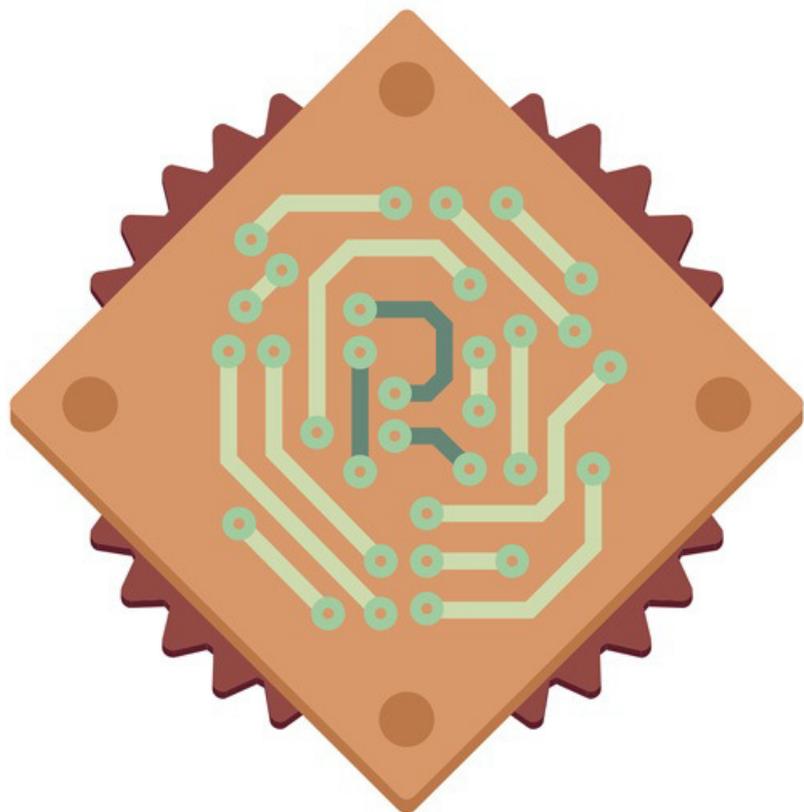


Programação Funcional e Concorrente em Rust



© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Edição

Adriano Almeida

Vivian Matsui

Revisão

Bianca Hubert

Vivian Matsui

[2018]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

ISBN

Impresso e PDF: 978-85-94188-42-7

EPUB: 978-85-94188-43-4

MOBI: 978-85-94188-44-1

Caso você deseje submeter alguma errata ou sugestão, acesse
<http://erratas.casadocodigo.com.br>.

AGRADECIMENTOS

Primeiro gostaria de agradecer à minha família e aos meus pais, por estarem sempre comigo e me apoiando. Obrigada, Diego e Kinjo, e obrigada aos pequenos sempre alegres e contentes Fluffy, Ffonxio, Highlander e aos miniffonxios.

Tenho de agradecer à equipe maravilhosa que colabora comigo na ThoughtWorks, especialmente Enzo, Icaro, Inajara, Roberto, Thait e Thais. Ainda na ThoughtWorks, agradeço ao pessoal de marketing, que faz milagres acontecerem e especialmente ao Paulo Caroli, por estar sempre me apoiando, e ao Luciano Ramalho, que é um dos desenvolvedores mais incríveis que conheço. Além disso, de fora da ThoughtWorks, quero agradecer à Ana Paula Maia, por nossas conversas sobre Rust, e ao Bruno Tavares, por também ser um entusiasta de Rust. À Mozilla, que apoiou a criação do Rust e soube conduzir a linguagem a um momento incrível.

Quero agradecer a toda a equipe da Casa do Código, que me ajudou no desenvolvimento deste livro, especialmente a Vivian e a Bianca.

Por último, agradeço a você, leitor, que dedicou um tempo para aprender Rust e agora quer procurar novas ideias para essa linguagem tão legal.

SOBRE A AUTORA

Julia Naomi Boeira é desenvolvedora de software na ThoughtWorks Brasil, liderando o desenvolvimento de Realidade Aumentada e sempre apoiando novas linguagens para trazer uma cultura poliglota. Atua com o objetivo de trazer mais *know-how* de linguagens funcionais e permitir maior diversidade de pensamentos nos projetos.

Sobre o revisor técnico

Bruno Lara Tavares acompanha o desenvolvimento de Rust desde 2015 e interessa-se em como integrar esse novo ecossistema em projetos atuais. Já trabalhou em diversas linguagens como consultor e, hoje em dia, desenvolve microsserviços em Clojure.

INTRODUÇÃO

Este livro apresenta Rust de forma que a programação funcional seja inerente ao desenvolvimento, assim como identifica uma das grandes forças do Rust, a concorrência. Para isso, o livro foi dividido em quatro partes.

Na primeira, temos um resumo do potencial do Rust e sua história. Passaremos por uma explicação de por que Rust deve ser considerado uma ótima opção, bem como por suas principais características, e apresentaremos um breve módulo de como pode ser feito TDD em Rust. Na segunda, mostramos a programação funcional na perspectiva dessa linguagem, comparando ao Clojure. Nosso foco será principalmente em funções, *traits*, *iterators*, *adapters* e *consumers*.

Na terceira parte, apresentamos a concorrência nos diversos modos que o Rust oferece, como a criação de *threads*, o compartilhamento de estados e a transferência de informações por canais. Então, na última parte, resumimos as outras, de forma a apresentar 4 frameworks HTTP, sendo dois de alto nível (Iron e Nickel), um de baixo nível (Hyper) e um de programação assíncrona (Tokio).

Público-alvo

Este livro tem como objetivo usar Rust para desenvolver o pensamento funcional e concorrente em qualquer pessoa que goste de programação, por conta dos motivos já explicados. Logo, é recomendado ter um básico conhecimento nessa linguagem, pois

muitas das implementações de código esperam um conhecimento raso dele.

Outros aspectos importantes da escolha de Rust foram a paixão da autora pela linguagem, já que poupou muitas dores de cabeça por não ter mais de se preocupar em implementar sistemas concorrentes em C++, e a facilidade da sintaxe, que tem uma sensação de linguagem de alto nível.

Sumário

Por que Rust?	1
1 Introdução ao Rust	2
1.1 História do Rust	4
2 Por que Rust?	8
2.1 Type Safety	10
2.2 Entendimento da linguagem e sua sintaxe	12
2.3 Segurança de memória	14
2.4 Programação concorrente	18
2.5 Mais sobre Rust	20
3 TDD em Rust	22
3.1 Por que TDD?	22
3.2 Um exemplo em Rust	25
Programação funcional	30
4 O que é programação funcional?	31

4.1 Imutabilidade	34
4.2 Laziness	35
4.3 Funções	36
5 Definindo funções	38
5.1 Funções de ordem superior	41
5.2 Funções anônimas	42
5.3 Funções como valores de retorno	45
6 Traits	48
6.1 Trait bounds	50
6.2 Traits em tipos genéricos	51
6.3 Tópicos especiais em traits	52
7 If let e while let	58
7.1 if let	59
7.2 if let else	60
7.3 while let	61
8 Iterators, adapters e consumers	62
8.1 Iterators	64
8.2 Maps, filters, folds e tudo mais	67
8.3 Consumer	71
Programação concorrente	74
9 O que é programação concorrente?	75
9.1 Definição de concorrência	75
9.2 Por que Clojure é uma boa referência de comparação?	76

9.3 E o Rust? Como fica?	77
9.4 Rust: concorrência sem medo	79
9.5 Quando utilizar concorrência?	81
10 Threads — A base da programação concorrente	82
10.1 Lançando muitas threads	84
10.2 Panic! at the Thread	87
10.3 Threads seguras	88
11 Estados compartilhados	90
12 Comunicação entre threads	95
12.1 Criando channels	96
12.2 Enviando e recebendo dados	97
12.3 Como funciona?	98
12.4 Comunicação assíncrona e síncrona	99
Aplicando nossos conhecimentos	102
13 Aplicando nossos conhecimentos	103
13.1 Iron	104
13.2 Mais detalhes do Iron	112
13.3 Iron testes	117
14 Brincando com Nickel	129
14.1 Routing	131
14.2 Lidando com JSON	132
14.3 Templates em Nickel	133
15 Hyper: servidores desenvolvidos no mais baixo nível	136

15.1 Criando um servidor Hello World mais robusto	138
15.2 Fazendo nosso servidor responder um Post	140
15.3 Uma API GraphQL com Hyper e Juniper	145
16 Tokio e a assincronicidade	177
16.1 Tokio-proto e Tokio-core	178
16.2 Futures	187
16.3 A versão assíncrona	190
17 Bibliografia	196

Por que Rust?

- Introdução ao Rust
- Por que Rust?
- TDD em Rust

INTRODUÇÃO AO RUST

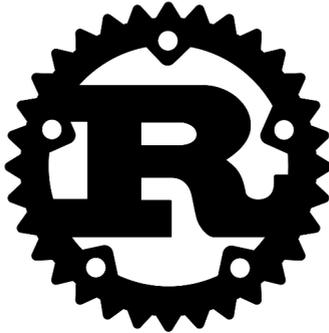


Figura 1.1: Logo da linguagem Rust

Rust é uma linguagem de sistema, como C e C++, com algumas características que a tornam muito diferenciada, além de possuir uma comunidade bastante ativa para uma linguagem tão jovem e *open source*. Algumas dessas características marcantes são:

- Abstrações sem custo, pois é fácil abstrair com `traits`, especialmente com o uso de genéricos.
- Semântica de `move` (veremos bastante na *Parte 3*).
- Segurança de memória garantida.
- Trabalhar com múltiplas *threads* é fácil devido ao fato de ter segurança de memória, auxiliada pelo compilador na detecção de *data races*.

- Programação orientada a dados é simples, especialmente por ter *pattern matching*, imutabilidade por padrão e outras características funcionais.
- Inferência de tipo.
- Macros: regras e correspondência de padrões que se parecem funções, mas que aumentam muito a versatilidade do que podemos fazer com nossas funções.
- Fácil geração de executáveis binários com alta integração com diferentes sistemas, com um sistema de execução mínimo e *bindings* eficientes para diversas linguagens, especialmente C e Python.

O objetivo do Rust é ser *memory safe* (segurança a nível de memória) e *fearless concurrency* (concorrência sem medo). Ou seja, o acesso de dados não vai causar efeitos inesperados, permitindo desenvolver aplicações concorrentes sem se preocupar com acesso a informações inexistentes, ou acesso à mesma informação de pontos diferentes.

Além disso, a linguagem ganhou o prêmio de "*most loved programming language*" do *Stack Overflow Developer Survey*, em 2016 e 2017. Alguns princípios que servem como base para o Rust e seus reflexos já foram comentados: segurança sem coletor de lixo, concorrência sem corrida de dados, abstração sem *overhead* e um compilador que garante segurança no alocamento de recursos. Não se preocupe, ao longo do livro, especialmente no próximo capítulo, vamos ver estas características em detalhes.

Algo que considero muito importante para o contexto do livro é explicar o motivo do meu interesse em Rust. Sou desenvolvedora de jogos, principalmente C++ e C#, e de microsserviços,

principalmente em Clojure. Porém, desenvolver jogos performáticos em C/C++ com alto controle de memória raramente é seguro, e C# dificilmente é performático por conta própria, mesmo que seja bastante seguro.

Já no aspecto do Clojure, temos uma linguagem extremamente funcional com uma sintaxe limpa gerada pela sintaxe Lisp, e uma forma eficiente de trabalhar com macros e concorrência. Assim, sempre foi um grande desejo e uma necessidade uma linguagem segura, pouco verbosa e com aspecto funcional, a nível de sistema.

Para começar nossa trajetória e justificar nossa escolha por Rust, vamos à sua história.

1.1 HISTÓRIA DO RUST

A linguagem Rust surgiu de um projeto pessoal iniciado em 2006 pelo funcionário da Mozilla, Graydon Hoare. Segundo ele, o projeto foi nomeado em homenagem à família de fungos da ferrugem (*rust*, em inglês). Em 2009, a Mozilla começou a patrociná-lo e anunciou-o em 2010.

Inicialmente, o compilador escrito por Hoare era feito em OCaml, porém, em 2010, ele passou a ser auto-hospedado e escrito em Rust. O compilador é conhecido como *rustc*, e utiliza como back-end a LLVM (*Low Level Virtual Machine*, ou Máquina Virtual de Baixo Nível), tendo compilado com sucesso pela primeira vez em 2011 (KAIHLAVIRTA, 2017).

A primeira versão pré-alfa do compilador Rust ocorreu em janeiro de 2012. Já a primeira versão estável, Rust 1.0, foi lançada em 15 de maio de 2015. Após isso, tomou-se hábito lançar updates

a cada seis semanas. Fala-se que os recursos são desenvolvidos na ferrugem noturna (*Nightly Rust*), sendo testadas em versões betas que duram 6 semanas.

O estilo do sistema de objetos mudou consideravelmente em suas versões 0.2, 0.3 e 0.4, pois a 0.2 introduziu os conceitos de classes pela primeira vez. E com a versão 0.3, foi adicionada uma série de recursos, incluindo destrutores e polimorfismos por meio do uso básico de interfaces, bem como todas as funcionalidades ligadas a classes.

Em Rust 0.4, os *traits* foram adicionados como um meio para fornecer herança. As interfaces foram unificadas com eles e colocadas como uma funcionalidade separada. As classes também foram removidas, substituídas por uma combinação de implementações e tipos estruturados.

Começando em 0.9 e terminando em 0.11, o Rust tinha dois tipos de ponteiros embutidos: `~` (til) e `@` (arroba). Para simplificar o modelo de memória, a linguagem reimplementou esses tipos de ponteiros na biblioteca padrão com o `Box`. Por último, foi removido o garbage collector (KAIHLAVIRTA, 2017).

Em janeiro de 2014, o editor-chefe do Dr. Dobb comentou sobre as chances de Rust se tornar um concorrente para C++ e C, bem como para as outras linguagens próximas, como D, Go e Nim (então Nimrod): *"de acordo com Binstock, enquanto Rust era 'uma linguagem extraordinariamente elegante', sua adoção pela comunidade permaneceu atrasada, porque ela continuava mudando continuamente entre as versões"*.

Sobre as constantes mudanças

Rust começou com o objetivo de criar uma linguagem de programação de sistemas segura. Em busca deste objetivo, explorou muitas ideias, algumas das quais mantiveram vidas ou traços, enquanto outras foram descartadas (como o sistema *typestate*, o *green threading*).

Além disso, no desenvolvimento pré-1.0, uma grande parte da biblioteca padrão foi reescrita diversas vezes, já que os primeiros desenhos foram atualizados para melhor usar os recursos da Rust e fornecer APIs de plataforma cruzada de qualidade e consistente. Agora que Rust atingiu 1.0, o idioma é garantido como "estável"; embora possa continuar a evoluir, o código que funciona no Rust atual deve continuar a funcionar em lançamentos futuros.

Para concluir, Rust merece atenção por ser uma linguagem de alta performance, com número reduzido de bugs e características agradáveis de outras linguagens modernas em uma comunidade muito receptiva. Uma demonstração da preocupação da receptividade da comunidade é a seguinte frase sobre os materiais para aprendizado de Rust terem um alcance melhor:

"Alguns grupos que estão sub-representados nas áreas de tecnologia gostaríamos especialmente de incluir na comunidade Rust, como mulheres (cis & trans), pessoas não binárias, pessoas de cor, falantes de outros idiomas além do inglês, pessoas que aprenderam programação mais tarde na vida (mais velhos ou apenas na faculdade, ou em um bootcamp como parte de uma mudança de carreira da meia-idade), pessoas com deficiência ou pessoas que têm diferentes estilos de aprendizagem" (NICHOLS, 2017).

