc::channel::<String>()` tornando explicito o tipo do canal (que, no código acima, é inferido pelo compilador ao analisar as mensagens enviadas). Isto significa que não é possível, por exemplo, enviar uma string e um inteiro pelo mesmo canal de comunicação.

Já o segundo elemento, representa a criação de uma thread. Para a chamada `thread::spawn()`, é passada uma *closure*, que é um elemento parecido com funções em Rust, mas que, entre outras coisas, é capaz de capturar variáveis do ambiente no qual ela é definida. Por conta desta captura é que existe a palavra `move` no começo da chamada. Com ela, o compilador é informado de que toda variável utilizada dentro da *closure* mas definida fora dela (no caso, apenas `tx`) deve ser movida para dentro, ou seja, a ownership do valor de `tx` pertence agora ao escopo da *closure*, que por sua vez, vive em outra thread. Todo código escrito dentro da *closure* é executado na nova thread.

A análise sobre mover uma ownership para dentro da *closure* é mais complexa do que isto, pois pode ser possível apenas emprestar a variável (`borrowing`), e nesse caso o a variável continuaria no escopo prévio dela, mas tal análise detalhada foge ao escopo deste texto.

2.2 Compartilhamento de memória: *mutex*

Ainda que existam diretivas para a criação de paralelismo e concorrência via mensagens seguindo o que foi proposto na linguagem Go, como foi citado anteriormente, Rust busca não opinar sobre a melhor forma de resolver um determinado problema (fora a própria arquitetura da linguagem), e portanto oferece também a possibilidade de compartilhar memória de forma segura via *mutex*.

O interessante desta estrutura em Rust é que ela encapsula o espaço de memória a ser compartilhado e garante que apenas um agente por vez tenha acesso a ele. Curiosamente, a forma com a qual deve-se raciocinar sobre a utlização de *mutexes* é muito similar a qualquer outra utilização de memória em Rust, uma vez que `ownership`, `borrowing` e `lifetimes` garantem a segurança de memória por todo o programa.

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

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

    for _ in 0..10 {
        let moved_counter = counter.clone();
        let handle = thread::spawn(move || {
            let mut num = moved_counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

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

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

Neste exemplo temos dois elementos relevantes na criação do *mutex*: `Arc` e `Mutex`. O primeiro deles é um *Atomic Reference Counter*, isto é, um tipo de referência especial que mantém um contador com quantas referências existem para tal valor e, além disso, atualiza este contador de forma atômica. Este tipo é relevante aqui porque é importante saber quando o *mutex* não é mais utilizado (referenciado) para então ter a memória liberada; entretanto, como as referências são utilizadas por diferentes threads, a atualização do contador deve ser feita atomicamente para se manter correta.

O segundo deles é o próprio `Mutex`. Talvez não tenha ficado claro no exemplo acima, mas assim como todas as variáveis em Rust, o *mutex* `counter` tem seu tipo bem definido, o que significa que o valor que ele guarda é definido. No exemplo este valor começa com 0 que é tratado como um `i32` por padrão.

Assim, a variável `counter` é na verdade um encapsulamento do inteiro 0, no qual só é possível entrar após a chamada do `.lock()`. Esta chamada ou permite o acesso local e impede todos os outros ou, se outra parte do código já possui acesso, espera até que o *mutex* seja liberado.

Assim que o `counter` sai de escopo em uma thread, o *mutex* é liberado para que outra chamada do `.lock()` seja liberada da espera.

As chamadas de `.unwrap()` servem apenas para informar ao compilador que nenhum erro é esperado para tais chamadas, isto não deve ser feito em um sistema em produção, apenas para prototipação e testes com a linguagem.

2.3 *Fearless Concurrency*

Com estas formas de implementar concorrência e paralelismo, os desenvolvedores de Rust encontraram um novo ponto no desenvolvimento de programas desta classe de complexidade que apresenta menos riscos para o programador.

Com estes paradigmas, boa parte dos erros que outrora se tornariam erros em tempo de execução, surgem em tempo de compilação e tornam o programador mais atento à forma pela qual o seu programa lida com a memória do computador entre as estruturas e algoritmos utilizados.

Assim, busca-se encontrar um ambiente no qual seja possível escrever programas paralelos e concorrentes sem medo em uma linguagem muito eficiente. O que surge como uma bela alternativa à realidade de bugs extremamente difíceis de reproduzir e consertar que são os programas existentes hoje em dia em linguagens tradicionais.