No próximo capítulo, entenderemos um pouco melhor por que Rust é uma ótima escolha e como podemos enxergar seus atributos de segurança a nível de memória, concorrência sem medo, sintaxe simples e tipagem segura.

Para saber mais

- Rust – <https://www.rust-lang.org/en-US/>
- Open Source – <https://github.com/rust-lang/rust>
- FAQs – <https://www.rust-lang.org/en-US/faq.html>
- LLVM – <https://en.wikipedia.org/wiki/LLVM>
- The History of Rust – https://youtu.be/79PSagCD_AY

POR QUE RUST?

Este capítulo trata bastante de questões técnicas de baixo nível. Caso você não tenha interesse neste momento, é um capítulo que pode ser lido ao final do livro.

Como explicado por Blandy, em seu livro *Why Rust?*, existem pelo menos 4 bons motivos para se utilizar essa linguagem:

1. **Tipagem segura (*Type Safety*)** – Isso quer dizer que Rust tem a preocupação de que sejam prevenidos comportamentos de erro de tipagem, causados pela discrepância dos comportamentos esperados para cada tipo. Na maioria das linguagens, isso fica a cargo do programador ou, em muitos casos, a linguagem é dinâmica, mas em Rust isso é uma funcionalidade dela própria.
2. **Entendimento da linguagem e sua sintaxe** – Rust possui uma sintaxe inspirada em outras linguagens, por isso ela é bastante familiar.
3. **Uso seguro da memória** – Linguagens como C/C++ possuem muito controle sobre o uso de memória e performance, sem segurança. Já linguagens como Java, que

utilizam coletores de lixo, praticam a segurança do uso de memória, mas sem controle de memória e performance. O Rust procura conciliar ambos com o uso da propriedade de referências, *ownership*.

4. **Programação concorrente** – Rust implementa soluções de concorrência que impeçam *data races*.

A execução de um programa contém *data races* se este possuir duas ações conflitantes em, pelo menos, duas threads diferentes.

Há algum tempo, aproximadamente duas gerações de linguagens atrás, começamos a procurar soluções mais simples em linguagens de alto nível – que evoluíam de forma muito rápida –, para resolver tarefas que antes eram delegadas a linguagens de nível de sistema. Felizmente, mesmo nas linguagens de baixo nível, houve uma pequena evolução. Além disso, alguns problemas existentes não são facilmente resolvidos de forma tão eficiente em linguagens de alto nível. Alguns motivos são:

- É difícil escrever código seguro. É comum explorações de segurança se aproveitarem da forma como o C e o C++ lidam com a memória. Assim, Rust apresenta uma baixa quantidade de bugs em tempo de compilação por causa das verificações do compilador.
- É ainda mais difícil escrever código concorrente de qualidade, que realmente seja capaz de explorar as habilidades das máquinas modernas. Cada nova geração

nos traz mais cores, e cabe às linguagens conseguirem explorar isso da melhor forma possível.

É neste cenário que Rust aparece, pois supre os defeitos que as linguagens anteriores têm dificuldade de evoluir.

2.1 TYPE SAFETY

É triste que as principais linguagens de nível de sistema não sejam type safe (C e C++) enquanto a maior parte das linguagens populares é, especialmente considerando que C e C++ são fundamentais na implementação básica de sistemas. Sendo assim, tipagem segura parece-me algo bastante óbvio para se ter em linguagens de sistema.

Esse paradoxo de décadas é algo que o Rust propõe-se a resolver: ao mesmo tempo que é uma linguagem de sistema, também é type safe. Conseguimos um design que permite performance, controle refinado de recursos e, ainda assim, garante a segurança de tipos.

Tipagem segura pode parecer algo simples e pouco promissor, mas essa questão torna-se surpreendente quando se analisa as consequências de programação concorrente. Além disso, concorrência é um problema bastante complexo de se resolver em C e C++, causando muita dor de cabeça. Desse modo, programadores focam em concorrência somente quando esta é absolutamente a única opção. Mas a tipagem segura e a imutabilidade padrão do Rust tornam código concorrente, muito mais simples, aliviando grande parte dos problemas de *data races*.

DATA RACES

Data races ocorrem quando, em um único processo, pelo menos duas threads acessam a mesma referência de memória concorrentemente, e pelo menos uma delas é de escrita sem que `locks` (bloqueios) sejam aplicados pela thread para bloquear a memória.

Safe e unsafe

Rust garante comportamentos muito seguros devido à sua tipagem estática, porém checagens de segurança são conservadoras, especialmente se o compilador tem de se encarregar disso. Para isso, é preciso fazer com que as regras do compilador fiquem mais flexíveis.

A keyword `unsafe` nos permite isso. Um exemplo de funções que devem ser marcadas como `unsafe` são as de FFI (Interfaces de Funções Externas). Existem 3 grandes vantagens de utilizar esse código:

1. Acessar e atualizar uma variável estática e mutável;
2. Diferenciar ponteiros brutos;
3. Chamar funções inseguras.

Um exemplo para acessar e atualizar valores é na forma de realizar operações inseguras que possam levar a problemas, como transformar valores do tipo `&str` em `&[u8]` (`std::mem::transmute::<T, U>(t: T)`).

```
fn main() {
    let u: &[u8] = &[10, 20, 30];

    unsafe {
        assert!(u == std::mem::transmute:::<&str, &[u8]>("123"));
    }
}
```

A lógica a seguir pode resultar em ponteiros vazios. Para não termos isto, precisamos diferenciar ponteiros brutos.

```
fn main() {
    let ponteiro_bruto: *const u32 = &10;

    unsafe {
        assert!(*ponteiro_bruto == 10);
    }
}
```

2.2 ENTENDIMENTO DA LINGUAGEM E SUA SINTAXE

Algo importante no desenvolvimento do Rust foi evitar uma sintaxe inovadora, portanto, várias partes da sintaxe serão bastante familiar. Considere as funções a seguir:

```
fn main() {
    println!("{:?}", 1 + 1);
}
```

Esta função apresenta uma função principal, `main`, também conhecida como `thread` principal, que executa um simples comando de imprimir na tela do console o resultado de `1 + 1`.

```
fn filtra_multiplos_de_seis(quantity: i32) -> Vec<i32> {
    (1i32..)
        .filter(|&x| x % 2i32 == 0i32)
        .filter(|&x| x % 3i32 == 0i32)
        .take(quantity as usize)
}
```

```
.collect::<Vec<i32>>()  
}
```

- A *keyword* `fn` introduz a definição de uma função, enquanto o símbolo `->` após a lista de argumentos define o tipo de retorno. Muitas linguagens de programação utilizam termos como `fun`, `def` e `fn` para definir funções e tipos de retorno especificados ao final da declaração da função, como o `:` (dois pontos) em Scala.
- Variáveis são imutáveis por padrão, assim, a *keyword* `mut` deve ser adicionada às variáveis para que elas possam ser resignadas.
- O valor `i32` representa um valor inteiro de 32 bit com sinal. Outras opções são o `u32`, que seria um valor de inteiro de 32 bits sem sinal. Outros tipos possíveis são `i8`, `u8`, `i16`, `u16`, `i32`, `u32`, `i64`, `u64`, `f32` e `f64`.
- A expressão `(1i32..)` representa uma sequência de valores que varia de `1i32` até infinito.
- As `filter()`, `take()` e `collect()` representam funções para iterar sobre a sequência, filtrando números divisíveis por 2 e por 3, tomando os 5 primeiros desta sequência e transformando-os em um vetor do tipo `i32`, respectivamente.
- Rust possui a *keyword* `return`, porém o compilador interpreta o último valor sem `;` (ponto e vírgula) como valor de retorno.

Na função a seguir, podemos ver a ausência de valores de retorno:

```
fn imprime_numero_se_chute_correto(chute: i32) {  
    let numero = 3i32;  
    assert_eq!(chute == numero);  
}
```

```
println!("0 numero {} está correto!", chute);
}
```

- A keyword `let` introduz variáveis locais, e o Rust infere tipos dentro de funções.
- Existem duas macros presentes nesta função: `assert_eq!`, que verifica se a condição é válida e, caso não seja, o programa é interrompido com um `panic`; e `println!`, que imprime o valor `chute` no lugar do `{}`.

2.3 SEGURANÇA DE MEMÓRIA

Agora precisamos ver o coração das reivindicações do Rust e o motivo pelo qual ele alega ter concorrência segura. Essa característica é definida por 3 princípios-chave:

1. Sem acesso a valores nulos, ou seja, sem dereferência de *null pointers*. Seu programa não vai ter um *crash* se você tentar uma dereferência a um *null pointer*.
2. Todos os valores estão associados a um contexto, ou seja, sem ponteiros soltos. Cada variável vai ter seu *lifetime* (tempo de vida) pelo tempo que for necessário viver. Portanto, seu programa nunca utilizará valores alocados na *heap* depois de ela ter sido liberada.
3. Sem *buffer overruns*. Seu programa nunca acessará elementos além do final ou antes do início de um array.

Dereferências de null pointer

Null pointer é quando existe um ponteiro para uma referência nula (vazia) na memória heap. Uma forma bastante simples de não permitir o uso de null pointer é nunca deixá-los serem criados.

Em Rust, não há o equivalente do `null` do Java, `nullptr` do C++ ou o `NULL` do C. Além disso, Rust não converte inteiros para ponteiro, logo, não é possível usar inteiros como se faz em C++. Portanto, essa linguagem não permite a introdução de *null pointers*.

Para que seja possível utilizar o conceito de valor vazio, que é muito útil, e não ter os problemas causados por null pointers, o Rust usa em sua implementação padrão o `enum Optional<T>` (<https://rustbyexample.com/std/option.html>). Um valor deste tipo não é um ponteiro, e tentar diferenciar é um *type error*. Somente quando verificarmos qual variação de `Optional<T>` tivemos, por meio da declaração `match`, poderemos encontrar o `Some(ptr)` e extrair o ponteiro `ptr`, que agora possui garantia de ser não nulo.

Pointeiros soltos

Programas Rust nunca tentam acessar valores alocados na heap após terem sido liberados. Isso não é algo incomum, todas as linguagens com garbage collectors (coletores de lixo, ou GC) garantem isso, em maior ou menor grau. O que o Rust faz de diferente é fazer isso sem GC e sem contagem de referências.

MEMÓRIA STACK E HEAP

A stack (ou pilha) é muito rápida e é onde a memória é alocada por padrão no Rust. Mas a alocação em stack é limitada à chamada de uma função, pois possui tamanho bastante limitado. A heap, por outro lado, é mais lenta e precisa ser explicitamente alocada. Ela tem tamanho efetivamente ilimitado, mas aloca blocos de memória de forma arbitrária.

GARBAGE COLLECTOR

GC é um processo para a automação do gerenciamento de memória pelo programa. Com ele, é possível recuperar uma área de memória sem ponteiros apontando para ela. Isso ajuda a evitar problemas de vazamento de memória, resultando no esgotamento da memória livre para alocação.

O GC foi inventado em 1959 por John McCarthy para resolver problemas de memória em Lisp. Uma linguagem que opta por ter coletor de lixo é uma que escolhe possuir uma maior e mais complexa operação de *runtime* que o Rust. Rust é usado direto no hardware, sem runtime extra, e coletores de lixo são usualmente a causa de comportamentos não determinísticos.

Para evitar esses problemas, Rust possui 3 regras para determinar quando um valor é liberado, e garantir que não exista mais nenhum ponteiro quando se atinjam esses pontos. Outro fator interessante é que essas regras são resolvidas em tempo de compilação.

- **Regra 1:** *todos os valores possuem um único dono (owner) em um dado momento.* Um valor pode ser movido (*move*) de um owner para outro, porém, quando esse owner desaparecer, o valor será liberado.
- **Regra 2:** *you can borrow (pegar emprestada) a referência a um valor, desde que ela não sobreviva mais que o dono.* Referências emprestadas são ponteiros temporários, permitindo operar sobre valores que você não possui.
- **Regra 3:** *you can only mutate a value when you have exclusive access to it.*

Vale fazer um adendo sobre a cópia de tipos específicos em Rust, pois existem duas categorias. O primeiro caso é quando podemos copiar bit a bit, sem tratamento extra. Isso pode ser feito por meio de implementações especiais do trait `Copy`.

Utilizar cópias permite que a referência mantenha seu dono sem que a cópia sofra com a função `drop` pelo meio do caminho. A trait `Copy` não é designada automaticamente para tipos novos pela anotação `#[derive(Copy, Clone)]` ou `impl Clone for Tipo`. O segundo caso é para todos os outros tipos, que devem ser movidos por designação e não podem ser copiados.

Buffer overrun

Um *buffer overrun* acontece quando um software excede o uso

de memória resignado a ele pelo sistema operacional, passando então a escrever no setor de memória contíguo. Em outras linguagens de nível de sistema, como C e C++, não se indexa efetivamente arrays, mas sim ponteiros, que contêm informações do início e fim do array, ou de quaisquer outros objetos relacionados.

Isso faz com que os programadores sejam responsáveis por verificar as fronteiras do array, e eles usualmente cometem pequenos erros nesse sentido. Em Rust, não se indexam ponteiros, mas sim arrays e slices, que possuem fronteiras bem definidas. Isso garante que ataques como o vírus Morris (primeiro vírus do tipo verme distribuído pela internet, que explorava buffer overruns) não aconteçam e que as pessoas não possam explorar falhas de buffers.

2.4 PROGRAMAÇÃO CONCORRENTE

Após termos explicado `type safety` e `ownership`, podemos apresentar o grande diferencial do Rust: programação concorrente sem `data race`. Isso tem o objetivo de permitir que a concorrência seja o primeiro recurso do seu programa, e não o último.

Para um entendimento claro, a concorrência ocorre quando existem dois ou mais processos executados em simultâneo, porém, logicamente independentes.

Criação de threads

A primeira etapa antes da sincronização de threads é ver como criá-las. Esse tópico será explorado mais a fundo durante o livro,

mas agora vamos fazer uma apresentação básica. Para isso, utilizamos a função `std::thread::spawn` com uma closure (mais sobre funções e closures na *Parte 2*) de argumento. Imagine uma thread que retorna o poder de Goku após a medida de ki:

```
let thread_poder = std::thread::spawn(|| {
    println!("Medindo o ki");
    return 10000;
});
```

Essa thread imprime uma string e encerra retornando um `JoinHandle`, um valor que espera o método `join()` para encerrá-la. Assim, para medir se o ki é mais de 8000, faríamos o seguinte:

```
let thread_poder = std::thread::spawn(|| {
    println!("Medindo o ki");
    return 10000;
});
assert!(try!(thread_poder.join()) >= 8000);
```

O que aconteceria se o Goku tentasse fazer um kaioken 10x ao mesmo tempo que um kaioken 20x?

```
let mut poder_goku = 1;
let kaioken_10x = std::thread::spawn(|| { poder_goku *= 10 });
let kaioken_20x = std::thread::spawn(|| { poder_goku *= 20 });
```

O compilador Rust proíbe com o seguinte erro: `error: closure may outlive the current function, but it borrows 'x', which is owned by the current function.`

Como nossa closure utiliza o poder de Goku de seu ambiente externo, Rust trata a closure como uma estrutura de dados que pegou emprestado (*borrow*) a referência mutável para `poder_goku`. O erro acontece, pois o Rust não consegue ter certeza sobre a qual função `poder_goku` pertence neste

momento, assim o compilador impede que estas duas threads ocorram.

Uma solução seria usar `std::thread::scoped`, que cria um `JoinGuard` em vez de uma `JoinHandle`. A diferença é que a `JoinGuard` tranca a thread até que ela termine, impedindo que ela dure mais que sua `JoinGuard` permite.