3 Comunidade e projetos

Apesar de ser uma linguagem nova, Rust já possui uma comunidade entusiasmada, e algumas organizações já a utilizam em produção. Nesta seção, citaremos alguns projetos que já estão usando Rust.

3.1 Mozilla Servo

Servo é um browser moderno e de alta performance projetado tanto para aplicações como para uso embarcado. Patrocinado pela Mozilla e escrito em Rust, o projeto ambiciona alcançar melhor paralelismo, segurança, modularidade e desempenho.

Junto com o próprio compilador do Rust, esse é o maior projeto *open source* escrito em Rust.

Servo ainda está em fase de desenvolvimento e testes e atualmente libera *nightly builds* para diversos sistemas operacionais.

3.2 PaaS Logs

OVH é uma empresa francesa, fornecedora de acesso à internet que oferece servidores dedicados, hospedagem compartilhada e na nuvem, registro de domínio, e serviços de telefonia VOIP.

A companhia possui 20 datacenters hospedando em torno de 1,500,000 de máquinas e atua, no momento, apenas na Europa.

A RunAbove, setor de inovação da OVH, desenvolveu em Rust o **PaaS Logs** (RAQUIL, 2016), uma ferramenta de gerenciamento de logs, que verifica dados de logs e permite visualização das informações mais importantes.

3.3 Coursera

Coursera é uma das maiores plataformas de tecnologia educacional do mundo, e oferece cursos online elaborados por mais de 80 universidades parceiras de vários países.

Um dos componentes mais sofisticados da plataforma é a infraestrutura de atribuições de notas às submissões dos alunos, e alguns dos vários scripts envolvidos no processo de orquestração dos componentes foram escritos em Rust (SAETA, 2016).

3.4 Dropbox

Com mais de 500 milhões de usuários ao redor do mundo, o Dropbox lida com servidores de armazenamento massivo altamente distribuídos, o que requiriu que usassem para determinados componentes de sua infraestrutura linguagens de alto desempenho. Entre as possibilidades, decidiram pelo Rust - o critério final de decisão entre Rust e Go foi o consumo menor de memória de Rust (METZ, 2016).

Considerações finais

Após anos de desenvolvimento, Rust tem se mostrado cada vez mais interessante para os programadores. Há inclusive uma pesquisa que a aponta como a linguagem mais amada por eles (CIBOROWSKI, 2017).

Diversos pontos interessantes foram observados em relação à linguagem e concluídos como positivos. Rust pode em alguns aspectos ser desafiadora para um programador pouco familiarizado com gerenciamento explícito de memória, pois se utiliza de estruturas complexas como as citadas na seção *Conceitos*.

No entanto, uma análise mais completa mostrou que, ao trazer, com tais estruturas, segurança de memória garantida em tempo de compilação, o compilador de Rust antecipa problemas que poderiam gerar *bugs* num programa que, em outra lingugagem, não apresentaria erros de compilação - depurar tais programas muitas vezes é um trabalho árduo. Ao forçar o programador a pensar de forma mais estruturada no gerenciamento de memória, Rust consegue atingir seu objetivo de oferecer um ambiente de programação *memory-safe* e *thread-safe* sem medo.

Referências

CIBOROWSKI, J. *Why Rust is the Most Loved Language by Developers*. 2017. Blog da Mozilla. Disponível em: <https://medium.com/mozilla-tech/why-rust-is-the-most-loved-language-by-developers-666add782563>. Acesso em: 25 jun. 2017. Citado na página 11.

KLABNIK, S. *The History of Rust*. ACM Digital Library, 2016. Palestra do "Applicative 2016". Disponível em: <https://doi.org/10.1145/2959689.2960081>. Acesso em: 25 jun. 2017. Citado na página 1.

METZ, C. *The Epic Story of Dropbox's Exodus From the Amazon Cloud Empire*. Wired, 2016. Disponível em: <https://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire>. Acesso em: 25 jun. 2017. Citado na página 10.

RAQUIL, M. *PaaS Logs: an inside look at our log management infrastructure*. 2016. Blog da OVH. Disponível em: <https://www.ovh.co.uk/news/articles/a2202.paas-logs-management-ovh>. Acesso em: 25 jun. 2017. Citado na página 10.

RUST by Example. 2017. Disponível em: <https://rustbyexample.com/>. Acesso em: 25 jun. 2017.

SAETA, B. *Rust and Docker in production @ Coursera*. 2016. Blog do Coursera. Disponível em: <https://building.coursera.org/blog/2016/07/07/rust-docker-in-production-coursera>. Acesso em: 25 jun. 2017. Citado na página 10.