Como entraremos em mais detalhes sobre concorrência no futuro, vale passar aqui apenas um pequeno resumo de uma de suas maiores vantagens, os `Mutex`. Quando múltiplas threads devem ler ou modificar dados compartilhados, elas precisam de cuidados redobrados para garantir sincronia entre elas. Portanto, uma maneira de proteger estruturas de dados é utilizar `Mutex`.

Somente uma thread poderá prender (*lock*) a mutex por vez. Assim, a thread acessará a estrutura somente quando trancar a mutex pelo *lock*. Os passos de *lock* e *unlock* que a thread realiza servem para sincronizar a operação, impedindo comportamentos indefinidos.

2.5 MAIS SOBRE RUST

Apesar de ser uma linguagem nova, ela é bastante robusta, tem features importantes e uma comunidade muito ativa em seu desenvolvimento, que veremos um pouco adiante no livro. Veja a seguir algumas características importantes:

- Bibliotecas completas de coleções com sets, maps, vectors, sequências e assim por diante. A biblioteca padrão já vem com implementações muito boas de coleções, o que permite um ótimo desenvolvimento no nível de

programação funcional e estrutura de dados.

- Apoio ao desenvolvimento de macros, como as apresentadas anteriormente `println!()` e `assert!()`. O Lisp e o Clojure são linguagens que permitem muita flexibilidade e abstração com macros poderosas, já as macros do Rust estão muito mais próximas da liberdade que o Lisp nos dá do que o C++ nos permite.
- Ótimo sistema de módulos. Cargo, o gerenciador de dependências, apoia muito o desenvolvimento de libs, utilização de crates e sistemas de módulos isolados que podem ser consumidos de forma pontual.
- Gerenciador de crates (dependências) cargo. As crates de Rust localizam-se no `crates.io`, e o cargo auxilia em seu gerenciamento através do `cargo.toml` e na compilação, execução e testes de suas bibliotecas e programas.

Agora que já sabemos por que usar Rust, no próximo capítulo vamos para um exemplo de como aplicar TDD nessa linguagem.

Para saber mais

- Ownership – <https://github.com/bltavares/presentations/blob/gh-pages/rust-tipos-e-ownership/rust-tipos-e-ownership.org>
- Genéricos – <https://doc.rust-lang.org/book/first-edition/generics.html>
- Enums – <https://doc.rust-lang.org/book/first-edition/enums.html>
- Panic – <https://doc.rust-lang.org/book/second-edition/ch09-01-unrecoverable-errors-with-panic.html>

TDD EM RUST

Antes de entrar em detalhes sobre TDD em Rust, quero contar uma história sobre isso. Não faz muito tempo que postei em uma comunidade Rust um questionamento sobre por que existem tão poucas bibliotecas com testes em Rust, mesmo a linguagem possuindo um suporte nativo para isso.

Outra coisa que comentei foi a falta de crates para melhorar a experiência de testes. A resposta me surpreendeu de tal forma que tive de deletar o comentário de tão pasma que fiquei: "Como Rust é uma linguagem segura, não precisamos de testes para garantir qualidade e segurança".

Tendo feito esse pequeno disclaimer, vamos ao motivo de usarmos TDD.

3.1 POR QUE TDD?

TDD (*Test-Driven Development*, ou desenvolvimento guiado a testes) é um ciclo de desenvolvimento de software que parte do princípio "teste primeiro, code depois". Mas é claro que não se

trata só disso, e a figura a seguir é um exemplo do ciclo de desenvolvimento de software por TDD.

Não é função deste livro ensinar sobre TDD, mas creio que seja importante gerar a mentalidade de que, mesmo para linguagens seguras, testes são importantes, já que Rust não lhe permite causar problemas sem querer, mas permite causar problemas por querer.

Então, voltando à pergunta, por que TDD? Porque ele nos permite quebrar o loop de feedback negativo ou consecutivo, e possibilita manter um custo de desenvolvimento constante com alguns ganhos.

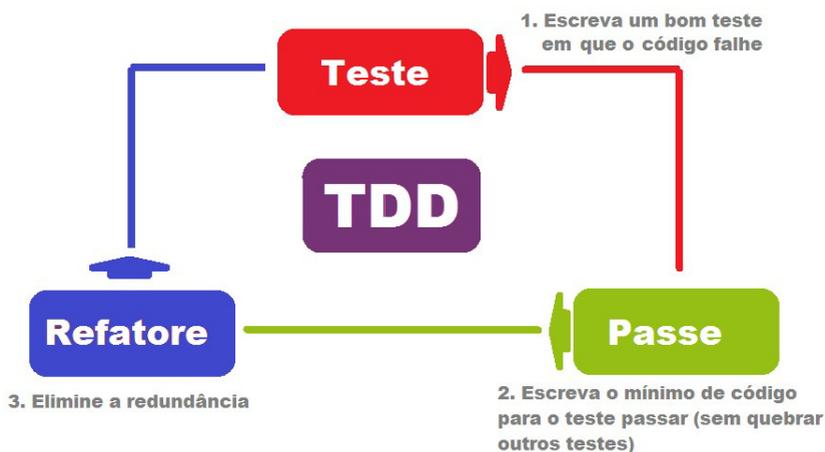


Figura 3.1: Ciclo do TDD

No livro *The Rust Programming Language*, encontramos a seguinte citação de Edsger W. Dijkstra, em *The Humble Programmer* (1972):

Teste de programas pode ser uma maneira muito eficiente para encontrar bugs, mas desesperadamente inadequada para mostrar a falta deles.

Esta frase é bastante antiga, e o mindset de testes já evoluiu muito. O que a frase representa para mim é o fato de que testes garantem que bugs não acontecerão quando seus "casos de teste" estiverem cobertos.

Entretanto, é impossível determinar se o nosso código não tem bugs mesmo sendo testado, pois não podemos prever a existência de bugs para todos os casos. Por isso, práticas como o TDD nos ajudam a reduzir bugs, especialmente os semânticos, pois os sintáticos nos são mostrados pelo compilador, já que o design e o desenvolvimento do código ficam restritos aos testes.

Você pode ver mais informações no livro desenvolvido pela equipe do Rust, em: <https://doc.rust-lang.org/book/second-edition/ch11-00-testing.html>.

Alguns possíveis benefícios

- **Bom critério de aceitação:** é fácil entender testes e ver se estão validando os casos previstos. O desenvolvimento guiado a testes garante que todo novo código precisa passar em todos os testes, o que nos permite escrever código novo sem quebrar o programa.
- **Foco:** como já sabemos quais testes devemos implementar, o risco de perder a atenção diminui consideravelmente.

Fica fácil prever quais passos devemos satisfazer, evitando que tentemos implementar coisas desnecessárias.

- **Refatorar:** TDD garante muita segurança na hora de refatorar. A refatoração pode mudar o estilo do código, aplicando simplificações, mas não pode quebrar testes ou adicionar lógica sem teste.
- **Menos bugs:** ainda há discussões sobre a veracidade desse fato, mas, pela minha experiência, parece ser verdade. Também é uma referência ao argumento apresentado anteriormente.
- **Documentação atualizada:** cada nova implementação exigirá novos testes, por isso é fácil verificar o que queremos que o software faça. Testes representam o atual estado do código, tornando obsoleta a necessidade de escrever centenas de linhas de documentos para a compreensão do código.

3.2 UM EXEMPLO EM RUST

Este livro pressupõe que você já conheça o básico de Rust, por isso vou concentrar as informações na parte relacionada a testes. Uma forma de começar a pensar em Rust e TDD é utilizar o comando `cargo new <nome da lib> --lib`, pois gera um template de `lib` que começa pelos testes.

Nosso problema será bem simples: encontrar os divisores de um número inteiro maior que zero. Assim, ele deve ter um teste que garanta um `panic!()` caso o número for 0. Quando criamos uma `lib` em Rust, nosso arquivo `src/lib.rs` inicia já preparado para receber testes:

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
    }
}
```

Em Rust, costumamos criar um módulo de testes, para testes unitários, e uma pasta de testes, chamada `tests`, para testes de integração. Neste caso, vamos nos preocupar somente com os unitários, devido à simplicidade do problema.

A anotação `#[cfg(test)]` representa que este módulo deve ser compilado somente para testes, e a anotação `#[test]` representa que a função a seguir é um teste a ser rodado. Os testes em Rust funcionam da seguinte maneira:

- Se tudo ocorrer bem, o teste passa, como a função `it_works()` nos mostra.
- Se houver um `panic!`, o teste falha.
- Mas e quando queremos testar um `panic!`? Utilizamos a anotação `#[should_panic]`.

Nosso primeiro teste será o seguinte:

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic]
    fn zero_is_not_valid() {
        divisores(0);
    }
}
```

Primeiro precisamos rodar o comando `cargo test`, que fará um build e verificará se os testes passam. Para passar este teste,

adicionamos a função a seguir. Ela retorna um monte de avisos de mau uso gerados pelo compilador em relação à variável `valor`, inclusive avisando quais funções que nunca foram utilizadas:

```
fn divisores(valor: i64) {
    panic!("0 não é um valor válido");
}
```

Agora, com o teste, queremos testar outro valor menor que zero:

```
#[test]
#[should_panic]
fn negativos_nao_sao_validos() {
    divisores(-10);
}
```

Felizmente, ele passa, mas a mensagem está estranha. Uma boa possibilidade é mudar para `{ } <= 0 não é valido, valor`. Também vale a pena adicionar a anotação `# [allow(dead_code)]` no método `divisores`, pois assim o compilador entenderá que é uma decisão consciente.

Agora vamos adicionar um teste que faz o compilador falhar:

```
#[test]
fn dois_divide_dois() {
    assert_eq!(divisores(2), vec![2])
}
```

A mensagem de erro é a seguinte: `expected (), found struct std::vec::Vec`. Para corrigir isso, precisamos garantir que a função `divisores` retorne um `Vec<i64>`.

```
#[allow(dead_code)]
fn divisores(valor: i64) -> Vec<i64> {
    if valor <= 0 {
        panic!("{ } <= 0 não é valido", valor);
    } else {
```

```

        vec![2]
    }
}

```

Agora percebemos que não estamos retornando 1, por isso adicionaremos um teste que garante `panic!` quando for passado 1 como argumento. Faremos isso por motivos de simplicidade e não complexidade da teoria dos números.

```

#[test]
#[should_panic]
fn um_nao_eh_valido() {
    divisores(1);
}
...
#[allow(dead_code)]
fn divisores(valor: i64) -> Vec<i64> {
    if valor <= 1 {
        panic!("{}", "1 não é valido", valor);
    } else {
        vec![2]
    }
}
}

```

Agora precisamos garantir que os divisores de 3 sejam apenas [3].

```

#[test]
fn tres_divide_tres() {
    assert_eq!(divisores(3), vec![3])
}

```

Nosso vetor de retorno é [2], mas esperávamos [3]. O seguinte código nos permite resolver esse problema:

```

fn divisores(valor: i64) -> Vec<i64> {
    if valor <= 1 {
        panic!("{}", "1 não é valido", valor);
    } else {
        vec![valor]
    }
}
}

```

E quanto ao 4 ?

```
#[test]
fn divisores_de_quatro() {
    assert_eq!(divisores(4), vec![2, 4])
}
```

Este teste falha por retornar apenas [4] , mas a solução para este problema não é complexa:

```
#[allow(dead_code)]
fn divisores(valor: i64) -> Vec<i64> {
    if valor <= 1 {
        panic!("{}", <= 0 não é valido", valor);
    } else {
        (2..valor + 1).filter(|x| valor % x == 0).collect::<Vec<i64>>()
    }
}
```

Agora vamos testar uma coleção mais complexa:

```
#[test]
fn divisores_de_vintequatro() {
    assert_eq!(divisores(24), vec![2, 3, 4, 6, 8, 12, 24])
}
```

Nosso código ainda é válido. Perceba que a solução do teste é bastante funcional: `(2..valor + 1).filter(|x| valor % x == 0).collect::<Vec<i64>>()` . Isso nos leva à próxima parte: programação funcional em Rust.

Programação funcional

- O que é programação funcional?
- Definindo funções
- Traits
- if let e while let
- Iterators, adapters e consumers

O QUE É PROGRAMAÇÃO FUNCIONAL?

Veja um *approach* baseado em Clojure.

DISCLAIMER

1. Rust não é uma linguagem do paradigma funcional, mas se aproveita de muitos aspectos dele.
2. Escolhi comparar com Clojure por ser a linguagem funcional que mais tenho domínio e teoria para suportar minhas ideias.

Definição de programação funcional

Pessoalmente este é um dos tópicos com qual mais me divirto: "a sagrada definição de funcional." Há desde pessoas falando que Java 8 é funcional até outras procurando definições claras do que é programação funcional. Para mim, uma das referências mais importantes foi no livro *The Joy of Clojure*, no qual a primeira frase sobre programação funcional é a seguinte:

Rápido, o que é programação funcional? *Resposta errada.*

A parte relevante desta pequena frase de impacto é que ninguém sabe o que é programação funcional, e não há uma definição muito clara a respeito disso. É um daqueles termos da computação que tem uma definição amorfa. Cada pessoa vai ter sua definição e todas serão diferentes, mas o interessante será ver algumas características comuns.

Felizmente, cada pessoa vai puxar mais para a linguagem que prefere, como Haskell, ML, Racket, Shen e Clojure. O ponto interessante é que, no geral, funções serão a característica central das definições de programação funcional, sendo elas cidadãs de primeira classe no paradigma.

Para tanto, vou introduzir a definição mais próxima ao Clojure de funcional: pureza (funções puras), imutabilidade, recursão, *laziness* (algo como preguiça, falando especialmente de sequências) e transparência de referências.

Rust não tem funções puras e nem é o foco da linguagem, mas as variáveis são imutáveis por padrão. É possível trabalhar com sequências de forma *lazy*, e o forte do Rust é o modo como as referências são tratadas. Quanto à recursão, confesso que não é o forte da linguagem e pouco usei, mas existem duas soluções:

1. Definir uma função *inline* (que é uma definição de uma operação fatorial) e chamá-la internamente, aplicando o novo padrão na própria chamada – muito parecido com o

que podemos fazer em Clojure. Podemos ver no código a seguir que a função `fatorial` está sendo usada em sua própria definição.

```
fn main() {
  fn fatorial(valor: u32) -> u32 { if valor == 0 {1} else
  {valor * fatorial(valor - 1)} }

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

1. Definimos *closures* dentro de *structs* e passamos a *struct* para a *closure*, assim como a função anterior, mas podemos utilizar nossa struct `fatorial` para receber uma função anônima que opera sobre ela mesma. Na função a seguir, definimos uma struct que possui uma função como argumento. Podemos ver que esta recebe ela mesma e um tipo `u32` como argumentos, retornando um novo `u32`, `f: &'s Fn(&Fatorial, u32) -> u32`. Quando essa struct é definida em um `let`, vemos que ela recebe o valor desse `let` e um que é retornado.

```
fn main() {
  struct Fatorial<'s> { f: &'s Fn(&Fatorial, u32) -> u32
  }

  let fact = Fatorial {
    f: &|fact, valor| if valor == 0 {1} else {valor *
(fact.f)(fact, valor - 1)}
  };

  println!("{}", (fact.f>(&fact, 5));
}
```

Particularmente, acho a primeira solução muito mais elegante e "funcional", pois ela é mais simples e resolve o problema com menos *boilerplate*.

Boilerplate refere-se ao excesso de código que muitas linguagens precisam para conseguir fazer tarefas mínimas, ou à verbosidade necessária para atingir um estado. Em Orientação a Objetos (OO), é bem comum chamar *getters* e *setters* de boilerplate.

4.1 IMUTABILIDADE

Assim como Clojure, Rust provê imutabilidade como pilar central da linguagem, provendo associações imutáveis por padrão. É preciso explicitar que a variável é mutável, com `let mut var_mutavel = 3; ,` para poder alterar seu valor. Porém, para que serve a imutabilidade?

1. **Invariante:** provê garantias dos futuros estados do valor, impedindo que muitos estados indesejáveis aconteçam.
2. **Refletir sobre precauções:** o trabalho de pensar em como a variável vai estar e seus possíveis estados futuros não precisa ser resolvido nem pelo compilador, nem pela pessoa que desenvolve.
3. **Igualdade:** não faz sentido falar de igualdade de variáveis se elas são mutáveis, já que, se hoje elas são diferentes, provavelmente sempre serão – e mesmo que sejam iguais, é muito provável que no futuro deixem de ser. Quando falamos de imutabilidade, podemos garantir que, se dois objetos são iguais hoje, eles sempre serão iguais.
4. **Compartilhar:** no caso de programação concorrente, temos a garantia de que esse valor não será mutado pela thread, e

seu valor poderá ser usado para outros fins com mais liberdade.

4.2 LAZINESS

Clojure é uma linguagem parcialmente *lazy* na maneira como lida com suas sequências. Mas o que isso realmente significa? A separação entre *lazy* e *eager* (impaciente) é bastante simples:

- Lazy deixa para avaliar os argumentos somente quando necessário.
- Eager avalia os argumentos assim que eles são passados.

Temos o Haskell como exemplo de uma linguagem extremamente lazy. Mas linguagens eagers também possuem seus momentos lazy, e isso não as transforma em lazy. Um exemplo disso é o operador `&&` do Java.

No código a seguir, temos um `if` com duas condições que devem ser verdadeiras: a primeira verifica se o objeto existe, e a segunda verifica se o método retorna `true`. Veja:

```
if (obj != null && obj.isWorking()) {  
    ...  
}
```

O Java deixa para avaliar o resultado da expressão `&&` após validar se `obj != null`, ou seja, um comportamento lazy. O mesmo acontece em Clojure com a keyword `and`:

```
(defn grande-and [x y z]  
  (and x y z (do (println "tudo verdade") :verdade)))  
  
(grande-and () 100 true)  
; "Tudo verdade"
```

```
; => :verdade
(grande-and true false true)
; => false
```

A função clojure definida, em `defn` como `grande-and`, recebe 3 argumento `x`, `y`, `z`. E caso todos eles sejam `true`, imprime "Tudo verdade" e retorna a keyword `:verdade`.

Podemos ver que, no primeiro uso de `grande-and`, todos os casos foram avaliados como verdade, portanto, o `and` foi avaliado como verdade. Já o segundo caso retornou `false`, pois a avaliação do segundo item já retornou falso, interrompendo a sequência de validação. Rust também possui sua sequência lazy, o `Range`, escrito `(inicio..fim)`.

4.3 FUNÇÕES

O cerne da programação funcional é baseada em um sistema computacional conhecido como *lambda calculus*. Em Clojure, assim como em Rust, funções são cidadãs de primeira classe e podem ser passadas como argumento e como valor de retorno.

Além disso, Rust possui todas as atribuições relacionadas a closure e traits. O pecado de Rust nesse sentido são as funções puras. Java lida com funções de forma diferente, em que tudo deve ser resolvido em objetos, forçando todas as modelagens de comportamento a serem instâncias de classes. Em oposição ao paradigma do Java, Clojure trata funções como dados, permitindo que 4 itens importantes estejam fortemente presentes em Rust. As funções podem ser:

- Criadas sob demanda;

- Armazenadas em estruturas de dados;
- Passadas como argumento para funções;
- Retornadas como valores de funções.

Agora podemos entender melhor como Rust lida com funções, pois, como mencionado anteriormente, elas são o aspecto central da programação funcional e serão vistas com maior profundidade no próximo capítulo.

DEFININDO FUNÇÕES

A primeira função que você verá em Rust é a `fn main()`, que provavelmente foi gerada quando você rodou o comando `cargo new meuapp --bin`, ao brincar com a linguagem. E ela virá na seguinte forma:

```
fn main() {  
    println!("Hello, world!");  
}
```

Essa é a função central da operação de um projeto em Rust, e equivale ao `main` em outras linguagens como Clojure, Java, C/C++, C#. Funções são definidas pela palavra reservada `fn`, seguidas pelo nome da função, de preferência em `snake_case` – caso você erre os padrões de estilo, o *cargo* avisará –, e parênteses `()` (ainda muito semelhante a muitas linguagens). Após isso, virão as chaves `{...}` com o corpo da função.

Note que o Rust tem uma sintaxe muito parecida com várias linguagens. O objetivo disso foi torná-lo mais fácil de aprender.

No capítulo *TDD em Rust*, já apresentamos algumas macros como `assert_eq!` e `panic!`, porém temos mais três interessantes para falar: o `print!` e o `println!`, que são macros que imprimem valores no console, e o `assert!`, que funciona muito como o `assert_eq`. A diferença é que este recebe dois

valores e compara-os, enquanto o `assert!` recebe uma expressão lógica como `assert!(expected >= actual)`.

Neste livro, vamos utilizar as principais macros. Porém, se você deseja ver mais possibilidades, o livro *The Rust Programming Language* contém um capítulo bastante sólido sobre macros. Para mais, acesse: <https://doc.rust-lang.org/book/first-edition/macros.html>.

Agora queremos que nossa função receba um valor inteiro. Rust possui algumas características interessantes a respeito de inteiros e floats, como seu modo de definir o tipo específico de inteiros e floats. Nossa função `outra_func` deverá receber um valor inteiro do tipo `i32` e imprimi-lo na tela:

```
fn outra_func(x: i32) {  
    println!("{}", x);  
}
```

Podemos ver que o argumento recebido pela função foi `x` do tipo `i32`. Com dois argumentos, nossa assinatura da função ficaria assim: `fn outra_func(x: i32, y: u64) {`. No corpo das funções, deparamo-nos com duas situações: declarações e expressões.

- Declarações são instruções que realizam alguma ação, mas não retornam valores. Criar uma variável é uma declaração `let x = "declaracao"`. Declarações duplas não são permitidas, pois elas não retornam valor, como em `let x = (let z = "Nao permitido")`.
- Expressões avaliam para retornar valores. Expressões são como a gerada a seguir, na qual o bloco avalia para retornar 2:

```
{
  let x = 1;
  x + 1
}
```

Agora digamos que a função `pi()` retorna um float do tipo `f32`. Para isso, podemos escrever essa função dessa forma:

```
fn pi() -> f32 {
  3.1415f32
}

fn main() {
  let pi = pi();
}
```

O símbolo `->` define o tipo de valor de retorno e, caso ele não esteja presente, a função não tem valor de retorno. Neste caso, estamos retornando um `f32`. A falta de `;` depois do `3.1415f32` é intencional, pois esta é a forma com a qual o Rust retorna valores (a última linha que não possui `;` e é um valor).

Felizmente, há outras formas de retornar valores em Rust, como o uso da palavra `return`, e você verá em alguns programas a palavra `ret` também. Veja o exemplo seguinte:

```
fn pos(x: i32) -> i32 {
  if x > 0 {
    return x;
  } else {
    1
  }
}
```

Retornar valores com o `return` no fim da função é considerado um péssimo estilo, e o compilador reclamará:

```
fn pi() -> f32 {
  return 3.1415f32;
}
```

Um último tópico geral sobre funções são atribuições de variáveis que apontam para funções. Digamos que temos a função `inc(x: i32) -> i32`; ela incrementa o valor de `x`, um `i32`, e retorna um novo valor `i32`:

```
fn inc(x: i32) -> i32 {  
    x + 1  
}
```

Existem duas maneiras de atribuirmos essa função a uma variável. A primeira delas é com inferência, `let f = inc;`, e a segunda é sem inferência, `let f: fn(i32) -> i32 = inc;`. Agora podemos chamar a função `f` livremente:

```
fn inc(x: i32) -> i32 {  
    x + 1  
}
```

```
let f = inc;  
let dois = f(2);
```

Creio que este seja o primeiro passo para considerar funções em Rust como cidadãs de primeira classe. O próximo é entender como elas utilizam outras funções.

5.1 FUNÇÕES DE ORDEM SUPERIOR

Como já atribuímos uma função a um parâmetro, agora precisamos poder fazer algo com isso. Quem sabe passar essa função como parâmetro? É nessa hora que entram as funções de ordem superior:

```
fn ordem_superior<F>(valor:i32, func: F) -> i32  
    where F: Fn(i32) -> i32 {  
    func(valor)  
}
```

Essencialmente, isto é uma função de ordem superior, mas a parte interessante é o tipo do segundo parâmetro, `func`. `F` é um tipo genérico definido dentro da cláusula `where`. A cláusula `where` é usada para fazer a ligação entre os tipos utilizados e os tipos genéricos.

No nosso exemplo, a ligação de tipo do `F` é o tipo `FN`, ou seja, um trait para o tipo função, que nesse caso recebe um argumento do tipo `i32` e retorna outro argumento do tipo `i32`.

Lembra da nossa função `inc()`? Ela é uma do tipo `FN`, que recebe um `i32` como argumento e retorna um `i32`. Sendo assim, podemos aplicá-la na nossa função `ordem_superior`. Vamos primeiro criar um teste para verificar a veracidade disso:

```
#[test]
fn funn_ordem_superior() {
    assert_eq!(onze, ordem_superior(10i32, inc))
}
```

Agora que validamos o teste, podemos aplicar o conhecimento:

```
let onze = ordem_superior(10i32, inc);
```

Utilizamos nosso conhecimento com sucesso, mas parece necessário gerar funções anônimas – ou, como chamamos em Rust, closures – para podermos passá-las como argumento para `ordem_superior`. É significativamente comum passar funções anônimas para outras funções, mas o caso que mais vejo é passar closure para funções da `std::iter::Iterator`, como o `map()`, `filter()`, `fold()`, `scan()`.

5.2 FUNÇÕES ANÔNIMAS

Funções anônimas (ou closures) são um grupo especial de funções do Rust e de muitas outras linguagens, que não possuem uma definição estática, sendo geradas inline. Uma closure parece assim:

```
let inc = |x: i32| x + 1;

assert_eq!(2, inc(1));
```

Mais especificamente, a parte `|x: i32| x + 1` representa a closure que adiciona `1` ao valor `x` do tipo `i32`. Os argumentos passados para a closure vão entre `| |`, e é possível ter closures com mais de uma linha utilizando `{ }`.

```
let inc_duplo = |x: i32| {
    let mut resultado = x;

    resultado += 1i32;
    resultado += 1i32;

    resultado
};

assert_eq!(3, inc_duplo(1));
```

Uma questão interessante a se observar é que o tipo do retorno não está explícito na closure, e poderia não estar explícito no tipo do argumento. Isso ocorre pois, enquanto é útil saber os tipos das funções declaradas – já que auxilia com documentação e inferência de tipos –, as closures não são documentadas por serem anônimas, logo, não causam os tipos de erros que funções declaradas causam. Uma comparação interessante que pode ser feita em relação a declarações de closures é a seguinte:

```
fn multiplicar_pi_1 (x: f64) -> f64 { x * PI }
let multiplicar_pi_2 = |x: f64| -> f64 { x * PI };
let multiplicar_pi_2 = |x: f64|          x * PI ;
let multiplicar_pi_2 = |x|                x * PI ;
```

A primeira linha é uma declaração de função convencional, que recebe um argumento do tipo `f64` e retorna um `f64`, que é o argumento multiplicado por `PI`. As três linhas seguintes são funções anônimas na qual o grau de tipagem das variáveis foi reduzido, sendo a primeira a mais tipada e a segunda, a menos tipada.

Além disso, closures possuem algumas características interessantes: elas podem se relacionar com o ambiente na qual estão, podendo usar conceitos de *borrow* (emprestar). A listagem a seguir mostra uma closure acessando uma variável externa a ela:

```
let num = 10i16;
let inc_num = |x: i16| x + num;

assert_eq!(20i16, inc_num(10i16));
```

Move

Uma keyword muito comum em Rust é a `move`, que serve para garantir que a closure copie a *ownership* de seu ambiente externo para dentro dela. Mas por que utilizamos o `move` ?

Imagine o seguinte cenário na listagem anterior: `num` é imutável e foi passado como referência imutável para `inc_num`. Se tentarmos criar uma nova referência *mutável*, o compilador vai gerar um erro reclamando que a variável já foi movida por um *borrow* (empréstimo) imutável e não permitirá *borrow*s mutáveis. Para resolver este tipo de situação, utilizamos o `move`, que permite ao `num` implementar `Copy` e faz com que `inc_num` receba uma cópia de `num`.

Agora podemos continuar nossa linha de raciocínio de funções de ordem superior, retornando funções de funções.

5.3 FUNÇÕES COMO VALORES DE RETORNO

Para tornar nossas funções *cidadãs de primeira classe*, precisamos completar o ciclo de uso delas. Já sabemos como passar funções como argumento, como atribuir funções a variáveis e como declarar funções anônimas. Agora falta apenas aprender a retorná-las.

Confesso que a solução disso é um pouco confusa inicialmente, mas bem simples de entender com um mínimo esforço. Em um primeiro momento, é de se imaginar que, para indicar que o tipo de retorno é uma função, uma possível sintaxe seria apenas com a `trait Fn`, como o exemplo a seguir:

```
fn finc() -> (Fn(i32) -> i32) {
    |x| x + 1
}
```

Mas, infelizmente, a solução é um pouco mais verbosa e precisa fazer uso da `trait Box<>` com `Fn`:

```
fn finc() -> Box<Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}
```

Box

`Box` é um ponteiro inteligente (*smart pointer*), que permite você colocar um valor na heap. Podemos colocar um valor `i32` nela, como mostra a listagem a seguir:

```
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

Isso imprimirá "b = 5" na tela. Neste caso, podemos ver que é bastante simples acessar valores de dados dentro de um box, e é muito semelhante a como faríamos com valores em stack. Assim como qualquer outro valor que possui ownership de dados, box também sai de escopo, como b quando a main() termina.

A deslocação acontece para todas as partes associadas ao Box. Colocar um único valor na heap não parece muito útil, mas, como já vimos anteriormente, os box permitem que coloquemos implementações da trait Fn na heap.

Já falamos bastante sobre traits, mas é importante vê-las mais profundamente. Faremos isso no próximo capítulo.

Retornando funções com o Box

Entretanto, e se tivéssemos alguma variável na função?

```
fn func_num() -> Box<Fn(i32) -> i32> {
    let num = 5;
    Box::new(|x| x + num)
}
```

Nos depararíamos com o seguinte erro:

```
error: closure may outlive the current function, but it borrows `
num`,
which is owned by the current function [E0373]
Box::new(|x| x + num)
          ^~~~~~
```

Esse problema ocorre porque nossa closure está pegando emprestado num, que se encontra no stack, portanto, possui o mesmo tempo de vida (lifetime) que o stack frame. Para resolver isso, podemos utilizar o move:

```
fn func_num() -> Box<Fn(i32) -> i32> {
```

```

let num = 5;
Box::new(move |x| x + num)
}

```

Outra questão importante de lembrar é que, se alguma variável vinda de argumentos é passada para a função de resposta, é importante declarar o seu lifetime:

```

fn retorno_ordem_superior<'a>(passo:& 'a i32) ->
    Box<Fn(i32) -> i32 + 'a > {
    Box::new(move |x: i32| x + passo)
}

```

Este capítulo teve como objetivo apresentar a flexibilidade das funções Rust e utilizá-las em um contexto de programação funcional. Um dos conceitos importantes a se usar aqui é evitar funções que retornem `void`, pois isso quebraria um fluxo de transformação de dados, por exemplo, que veremos melhor mais adiante.

Para saber mais

- *The Rust Programming Language* – <https://doc.rust-lang.org/book/>

TRAITS

Traits é uma das joias de Rust, e um dos pilares que permite a programação funcional nessa linguagem. Traits podem ser traduzidas por *características*. É graças a elas que podemos generalizar `Box` e implementar novas funções com velhos nomes.

Uma `trait` permite grande potencial de abstração em Rust. Ela possui somente uma descrição dos métodos, ou seja, sua declaração de tipos ou assinatura e nome, sem implementação real. Em um jogo, por exemplo, teríamos a `trait Inimigo` que possui a função `atacar`, algo parecido com:

```
trait Inimigo {  
    fn atacar(&self);  
}
```

Agora pense que temos 3 inimigos: o Azul, o Vermelho e o Verde. Para implementar a `trait` no inimigo Azul, faríamos assim:

```
impl Inimigo for Azul {  
  
}
```

Isso resulta em um erro que sugere que nem todos os métodos da `trait` foram implementados. Para resolver esse problema, basta implementar o método `atacar`:

```
impl Inimigo for Azul {
    fn atacar(&self) {
        println!("Ataque realizado com dano de {:?}", self.dano);
    }
}
```

Podemos ver que, para implementarmos uma trait para um tipo, utilizamos a sintaxe: `impl NomeDaTrait for Tipo {lista de funções}`. O `GitHub` (<https://github.com/GoDiStudios/mathf/blob/master/src/vector2.rs>) possui algumas implementações interessantes para vetores e `f32`. Como exemplo de uma trait mais completa, poderíamos ter os seguintes métodos para serem implementados quando a usarmos:

```
trait Inimigo {
    fn nova(vida: u32, dano: u32) -> self;
    fn atacar(&self);
    fn movimentar(&self, distancia: i32);
}
```

As funções que aparecem em traits são chamadas de métodos, e o motivo disso é a presença do `&self` como argumento. Ou seja, o objeto que as invocou é um parâmetro. Além disso, uma trait pode prover métodos com uma implementação padrão. Do jeito que temos falado aqui, parece que traits são exclusivas de structs, mas elas podem ser implementadas em qualquer tipo, e qualquer tipo pode implementar muitas delas.

Existe um grupo de traits bastante particular: aquelas que podem ser implementadas pela anotação `#[derive(...)]`, como a `#[derive(Debug, Copy)]`. São traits padrões para todos os novos tipos, e utilizar anotações para definir suas implementações foi uma forma simples de resolver um problema comum.

6.1 TRAIT BOUNDS

Trait bounds seriam as "promessas de comportamento de cada trait". Isso permite que as funções genéricas explorem os tipos que elas aceitam para implementar. Considere a seguinte função que não compila:

```
fn instancia<T>(inimigo: T) {
    println!("O inimigo foi instanciado na posicao {} ", inimigo.spawn());
}
```

Precisamos garantir que o tipo associado ao inimigo possa ser `Spawnable` (isto é, utilizar a função `spawn`). Isso pode ser feito como no exemplo a seguir:

```
fn instancia<T: Spawnable>(inimigo: T) {
    println!("O inimigo foi instanciado na posicao {} ", inimigo.spawn());
}
```

A sintaxe `<T: Spawnable>` significa *"qualquer tipo que implemente a trait `Spawnable`"*. Como as traits definem assinaturas de tipo de funções, podemos ter certeza de que qualquer tipo que implemente `Spawnable` tenha um método `.spawn()`. Um exemplo de como ficaria isso seria:

```
trait Spawnable {
    fn spawn(&self) -> Point2D;
}

struct Inimigo {
    altura: i64,
    peso: i64,
    forca: f64,
}

impl Spawnable for Inimigo {
    fn spawn(&self) -> f64 {
```

```

        Point2D::new() //exemplo fictício
    }
}

fn instancia<T: Spawnable>(inimigo: T) {
    println!("O inimigo foi instanciado na posicao {}", inimigo.spawn());
}

```

6.2 TRAITS EM TIPOS GENÉRICOS

Structs genéricas podem se beneficiar de *trait bounds*, pois permitem garantir segurança na hora da implementação de uma trait. Para fazer isso, basta adicionar a "promessa" quando declarar o tipo do parâmetro.

Para o tipo elipse, podemos implementar um método que verifica se ela é na verdade uma esfera. Para isso, criamos uma struct com os parâmetros correspondentes ao que se espera de uma elipse e, no método `eh_esfera`, aproveitamos da trait `PartialEq` para garantir a igualdade do tipo `T` correspondente a `a` e a `b`.

Nosso código declara uma struct `Elipse` que possui dois valores correspondentes ao raio, `a: T` e `b: T`, que, para serem uma esfera, devem ter o mesmo valor. Para que possamos verificar se `a` e `b` são iguais, precisamos garantir que o tipo `T` implemente a trait `PartialEq`:

```

struct Elipse<T> {
    x: T,
    y: T,
    a: T,
    b: T,
}

```

```
impl<T: PartialEq> Elipse<T> {
    fn eh_esfera(&self) -> bool {
        self.a == self.b
    }
}
```

Podemos perceber que o método `eh_esfera()` precisa averiguar que `a == b`, ou seja, checar a igualdade entre os lados. Para isso, é preciso que o tipo `T` implemente a trait `core::cmp::PartialEq`. Isso permite que uma elipse, ou um retângulo, seja definida em termos de um tipo que implemente essa trait de igualdade.

6.3 TÓPICOS ESPECIAIS EM TRAITS

Veja algumas observações sobre implementações de traits:

- Uma trait não pode ser usada e definida dentro do mesmo escopo. Portanto, para usar uma, é necessário utilizar a palavra `use`, que nos permite incorporar estruturas de dados e funções de outros módulos.
- A outra regra é bastante lógica: a implementação da trait ou do tipo deve estar no mesmo arquivo, ou seja, `impl + trait || impl + type`.
- É possível ter uma trait com mais de uma "promessa" (bounds), basta utilizar o símbolo `+` para somar as bounds, por exemplo, `fn foo<T: Clone + Debug>(x: T)`. A função dessa trait precisa de dois bounds, `Clone` e `Debug`.
- Métodos padrão podem ser implementados diretamente na trait, caso se apliquem a todos os casos. A trait a seguir

demonstra esse conceito:

```
trait Foo {
    fn eh_valido(&self) -> bool;

    fn nao_eh_valido(&self) -> bool { !self.eh_valido()
}
}
```

- É possível ter bounds diferentes para tipos diferentes, conforme a listagem referenciada a seguir. Infelizmente, as coisas podem ficar meio complexas, e pode ser necessário utilizar a cláusula `where` em uma das duas maneiras apresentadas.

```
//Primeira possibilidade
fn foo<T: Clone, K: PartialEq + Debug>(x: T, y: K);

//Cláusula where simples
fn foo<T, K>(x: T, y: K) where T: Clone, K: PartialEq +
Debug;

//Cláusula where segunda opção
fn foo<T, K>(x: T, y: K) where
    T: Clone,
    K: PartialEq + Debug;
```

- Algumas traits repetitivas podem ser implementadas de forma mais simples, valendo-se da anotação `# [derive(...)]`. A lista é pequena, mas muito poderosa: `... = Clone, Copy, Debug, Default, Eq, Hash, Ord, PartialEq, PartialOrd`.
- Tipos associados são uma arma poderosa na implementação de traits. Eles são uma forma de associar uma reserva de tipo dentro de traits, especialmente na assinatura de métodos. A implementação da trait vai especificar o tipo específico que será usado em conjunto.

```

//exemplo retirado do https://doc.rust-lang.org/book/se
econd-edition
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}

```

A trait `Iterator` anterior define a função `next` para o tipo `Item`. Assim, o resultado da implementação de `Iterator` para `Counter` poderia ser assim, com a definição de `Item` sendo do tipo `u32`:

```

//exemplo retirado do https://doc.rust-lang.org/book/se
econd-edition
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {

```

- Além disso, é possível usar `trait` para fazer sobrecarga de operadores (*operator overload*). Rust não permite a criação de operadores particulares ou todos os que podem ser encontrados em outras linguagens, mas permite implementações deles por meio da biblioteca `std::ops`. Isso se dá pela implementação das traits presentes em `std::ops`. O exemplo a seguir foi retirado do repositório <https://github.com/GodiStudios/mathf>, e remete a sobrecarga dos operadores `ADD` e `Mul` para vetores 2D (`Vector2`):

```

#[derive(Clone, PartialEq, Debug)]
pub struct Vector2 {
    x: f32,
    y: f32,
}

//...

```

```

impl ops::Add for Vector2 {
    type Output = Vector2;

    //Implementa a trait '+' para a struct Vector2
    fn add(self, new_vec: Vector2) -> Vector2 {
        Vector2 {x: self.x + new_vec.x, y: self.y + new_vec.y}
    }
}

impl ops::Mul<f32> for Vector2 {
    type Output = Vector2;

    //Implementa a multiplicação de um Vector2 por um escalar.
    fn mul(self, value: f32) -> Vector2 {
        Vector2 {x: self.x * value, y: self.y * value}
    }
}

impl ops::Mul<Vector2> for Vector2 {
    type Output = f32;

    //Implementa o produto escalar de dois Vector2 como '*'.
    fn mul(self, new_vec: Vector2) -> f32 {
        self.x * new_vec.x + self.y * new_vec.y
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    //...

    #[test]
    fn adds_right_and_left_vectors() {
        let actual = Vector2::RIGHT() + Vector2::LEFT();
        assert_eq!(actual.x, 0f32);
    }

    #[test]
    fn adds_right_and_up() {
        let actual = Vector2::RIGHT() + Vector2::UP();
        assert_eq!(actual.x, 1f32);
        assert_eq!(actual.y, 1f32);
    }
}

```

```

#[test]
fn mult_one_by_3() {
    let actual = Vector2::ONE() * 3f32;
    assert_eq!(actual.x, 3f32);
    assert_eq!(actual.y, 3f32);
}

#[test]
fn dot_product() {
    let vec1 = Vector2 {x: 2f32, y: 1f32};
    let vec2 = Vector2 {x: 1f32, y: 2f32};
    let actual = vec1 * vec2;
    let expected = 4f32;
    assert_eq!(actual, expected);
}

//...
}

```

Podemos ver que a trait `ops::Add (+)` para a soma de dois vetores é constituída de um tipo de saída `Output Vector2` . Portanto, quando fizermos `Vector2 + Vector2` , teremos um novo `Vector2` .

A segunda implementação remete a multiplicação de um `Vector2` por um tipo `f32` , que retorna outro `f32` . É interessante notar que, neste caso, temos a declaração da implementação `impl ops::Mul<f32> for Vector2` , que implementa o operador `Mul` para `Vector2` quando é multiplicado por um tipo `f32` .

Já na última implementação, temos um outro caso interessante. O produto escalar de dois vetores, no qual `Vector2 Mul Vector2 = f32` , foi definido o tipo de retorno como um `f32` , e a declaração da implementação de `Mul` ficou `impl ops::Mul<Vector2> for Vector2` , que significa a

implementação do operador `Mul` para dois `Vector2`. No fim da listagem, podemos observar alguns testes que foram usados para desenvolver as sobrecargas.

Agora podemos avançar para os próximos passos da programação funcional. Para isso, gostaria de mostrar dois `shortcuts` da linguagem que acredito serem muito úteis.

Para saber mais

- Informações sobre tipos – <https://doc.rust-lang.org/book/second-edition/ch19-04-advanced-types.html>

IF LET E WHILE LET

A primeira vez que vi o `if let` e o `while let` foi lendo o livro *The Joy of Clojure* e, na época, achei uma solução brilhante. Porém, descobri que o Swift também possui uma feature parecida. A inclusão deste tópico como programação funcional se deu pelo que percebi de suas aplicações em Clojure.

Partindo para um pouco de detalhes, `if-let` e `while-let` são macros em Clojure, e são muito úteis para caso você sinta necessidade de atribuir o valor de uma expressão a uma variável. Isto evita a necessidade de `if/while` encadeados com `lets`, como mostra o exemplo a seguir:

Em Rust, o `if let` e o `while let` escritos separados. Mas em Clojure, por se tratar de uma macro, utilizamos o hífen (`if-let`).

```
(if :true
  (let [resultado :true] (println resultado)))
;; => :true
```

Com a macro `if-let`, teríamos:

```
(if-let [resultado :true] (println resultado))
```

```
;; => :true
```

Em Rust, existem soluções assim também: o `if let` e o `while let`.

7.1 IF LET

A sintaxe do `if let` permite combinar o `if` com o `let` de uma forma menos verbosa, especialmente se o objetivo é fazer um *pattern matching* (correspondência de padrões). Por exemplo, veja a listagem a seguir:

```
let valor = Some(0u8);
match valor {
    Some(8) => println!("imprime 8"),
    _ => (), // não faz nada.
}
```

O objetivo dessa listagem é imprimir `8` se `valor == Some(8)`; caso contrário, não fazer nada. Portanto, precisamos do `_ => ()` para que todos os casos do `match` sejam satisfeitos, o que adiciona um boilerplate desnecessário. Assim, poderíamos resolver o seguinte código de forma mais simples com um simples `if let`:

```
if let Some(8) = valor {
    println!("imprime 8");
}
```

Podemos ver que o `if let` torna o *pattern matching* bem menos verboso, pois recebe um padrão e uma expressão, separados por um `=`, e realiza a condição caso seja verdadeira. Alguns dos benefícios são menos digitação, menos indentações, mais consistência e menos boilerplate – um *syntax sugar* do `match` para casos específicos.

Por outro lado, perdemos a verificação exaustiva que o `match` realiza. Portanto, é preciso ter cuidado na hora de escolher entre `if let` ou `match`, tendo consciência de que existem ganhos e perdas.

Pattern matching (ou correspondência de padrões) é o ato de verificar se uma sequência de valores possui um determinado conjunto de padrões. Já *syntax sugar* (ou açúcar sintático) é um padrão de design de código que tem como objetivo torná-lo mais simples e fácil de ler.

7.2 IF LET ELSE

Felizmente, existem algumas soluções para validar casos não testados pelo `if`, o `if let else`. Imagine que temos um contador e que queremos que ele reaja de forma diferente em casos específicos. Por exemplo, queremos verificar se o valor é do tipo `Some(x)` para chamar `foo(x)`; se não, chamamos `bar()`.

```
if let Some(x) = contador {
    foo(x);
} else {
    bar();
}
```

Assim, entende-se que o `if let` desestrutura o contador em `Some(x)`: se for verdadeiro, avalia o bloco `{foo(x)}`; caso contrário, retorna o bloco `{bar()}`. Além disso, é possível adicionar um `else if` no meio do caminho como uma opção de verificação.

7.3 WHILE LET

De forma similar, o `while let` pode ser usado para fazer um loop condicional enquanto um padrão seja satisfeito. Assim, teremos:

```
let mut v_primos = vec![2, 3, 5, 7, 11];
loop {
    match v_primos.pop() {
        Some(x) => println!("Este número é primo= {}", x),
        None => break,
    }
}
```

Isso se transformaria em algo como:

```
let mut v_primos = vec![2, 3, 5, 7, 11];
while let Some(x) = v_primos.pop() {
    println!("Este número é primo= {}", x);
}
```

Agora que entendemos um pouco mais sobre as simplificações permitidas no desenvolvimento de *patterns* em Rust, podemos terminar o tópico de programação funcional com *iterators*, *adapters* e *consumers*.

ITERATORS, ADAPTERS E CONSUMERS

Programação funcional, como já foi "não definida" anteriormente, possui uma série de características. Tendo como referência o Clojure, uma linguagem de programação funcional impura – na qual todas as sequências são avaliadas e os dados são imutáveis por padrão –, há facilidade de se trabalhar com *laziness*, de implementar funções puras e de trabalhar com funções de ordem superior.

As linguagens de programação que dizem aplicar conceitos funcionais trazem, na maior parte, simplificações terríveis do que seria o paradigma funcional. Programação funcional não são maps, filters e reduces; não é tipagem nem é a presença de lambdas, mas são conceitos presentes em muitas das linguagens funcionais.

Nesse sentido, parece que Rust toma um caminho semelhante ao do Clojure, já que, ao usarmos esses conceitos, conseguimos agregar mais valor a suas funções. A seguir, veremos os tópicos importantes ao se discutir sobre programação funcional em Rust, especialmente por afetar os conceitos de iterators, adapters e consumers.

- **Imutabilidade:** como mencionado no capítulo *O que é programação funcional?*, se nada é imutável, não temos como garantir que não ocorram efeitos colaterais. Assim, se valores não são puros, nada poderá ser. Além disso, a linguagem também não permite criar variáveis mutáveis acidentalmente e, para consumi-las, é preciso explicitar. Sendo assim, Rust toma uma abordagem semelhante em relação aos efeitos colaterais.
- **Self em traits:** o uso de `self`, implícito ou explícito, é um grande efeito colateral, e Rust aplica esse conceito em diversos lugares, especialmente nas implementações das traits. Portanto, em uma situação ruim, o melhor é que seja explícito.
- **Mocking:** Rust não aplica conceitos de mocking. Ao observamos os mocks sob os olhos de efeitos colaterais, eles são uma grande bandeira vermelha de código impuro, portanto, aos olhos da programação funcional, são um sinal de que algo está errado.

Uma vez um colega de trabalho *javeiro* me perguntou como se faz mocks em Clojure, mas a resposta é que não aplicamos mocks em Clojure, pois a necessidade deles é um sinal de que precisamos refatorar o código – e essa mesma percepção existe no Rust.

Portanto, neste capítulo, vamos partir do pressuposto de que precisamos explorar em programação funcional a facilidade com que podemos operar cálculos sobre coleções, independente de seu tipo. Uma lista de variáveis é conhecida como *vector* em Rust, abreviada para `vec`. E ao aplicar técnicas de programação

funcional, é possível manipular vetores (ou outras coleções) com pouquíssimo código, evitando os tradicionais loops (`for`). Isso pode ser resolvido com a aplicação de maps, filters, zips etc.

8.1 ITERATORS

Em Rust, o tipo `Iterator` é responsável pela maior parte do "trabalho pesado" funcional. Aplicando as funções `iter()` e `into_iter()`, qualquer vetor pode ser transformado em um `Iterator`. A grande diferença entre as duas é como elas passam os valores para ele:

- `iter()` – Passa os valores por referência, evitando a criação, muitas vezes desnecessária, de cópia.
- `into_iter()` – Passa os valores por cópia, copiando elemento a elemento.

É importante lembrar de que isso é feito de forma lazy, portanto, os dados somente são consumidos caso seja necessário.

COMO FUNCIONA UM ITERATOR

Uma boa aproximação é pensar nas funções usadas pelos Iterators como "loops (for) especializados" que agem de forma recursiva no vetor ao aplicar uma série de funções `next()` com closures como argumento. Assim, cada passagem pelas funções retorna outro Iterator com todos os resultados. Independente da quantidade de funções que forem chamadas, ele continuará sendo um Iterator até que a função `collect()` seja chamada.

Como um Rust utiliza o modelo lazy de programação funcional, somente computará o que for preciso, igual ao Haskell. Ou seja, somente fará o mínimo de cálculos possíveis para atingir o resultado esperado.

Quando aplicar iter()?

A maior parte dos exemplos que posso pensar utiliza `iter()`. Por exemplo, quando chamamos `iter()` em vetores ou slices, isso cria um tipo `Iter<'a, T>`, que implementa a trait `Iterator` responsável por permitir que apliquemos funções como `map()`.

Vale lembrar de que `Iter<'a, T>` tem somente uma referência ao tipo `T`, que somente vai permitir iterar sobre o tipo `T` no qual foi aplicado o `Iter`. Um exemplo seria saber a quantidade de letras que meu nome tem:

```
fn main() {
```

```

let meu_nome = vec!["Julia", "Naomi", "Boeira"];

let num_letras = meu_nome
    .iter()
    .map(|nome: &&str| nome.len())
    .fold(0, |soma, len| soma + len );

assert_eq!(num_letras, 16);
}

```

Neste exemplo, a combinação de `map` com `fold` nos permite contar a quantidade de bytes (strings Rust são UTF-8) de todas as string no vetor. Sabemos que a função `len` pode utilizar uma referência imutável, por isso é correto usar o `iter()` em vez de `iter_mut` ou `into_iter`.

Além disso, isso nos permite mover o vetor `meu_nome` em outro momento, caso seja necessário. A closure aplicada no `map()` não exige que tipemos, mas, por motivos didáticos, optamos por tipar para facilitar a compreensão do que está sendo passado.

Algo curioso neste aspecto é que o tipo de `nome` é `&&str`, e não `&str`. Isso ocorre porque o tipo de `meu_nome` é `&str`, mas, após aplicarmos `iter()`, recebemos a referência `nome` `&&str` do `Iterator` de `meu_nome` `&str`. É interessante pensar em tipar closure quando queremos aplicar uma desestruturação. Caso seja necessário mutar dados durante um `map`, é possível utilizar o `iter_mut()`.

Quando aplicar `into_iter()`?

O `into_iter()` deve ser usado quando queremos mover (*move*) o valor em vez de pegá-lo emprestado. O `into_iter()` cria o tipo `IntoIter<T>`, que possui `ownership` dos valores

originais.

Como anteriormente, é o `IntoIter<T>` que implementa o `trait Iterator`. A palavra *into* geralmente representa que a variável `T` está sendo movida. A aplicação mais clara de `into_iter()` é quando existe uma função transformando valores.

```
fn nome(v: Vec<(String, usize)>) -> Vec<String> {
    v.into_iter()
        .map(|(name, _score)| name)
        .collect()
}

fn main() {
    let vec = vec!( "Julia".to_string(), 10);
    let nome = nome(vec);

    assert_eq!(nome, ["Julia"]);
}
```

A ideia de usar `into_iter()` aqui é o fato de que a tupla `(String, usize)` é transformada em string.

8.2 MAPS, FILTERS, FOLDS E TUDO MAIS

O tipo `Iterator` contém algumas funções centrais, como `map()`, `filter()`, `collect()`, `fold()` e muitas outras que falaremos adiante. Creio que estas 4 sejam as mais representativas e uma ótima introdução:

- `Map` aplica uma closure a cada elemento de um vetor, e retorna este resultado para a próxima função a ser aplicada.
- `Filters` funcionam de forma parecida, porém, eles retornam somente os elementos que satisfazem algum critério específico.

- Fold aplica uma closure cujo propósito é acumular, de alguma maneira, todos os elementos em uma única linha.
- Ao final de todas as chamadas de funções, a função `collect()` é utilizada para retornar um novo vetor de valores.

Agora podemos definir um vetor, como: `let vector = vec!(1, 3, 4, 5, 3);` . Queremos mapear todos estes números para serem pares, utilizando `map` :

```
let vetor_pares = vector.iter().map(|&x| x * 2).collect::

```

Ou talvez queremos filtrar todos aqueles que não correspondem à condição de valores pares, usando `filter` :

```
let valores_pares = vector.iter().filter(|&x| x % 2 == 0).collect::

```

Ou simplesmente queremos contar quantos pares temos no vetor? Logo, usaríamos `count` .

```
let contagem = vector.into_iter().filter(|x| x % 2 == 0).count();
```

É possível também encontrar a soma de todos os itens do vetor, com `fold` :

```
let soma = vector.iter().fold(0, (|acc, valor| acc + valor));
```

Podemos aplicar um `zip` :

```
let index_vec = 0..contagem;
let indexed_vector = vector.iter().zip(index_vec).collect::

```

Ou descobrir o valor máximo de um vetor com `max()` :

```
let max = vector.iter().max().unwrap();
```

Exemplos mais elaborados

Um bom primeiro exemplo seria criar um vetor com os 5 primeiros números ímpares ao quadrado, a partir do *range* de 1 ao infinito.

```
fn main() {
    let vector = (1..)
        .filter(|x| x % 2 != 0)
        .take(5)
        .map(|x| x * x)
        .collect::<Vec<usize>>();
    assert!(vec![1, 9, 25, 49, 81] == vector);
}
```

É interessante percebermos que, após a sequência `(1..)`, não temos um `iter` ou `into_iter`. Isso se deve ao fato de que ranges já são operadas de forma lazy, por padrão. Nosso `filter` procura por elementos não pares, e o `take` retira os 5 primeiros. Nelas a função `map` retorna o quadrado de cada um, e a função `collect` nos retorna um `vector`.

Agora imagine que queremos manipular meu nome, para encontrar minha letra preferida, o `o`:

```
fn main() {
    let nome = "Julia Naomi Boeira";
    let nomes: Vec<&str> = nome
        .split_whitespace()
        .collect();
    let nomes_com_o: Vec<&str> = nomes
        .into_iter()
        .filter(|nome| nome.contains("o"))
        .collect();
    assert!(vec!["Naomi", "Boeira"], nomes_com_o);
}
```

and_then

Para o tipo `Result`, Rust possui um adapter muito interessante, o `and_then()`. Trata-se de uma função chamada `se`, e somente `se`, o `Result` for do tipo `Ok()`. Agora imagine que queremos escrever uma função que eleve um número ao quadrado:

```
let resultado: Result<usize, &'static str> = Ok(2);
let valor = resultado.and_then(|n: usize| Ok(n * n));
assert_eq!(Ok(4), valor);
```

Caso a primeira linha recebesse um `Err`, a closure `|n: usize| Ok(n * n)` não seria chamada, como demonstramos a seguir:

```
let resultado: Result<usize, &'static str> = Err("Erro");
let valor = resultado.and_then(|n: usize| Ok(n * n));
assert_eq!(Err("Erro"), valor);
```

Agora pensei que queremos aninhar `and_thens`. Podemos ver um bom exemplo disso ao tentarmos realizar uma inversão de um número `n`, de modo que ele fique `1/n`. Temos de garantir que o valor `n` não é zero, como o exemplo a seguir nos mostra:

```
fn main() {
    let res: Result<usize, &'static str> = Ok(0);

    let valor = res
        .and_then(|n: usize| {
            if n == 0 {
                Err("Numero nao pode ser zero")
            } else {
                Ok(n)
            }
        })
        .and_then(|n: usize| Ok(1f32 / n as f32)); // <- closure
    // não é chamado

    assert_eq!(Err("Numero nao pode ser zero"), valor);
}
```

Antes de partir para o último tópico, vale conhecermos o

`adapter` `or_else` , chamado somente quando o `Err()` ocorre. Ele é menos frequente, mas muito bom de se ter em mente, pois é usado da mesma forma que o `and_then` .

No código a seguir, definimos duas funções que vão nos retornar dois valores `Option` , e faremos 3 testes com `assert_eq!` . No primeiro, nosso `Option` possui um tipo definido `Some(Barbaros)` e temos como retorno `Some(Barbaros)` . Agora, se nosso valor é um `None` , podemos retornar um tipo `Some(romanos)` ou outro `None` .

```
fn nao_identificado() -> Option<&'static str> { None }
fn romanos() -> Option<&'static str> { Some("romanos") }

assert_eq!(Some("Barbaros").or_else(romanos), Some("Barbaros"));
assert_eq!(None.or_else(romanos), Some("romanos"));
assert_eq!(None.or_else(nao_identificado), None);
```

8.3 CONSUMER

De certa forma, Consumers já foram explicados nos exemplos anteriores quando usamos a função `collect` e a função `fold` . Sua principal função é retornar uma sequência ou um tipo, de acordo com a iteração `lazy` gerada pelos iterators e adapters. Portanto, para um iterator retornar um valor, é necessário utilizar funções como essas citadas.

Um bom exemplo em relação ao `fold` pode ser feito com a demonstração a seguir, que nos apresenta uma comparação da soma dentro de um `for` , pelo `soma_for` , e a soma do `fold` , com o `soma_fold` . Nosso `fold` recebe dois parâmetros: o valor inicial, que vamos acumular (no nosso caso, zero), e uma closure, que tem como primeiro parâmetro o valor acumulado.

```

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

let mut soma_for = 0;

for i in &numeros {
    soma_for += i;
}

let soma_fold = numero.iter().fold(0, |acumulador, &x| acumulador
+ x);

assert_eq!(soma_for, soma_fold);

```

O `collect` transforma o iterator e seus adapters em uma coleção que é retornada. Temos uma lista de caracteres aparentemente desconexa sobre a qual vamos iterar.

O primeiro `map` transforma todos os nossos caracteres em `u8`. Já o segundo adiciona `1` a esses valores e devolve-os ao formato `char`. Para finalizar, aplicamos um `collect`, em que não precisamos explicitar um tipo, pois já foi explicitado em `String`, que por um acaso é uma coleção.

```

let chars = ['g', 'd', 'k', 'k', 'n'];

let hello: String = chars.iter()
    .map(|&x| x as u8)
    .map(|x| (x + 1) as char)
    .collect();

assert_eq!("hello", hello);

```

Agora conseguimos entender como utilizar funções em diversos aspectos, aplicá-las a tipos diferentes e especialmente iterar sobre sequências de coleções com iterators, adapters e consumers. Assim, conseguimos muitas formas de simplificar a forma como iteramos sobre estruturas de dados, o que facilitará muito a forma como lidamos com requests e reponses nas

diversas libs disponíveis no Rust.

Para saber mais

- Especificações da documentação do Iterator – <https://doc.rust-lang.org/stable/std/iter/trait.Iterator.html>

Programação concorrente

- O que é programação concorrente?
- Threads — A base da programação concorrente
- Estados compartilhados
- Comunicação entre threads

O QUE É PROGRAMAÇÃO CONCORRENTE?

A seguir, será feito um *approach* baseado em Clojure.

MAIS DOIS DISCLAIMERS

- O slogan do Rust é *Fearless Concurrency*, ou seja, concorrência sem medo.
- Por causa deste slogan, optei por comparar com a linguagem que conheço, que resolve da melhor forma possíveis problemas de concorrência, o Clojure. Claro que Go, C#, Python etc. têm suas soluções, mas nenhuma delas resolve da forma como eu creio ser a melhor.

9.1 DEFINIÇÃO DE CONCORRÊNCIA

Existem 3 termos que muitas vezes são associados aos domínios da programação concorrente, mas que representam coisas diferentes: concorrência, paralelismo e programação distribuída. A concorrência baseia-se em realizar diferentes tarefas

em threads diferentes. O paralelismo baseia-se em quebrar uma tarefa em diferentes threads. Já a programação distribuída considera o domínio de redes de computadores atuando como uma única máquina.

O foco neste momento é a programação concorrente, que representa um sistema desenhado para realizar processos lógicos de forma independente. A ideia por trás é que a concorrência dispara tarefas ao mesmo tempo, dividindo recursos em comum, mas não necessariamente realizando tarefas relacionadas.

9.2 POR QUE CLOJURE É UMA BOA REFERÊNCIA DE COMPARAÇÃO?

Como ponto central do Clojure está o gerenciamento de estados, que por sua vez leva a uma excelente forma de trabalhar com concorrência. Considerando que todos os valores são imutáveis, há uma grande facilidade de prontamente compartilhar esses valores entre as mais diferentes threads. Além disso, como veremos mais adiante, Clojure possui tipos específicos para cada situação, considerando a necessidade de sincronicidade, assincronicidade, independência e coordenação.

Em outros casos, como o Java, que opera em um modelo de concorrência de estados compartilhados, é necessário um grande malabarismo para gerenciar os *locks* que protegem os dados compartilhados. No Java, por mais que você consiga se localizar com todos os locks, ele raramente escala a qualquer ponto saudável.

Já a STM (*Software Transactional Memory*) do Clojure permite

uma forma não bloqueante de coordenar updates em valores de células mutáveis, já que mutações são permitidas somente dentro de uma mesma transação.

Como o Clojure faz?

Atualmente, há quatro tipos de referência para auxiliar a programação concorrente: *refs*, *agents*, *atoms* e *vars*. A tabela a seguir lista suas principais propriedades, mas essencialmente todas — exceto *vars* — são referências compartilhadas e permitem a visualização pela thread.

Veja a tabela de tipos de referência por propriedades:

	Ref	Agent	Atom	Var
Coordenada	x			
Assíncrona		x		
Repetível	x		x	
Thread-Local				x

Um dos pontos interessantes de se observar ao utilizar a STM é a falta de locks (ou travas), já que não há chances de deadlocks. O mais interessante é que, quando uma identidade é requisitada, a transação recebe uma "foto", uma cópia do atual estado do instante das propriedades. Porém, não é necessariamente a foto mais recente, mas sim a foto coerente com o estado da thread. Ela permite que não existam conflitos entre transações, o que evita deadlocks.

9.3 E O RUST? COMO FICA?

A segurança de memória do Rust também se aplica às questões de concorrência, já que até mesmo programas concorrentes devem ser seguros, impedindo a existência de data races. O sistema de tipos do Rust consegue garantir muita coisa no momento de compilar.

Além disso, Rust permite que você implemente suas próprias soluções de concorrência. Um desafio interessante seria implementar alguma que aplique STM como em Clojure (o port de Clojure para CLR tem tido dificuldades para manter a mesma concepção da STM para a JVM).

Concorrência segura sempre foi uma prioridade para a equipe do Rust, mas o mais incrível é que essa linguagem permite uma grande variedade de abstrações de paralelismo. Essa variedade tem como objetivo garantir a ausência de data races, sem empregar coletores de lixo, algo que nenhuma outra linguagem conseguiu fazer.

THREADS

A unidade de computação em diferentes núcleos em Rust é chamada de thread, assim como em muitas outras linguagens, que é um tipo definido pela biblioteca padrão `std::thread`. Cada thread possui sua própria stack e um estado local.

O Rust permite a criação de muitas threads, e cada uma pode ser mãe de muitas outras threads filhas. Veja o que pode ser feito com dados nas threads:

- Dados podem ser compartilhados por diferentes threads, quando queremos um estado compartilhado mutável.
- Dados podem ser enviados por meio de threads, com "comunicação".

9.4 RUST: CONCORRÊNCIA SEM MEDO

Como já mencionado anteriormente, o Rust foi desenvolvido para solucionar dois problemas em especial:

- Como programar sistemas de forma segura?
- Como fazer a concorrência não virar uma dor de cabeça?

Inicialmente, esses problemas foram tidos como ortogonais, mas a equipe de desenvolvimento percebeu que a solução era a mesma, e que o problema era que: *bugs de memória e bugs de concorrência geralmente são causados por acessar informações quando não deviam*. Assim, a solução foi o sistema de *ownership*,

algo que o compilador realiza, e não algo que os programadores devem ter em mente.

Para a segurança de memória, isso significava que os coletores de lixo eram desnecessários e que o programa não acessaria uma memória não permitida (*segfaults*). Para a concorrência, isso significava que não existiram as armadilhas triviais, independente do paradigma que fosse aplicado — como mensagens, estados compartilhados, sem locks, funcional etc.

Algumas das soluções que apareceram foram:

- Channels (canais) transferem ownership por meio de mensagens enviadas, sendo possível enviar um ponteiro de uma thread a outra sem medo de dar a face quando o ponteiro for acessado. *Os canais procuram reforçar os conceitos de isolamento de threads.*
- Um lock sabe qual dado tem de proteger, e Rust garante que um dado somente será acessado quando o lock for mantido. Assim, estados nunca são compartilhados acidentalmente. A filosofia por trás das locks é: *segure (lock) dados e não código.*
- Cada tipo de dado sabe se pode ser enviado ou acessado entre múltiplas threads. Não há data races, mesmo para estruturas de dados que não necessitam locks. *Segurança em threads é lei, e não apenas documentação.*
- Você pode até compartilhar stack frames entre threads, que o Rust assegurará estaticamente que os frames permaneçam ativos enquanto outras threads estão usando-os. As formas de compartilhamento mais ousadas são

garantidas em Rust.

9.5 QUANDO UTILIZAR CONCORRÊNCIA?

A partir de agora, veremos de forma mais profunda a concorrência em Rust, iniciando por threads, depois compartilhamento de estados em threads e, por fim, channels. Porém, é interessante trazer alguns pontos que tornam a concorrência atrativa, como quando um jogo atualiza o estado de centenas de unidades ao mesmo tempo, enquanto toca uma música de fundo. Esse exemplo poderia muito bem quebrar os vários estados em diferentes threads.

Ou então, imagine um programa de inteligência artificial que divide seus cálculos entre os vários núcleos para ter uma melhor performance; ou um servidor web que lida com mais de um request ao mesmo tempo com o objetivo de maximizar suas saídas, assim como um servidor que precisa paralelizar funções independentes. Ou, por fim, imagine um caso clássico em que a interface do usuário continua funcionando perfeitamente enquanto o programa faz diversos requests ao mesmo tempo.

THREADS — A BASE DA PROGRAMAÇÃO CONCORRENTE

Já mencionei o básico sobre o que é uma thread no capítulo anterior, então agora podemos partir direto para o que importa: *criar uma thread*. Isso é feito pela função `thread::spawn`, que permite criar uma thread filha capaz de viver mais que a thread mãe.

A principal thread mãe é a `main`, que foi criada declarando `fn main() {...}`, geralmente com a opção `--bin` no Cargo. O exemplo a seguir mostra como isso funciona:

```
use std::thread;

fn main() {
    thread::spawn(move || {
        println!("Ola vindo da thread filha");
    });
    thread::sleep_ms(100);
    println!("Ola vindo da thread mãe");
}
```

Podemos ver que a função `spawn` recebe uma closure como argumento, e deve executar a thread filha de forma independente

da thread mãe (neste caso, `main()`). É importante salientar que a closure possui o `move` , pois a thread necessita ter propriedade sobre seus elementos.

Note que existe uma outra função no exemplo, a `sleep_ms` . Ela está presente, uma vez que a thread `main` é a pior mãe possível: quando ela encerra, suas filhas também são encerradas.

Para solucionar este problema, adicionamos o `sleep_ms` , que garante tempo para a thread filha encerrar. Como já mencionei anteriormente, thread filhas podem, em geral, viver além de seus pais, tornando desnecessária a função `sleep_ms` , porém a `main` foge a essa regra.

FUNÇÃO SPAWN

```
fn spawn<F, T>(f: F) -> JoinHandler<T>
  where F: FnOnce() -> T
        F: Send + 'static,
        T: Send + 'static
```

- `Send` — É um *marker trait*, o que significa que não implementa nenhum método, somente sinaliza que o valor é seguro de ser passado por threads. Além disso, é implementado automaticamente para estruturas que o compilador percebe não ter problemas para enviar entre threads.
- `JoinHandler` — Pode ser usado para sincronizar a execução de diferentes threads. A `Join()` exige que as threads terminem.

Utilizar o `sleep_ms` resolveu nosso problema, mas tornou

nosso código grosseiro e feio. Para isso, existe uma solução mais elegante, que consiste em usar o *Join Handle*, que é a variável gerada pelo `spawn`.

Portanto, chamar o método `join()` na variável `handle` gerada pela thread é uma forma de garantir sua execução, pois bloqueia a thread mãe até que a filha se encerre. O valor de retorno será um `Result`, e neste caso o `OK` será do tipo `()` ou `void`, já que nossa thread não retorna um valor. Vale lembrar também de que `Result` é um `enum` com as opções `OK` e `Err`.

```
use std::thread;

fn main() {
    let handle = thread::spawn(move || {
        println!("Ola vindo da thread filha");
    });

    println!("Ola vindo da thread mãe");
    let output = handle.join().unwrap();
    println!("A thread filha retornou {:?}", output);
}
```

Outra maneira de resolver isso seria utilizar o `join()` logo depois de lançar a thread. Entretanto, isso tornaria a execução de nosso código síncrona, o que só teria sentido se não fôssemos lançar nenhuma outra thread. Ficaria assim:

```
thread::spawn(move || {
    println!("Ola vindo da thread filha");
}).join();
```

10.1 LANÇANDO MUITAS THREADS

Como já mencionei anteriormente, cada thread tem seu próprio stack e estado local e, por padrão, nenhuma informação é

compartilhada entre elas se não forem imutáveis. O processo de iniciar threads é bastante simples. O exemplo a seguir cria 10_000 threads:

```
use std::thread;
static DEZ_MIL: i32 = 10_000;

fn main() {
    for i in 0..DEZ_MIL {
        let _ = thread::spawn(move || {
            println!("Chamando a thread de número {:?}", i);
        });
    }
}
```

A execução dessas threads ocorre de forma independente, portanto, os resultados podem ocorrer de forma desordenada:

```
Chamando a thread de número 2
Chamando a thread de número 1
Chamando a thread de número 4
Chamando a thread de número 6
Chamando a thread de número 3
Chamando a thread de número 5
...
```

QUANTAS THREADS DEVEMOS LANÇAR?

Existe uma regra que o número de threads deve ser o mesmo número de núcleos da CPU, ou pelo menos muito próximo disso. O Rust possui uma crate que soluciona esse problema, a `num_cpus`, que permite determinar os números de núcleos da CPU. Para utilizá-la, adicione as seguintes linhas em seu `Cargo.toml`:

```
[dependencies]
num_cpus = "*"

```

Assim, podemos reescrever o código anterior, considerando o número de núcleos:

```
extern crate num_cpus;
use std::thread;

fn main() {
    let ncpus = num_cpus::get();
    for i in 0..ncpus {
        let _ = thread::spawn(move || {
            println!("Chamando a thread de número {:?}", i);
        });
    }
    println!("Total de núcleos {:?}", ncpus);
}
```

Outra dependência interessante de se utilizar é a *threadpool*. É só adicionar `threadpool = "*"` às dependências do `Cargo.toml`. Sua função é executar um número de tarefas sobre um conjunto pré-definido de threads paralelas.

Além disso, a *threadpool* é capaz de reabastecer a pool com

novas threads, caso ocorra algum `panic`. Então, teríamos algo assim:

```
extern crate num_cpus;
extern crate threadpool;

use std::thread;
use threadpool::ThreadPool;

fn main() {
    let ncpus = num_cpus::get();
    let pool = ThreadPool::new(ncpus);

    for i in 0..ncpus {
        pool.execute(move || {
            println!("Chamando a thread de número {:?}", i);
        })
    }
    thread::sleep_ms(100);
}
```

10.2 PANIC! AT THE THREAD

O que acontece quando uma thread lança um `panic`? Essencialmente, não acontece nada, já que threads são isoladas umas das outras. Somente a thread que lançou esse `panic` vai colapsar após liberar seus recursos, não afetando a thread mãe.

Entretanto, ela pode testar o valor de retorno com a função `is_err()`, como mostra o exemplo a seguir:

```
use std::thread;

fn main() {
    let resultado = thread::spawn(move || {
        panic!("Oh! Nao! Estou em panico");
    }).join();

    if resultado.is_err() {
        println!("Seu filho entrou em panico!");
    }
}
```

```
}  
}
```

O resultado será:

```
thread '<unnamed>' panicked at 'Oh! Nao! Estou em panico', src/ma  
in.rs:5  
note: Run with `RUST_BACKTRACE=1` for a backtrace.  
Seu filho entrou em panico!
```

10.3 THREADS SEGURAS

Programação multithreads é tradicionalmente muito complexa se você permitir que as diferentes threads compartilhem as mesmas variáveis mutáveis, isto é, a dita memória compartilhada. Quando duas ou mais threads modificam simultaneamente uma variável, ocorre a formação de dados corrompidos, as *data races*. Isso acontece devido à falta de previsibilidade do comportamento de diferentes threads.

Threads são tidas como seguras quando diferentes threads não causam corrompimento dos dados que acessam. E é exatamente isso o que o compilador do Rust impede que aconteça, aplicando os princípios de ownership.

Para exemplificar, imagine o caso em que Goku está sendo atacado por diferentes Cells Jr. Goku possui 1000 pontos de vida, e cada Cell Jr. pode lhe causar entre 50 a 200 pontos de dano. A listagem a seguir mostra o que esperaríamos de uma implementação desse problema:

```
use std::thread;  
  
fn main() {  
    let mut vida_goku = 1000;  
    for cell in 1..5 {
```

```
        thread::spawn(move || {
            vida_goku -= cell * 50;
        });
    }
    thread::sleep_ms(2000);
    println!("A vida de Goku apos os ataque: {}", vida_goku);
}
```

Mas neste caso vemos que a vida de Goku permanece 1000, mesmo após os ataques. Isso ocorre porque as threads agiram sobre as cópias dos dados de `vida_goku` (devido à aplicação de `move`), e não na variável em si. Isto é, o Rust não permite que as threads operem sobre uma mesma variável para prevenir dados corrompidos.

No próximo capítulo, vamos mostrar como o estado compartilhado mutável ocorre.

ESTADOS COMPARTILHADOS

Como podemos fazer o programa do capítulo anterior nos retornar o valor correto? O Rust possui um tipo específico para essas situações: os atômicos, `std::sync::atomic`.

Eles permitem trabalhar sobre os dados mutáveis de forma segura. Para modificar os dados, é necessário encapsulá-los em tipos primitivos de `Sync`, como os `Arc`, `Mutex`, `AtomicUSize` etc.

De forma bastante simples, aplicamos os princípios de `lock` (que o Clojure evita), assim, o acesso exclusivo ao recurso é concedido à thread que possui o lock sobre o recurso. No caso da linguagem Rust, chamamos isso de *mutex* (em inglês, *mutually exclusive*, e mutualmente exclusiva, em português).

Uma lock será concedida somente a uma thread por vez, impedindo que duas modifiquem o mesmo valor ao mesmo tempo, o que evita data races. Quando uma thread termina seu trabalho, a lock é concedida a outra thread. Esse sistema é reforçado pelo próprio compilador. A vida encapsulada pelo `Mutex<T>` ficaria semelhante a isso:

```
let vida_atual = Mutex::new(vida_goku);
```

E em vida_atual, aplicaríamos uma lock assim que a thread chamar o recurso:

```
for cell in 1..5 {
  thread::spawn(move || {
    let mut vida_goku = vida_atual.lock().unwrap();
    //Resto do código
  });
}
```

Dessa forma, a chamada de `lock()` retornará uma referência ao valor contido dentro do Mutex, e bloqueará qualquer outra chamada até que a referência saia do escopo. No atual estado, teremos um erro (`capture of moved value: 'vida_atual'`), que significa que uma variável não pode ser movida para outras threads várias vezes.

Esse problema é bastante simples: *todas as threads precisam de referências aos mesmos recursos* (nossa variável `vida_goku`). Para resolver isso, podemos utilizar o conceito de ponteiro `Rc` , mas de forma mais segura, o `Arc<T>` , cujo significado é um ponteiro `Rc` atômico, ou contador de referências atômico.

A função do `Arc` é ser seguro por meio de todas as threads, conforme o exemplo a seguir:

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

fn main() {
  let mut vida_goku = 1000;
  let recursos = Arc::new(Mutex::new(vida_goku));

  for cell in 1..5 {
    let mutex = recursos.clone();
    thread::spawn(move || {
```

```

        let mut vida_goku = mutex.lock().unwrap();
        *vida_goku -= cell * 50;
    }).join().unwrap();
}
thread::sleep_ms(2000);
vida_goku = *recursos.lock().unwrap();
println!("A vida de Goku apos os ataque: {}", vida_goku);
}

```

Outra opção seria usar `expect` em vez de `unwrap`, pois isso permite entender no console qual foi e onde ocorreu o erro que obtivemos. Veja o exemplo a seguir:

```

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

fn main() {
    let mut vida_goku = 1000;
    let recursos = Arc::new(Mutex::new(vida_goku));

    for cell in 1..5 {
        let mutex = recursos.clone();
        thread::spawn(move || {
            let mut vida_goku = mutex.lock().expect("Goku se defendeu através do lock");
            *vida_goku -= cell * 50;
        }).join().expect("União dos golpes falhou");
    }
    thread::sleep_ms(2000);
    vida_goku = *recursos.lock().unwrap();
    println!("A vida de Goku apos os ataque: {}", vida_goku);
}

```

RC (REFERENCE COUNTED SMART POINTER)

A contagem de referência (significado de RC) acompanha o número de referências a um valor para saber se ele ainda está em escopo. Se não houver referências a um valor, sabemos que podemos limpá-lo sem que elas sejam inválidas.

Na maioria dos casos, a ownership é bastante trivial: você sabe exatamente qual recurso possui determinado valor, mas isso não pode ser aplicado sempre. Às vezes, você pode realmente precisar de vários owners (proprietários). Para isso, o Rust tem um tipo chamado `Rc<T>`, e seu nome é uma abreviatura para a contagem de referência.

Usamos `Rc <T>` quando queremos alocar dados na heap para que várias partes do programa possam ler, sem que precisemos determinar em tempo de compilação quais partes do nosso programa utilizarão por último. Observe que `Rc <T>` é apenas para o uso em cenários com uma única thread.

No cenário atual, cada thread atua em uma cópia do ponteiro obtido pelo método `clone()`, e a instância `Arc` manterá controle sobre todas as referências criadas na `vida_goku`. A chamada à `clone` vai incrementar a contagem das referências existentes na `vida_goku`.

A referência `mutex` sai de escopo assim que a thread termina, decrementando a contagem de referências, e `Arc` liberará os recursos associados assim que a contagem for zero. Além disso, a

resposta que a lock nos retorna é do tipo `Result<T, E>`, pois pode acontecer alguma falha, já que utilizamos a função `unwrap()` e assumimos que tudo estaria em ordem.

A função `unwrap` nos retorna o valor `T` caso a thread ocorra normalmente, e entrará em pânico (`panic!`) caso aconteça algum problema. Reunimos as threads chamando a dereferência de recursos (`*recursos`) para conseguir a `vida_goku` interna.

Esse mecanismo `Arc (mutex)` é recomendável para quando os recursos ocupam uma quantidade de memória significativa. Isso ocorre porque, com `Arc`, os recursos não serão mais copiados para cada thread, devido ao fato de ele atuar como referência para os dados compartilhados, então somente essa referência é compartilhada e copiada.

Uma informação importante é que `Arc<T>` implementa a trait `Sync`, exigida para que um tipo possa ser usado de forma concorrente. Qualquer tipo que contenha tipos que implementem `Sync` é automaticamente `Sync`. Exemplos disso seriam inteiros, floats, enum, structs e tuples - caso contenham tipos `Sync` em suas definições.

Mas ainda precisamos pensar em como nossas threads podem se comunicar, e vamos explorar isso no próximo capítulo.

COMUNICAÇÃO ENTRE THREADS

A comunicação entre threads pode ser realizada via *channels* (ou canais). Essa forma permite que as threads transmitam mensagens entre elas. Esses channels funcionam como dutos unidirecionais que conectam duas threads.

Os dados são processados de forma FIFO (o primeiro a entrar é o primeiro a sair), e os dados fluem entre as duas threads partindo do `Sender<T>` até o `Receiver<T>`. Nesse mecanismo, uma cópia do dado é feita para ser compartilhada com a thread recebedora, e por isso não se recomenda usar dados extensos.

Além disso, ultimamente se iniciou uma discussão com o objetivo de abolir os channels, por conta desse motivo recém-apresentado, mas creio que ele seja uma técnica útil para programas pequenos. A imagem a seguir exemplifica o seu funcionamento; no caso, criamos as threads a partir da `main`, mas elas poderiam ser desconectadas.

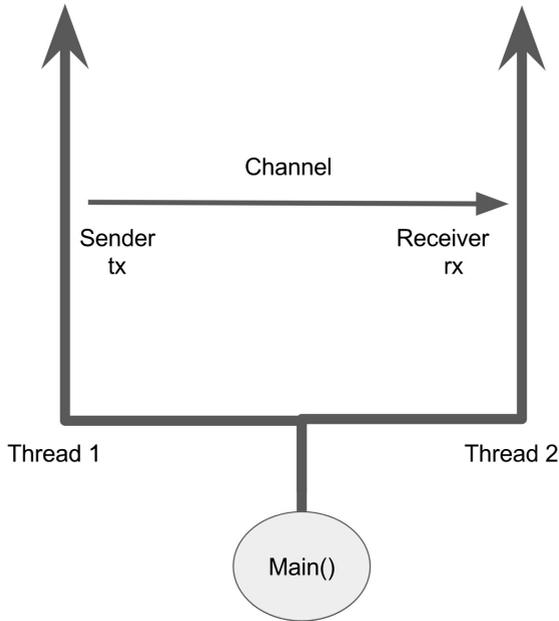


Figura 12.1: Desenho exemplificando channels

12.1 CRIANDO CHANNELS

Vamos ao que importa: como usamos channels? Para criar um channel, é preciso importar o submódulo `mpsc` (*multi-producer single-consumer communication primitives*, ou primitivos de comunicação produtor múltiplo consumidor único) da `std::sync`. Com isso, basta chamar o método `channel`:

```
use std::thread;
use std::sync::mpsc::channel;
use std::sync::mpsc::{Sender, Receiver};

fn main() {
    let (tx, rx): (Sender<f32>, Receiver<f32>) = channel();
}
```

Este código gera uma tupla de terminais `tx` (`t` de transmissão), que representa o `Sender` (quem envia), e o `rx` (`r` de receptor), que representa o `Receiver` (quem recebe). Aplicando `f32` ao genérico `T`, determinamos que o channel enviará informações do tipo `f32`, porém, as anotações de tipo não são compulsórias caso o compilador consiga inferir o que está sendo passado.

12.2 ENVIANDO E RECEBENDO DADOS

Para podermos enviar dados por channels, é preciso garantir que o que estamos enviando implemente a `trait send`, pois ela garante a transferência de `ownership` segura entre as threads. Dados que não implementam a `trait send` não conseguem ser transmitidos via channels — felizmente, não é o caso do `f32`.

No código seguinte, definimos um channel com as variáveis `tx : Sender<f32>` e `rx : Receiver<f32>`. Com isso, criamos uma thread com o `thread::spawn` e enviamos o valor de `pi` pelo `tx.send(3.1415f32).unwrap();`, para ser recebido por meio do receiver `let pi = rx.recv().unwrap();`.

```
use std::thread;
use std::sync::mpsc::channel;
use std::sync::mpsc::{Sender, Receiver};

fn main() {
    let (tx, rx): (Sender<f32>, Receiver<f32>) = channel();
    thread::spawn(move || {
        tx.send(3.1415f32).unwrap();
    });
    let pi = rx.recv().unwrap();
    println!("Pi is {:?}", pi);
}
```

Outra forma interessante de se escrever `tx` é definindo a ação. Neste caso, o `unwrap` é substituído por `.ok().expect("Pi não pode ser enviado")`.

```
tx.send(3.1415f32).ok().expect("Pi não pode ser enviado");
```

12.3 COMO FUNCIONA?

O `Send()` é executado pela thread filha, criando uma fila de mensagens (no nosso caso, `3.1415f32`) no channel, sem causar bloqueios. O `recv()` é feito pela thread mãe, que pega a mensagem do canal e bloqueia a thread se não há mensagens disponíveis.

Para fazer isso de forma não bloqueante, é possível usar o `try_recv()`. Ambas as operações `send()` e `recv()` retornam um tipo `Result`. Em caso de `Err`, o canal não funciona mais.

Um padrão interessante para utilizar é o apresentado a seguir, em que se retorna o tipo `Receiver<T>` a partir da função que cria os canais:

```
use std::sync::mpsc::channel;
use std::sync::mpsc::Receiver;

fn cria_canal(dado: i32) -> Receiver<i32> {
    let (tx, rx) = channel();
    tx.send(dado).ok().expect("Dado nao pode ser enviado");
    rx
}

fn main() {
    let rx = cria_canal(8000);
    if let Some(msg) = rx.recv().ok() {
        println!("O ki de Goku é mais de {:?}!!!", msg);
    }
}
```

O resultado disso exibe na tela 0 ki de Goku é mais de 8000!!! . Mas o mais importante neste exemplo é que a função `cria_canal` pode ser testada, algo que até então não tínhamos visto.

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn verifica_func_cria_canal() {
        let expected = Some(8000);
        let rx = cria_canal(8000);
        assert_eq!(expected, rx.recv().ok());
    }
}
```

Vale lembrar de que o método `ok()` nos retorna um `Option`, e não um tipo qualquer, por isso o `expected` deve ser `Some(x)`. Agora precisamos entender uma última etapa da comunicação entre threads.

12.4 COMUNICAÇÃO ASSÍNCRONA E SÍNCRONA

O tipo de comunicação que usamos até agora é assíncrona, ou seja, não bloqueia o código sendo executado. Porém, o Rust também possui channels síncronos, chamados de `sync_channel`, nos quais o `send` bloqueia caso seu *buffer* fique cheio, esperando até que a thread mãe comece a receber os dados. Segue o exemplo:

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

fn main() {
    let (tx, rx) = sync_channel(1);
```

```

tx.send("Kakaroto!!").unwrap();
thread::spawn(move || tx.send("Sou mais forte que você!").unw
rap());

thread::sleep_ms(2000);

if let Some(msg) = rx.recv().ok() {
    println!("Vegita: {:?}", msg);
}
if let Some(msg) = rx.recv().ok() {
    println!("Vegita: {:?}", msg);
}
}

```

O resultado será:

```

Vegita: "Kakaroto!!"
Vegita: "Sou mais forte que você!"

```

Nesse código, temos algumas questões específicas, que creio serem importantes salientar. O valor passado como argumento para `sync_channel` é chamado de *bound*, que representa o tamanho do buffer. Quando esse buffer fica cheio, `sends` futuros serão bloqueados esperando o buffer abrir.

Além disso, nosso `send` será recebido de forma sequencial, em que a `thread` terá um tempo de espera de 2000ms. A forma como implementamos o recebimento das informações pode não ser a ideal, mas mostra que conseguimos garantir que o recebimento de um buffer de tamanho `1` foi feito de forma síncrona.

Uma menção honrosa é a atual biblioteca assíncrona do Rust, a Tokio (<https://tokio.rs/docs/getting-started/futures/>). Entraremos em detalhes sobre isso no último capítulo.

Agora chegou o momento de começarmos a aplicar o conhecimento gerado. Felizmente, a maior parte das bibliotecas que vamos analisar já fez a abstração das threads para nós.

Para entendermos um pouco mais sobre como funciona concorrência em Rust, faremos uma análise das bibliotecas Iron, Nickel, Hyper e, por fim, a Tokio. Elas implementam fortemente as abstrações necessárias para garantir alta concorrência e facilidade de implementação.

Aplicando nossos conhecimentos

- Aplicando nossos conhecimentos
- Brincando com Nickel
- Hyper: servidores desenvolvidos no mais baixo nível
- Tokio e a assincronicidade
- Bibliografia

APLICANDO NOSSOS CONHECIMENTOS

Percorremos um longo caminho, mas agora finalmente teremos a oportunidade de explorar todo o potencial concorrente de Rust. Não é necessário se aprofundar muito na programação funcional, pois não se trata de uma linguagem puramente funcional, mas que apenas possui aspectos funcionais. Aqui abordaremos um problema que vai nos permitir explorar concorrência e um pouco dessa programação.

Nosso projeto se baseará em um banco de dados sobre *Dragon Ball*, que poderá operar de forma concorrente. Como o banco de dados não é o objetivo neste momento, vamos contornar isso com outras formas mais rudimentares. Nossos passos serão:

1. Subir uma aplicação Iron baseada em *Hello World*;
2. Configurar threads e timeouts para o nosso servidor;
3. Aplicar conceitos básicos de rotas;
4. Explicar mais a fundo o Iron;
5. Pensar em testes para o Iron;
6. Criar uma aplicação web básica.

Para isso, vamos utilizar o repositório `dbz-server`

(<https://github.com/naomijub/dbz-server>) no GitHub, com o seguinte comando:

```
cargo new dbz-server --bin
```

Para este projeto, usarei a crate Iron (<https://github.com/iron/iron>), um framework web mais maduro, mais próximo de sua versão estável neste momento, com uma biblioteca específica para testes e bibliotecas robustas de roteamento, encoding de URLs, parser de parâmetros, persistência e logging. Para podermos utilizar o Iron em nosso projeto, devemos adicionar sua crate no arquivo `cargo.toml` :

```
[dependencies.iron]  
version = "**"
```

13.1 IRON

O Iron é um framework de servidor baseado em middleware com desempenho rápido e flexível, que fornece uma base pequena (mas robusta) para criar aplicativos complexos e RESTful APIs. Nenhum middleware é empacotado com o Iron; em vez disso, tudo é *drag and drop* (arraste e solte), permitindo configurações ridiculamente modulares. Infelizmente, o Iron não possui uma boa documentação, então, este capítulo pode ser uma boa fonte de informações.

Olá Goku

Neste momento, precisamos estabelecer qual será o passo fundamental para iniciarmos nosso server. Para isso, proponho substituir o *Hello World* por *Olá Goku!* :

```
extern crate iron;

use iron::prelude::*;
use iron::status;

fn main() {
    Iron::new(|_: &mut Request| {
        Ok(Response::with((status::Ok, "Ola Goku!!!")))
    }).http("localhost:3000").unwrap();
}
```

Obtemos via postman (ferramenta para fazer requests) uma resposta `Ola Goku!!!` ao chamarmos `localhost:3000` :

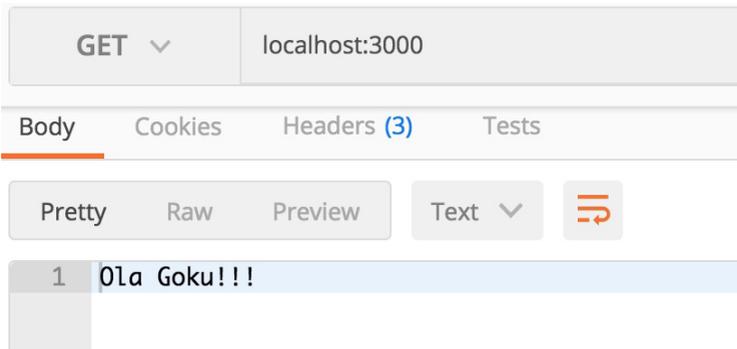


Figura 13.1: Olá Goku

Adicionando configurações

Falamos muito de concorrência, mas pouco mostramos isso até agora. Felizmente, o Iron faz isso por baixo dos panos para nós, com auxílio do Hyper (<https://github.com/hyperium/hyper>), que veremos com mais profundidade nos próximos capítulos. Mas como breve introdução, ele trata-se de uma biblioteca de implementação HTTP moderna e rápida, escrita em Rust.

Queremos inicialmente um servidor que possa utilizar todo o

poder computacional dos diversos cores da máquina. Para isso, podemos pressupor aproximadamente uma thread por core. Assim, para definirmos o número de threads do nosso servidor, basta adicionar a seguinte linha `iron.threads = NUMERO_THREADS; .`

Por meio de `.timeouts`, podemos definir casos de *timeout* para nosso serviço, já que não queremos que uma thread prenda o core por tempo indeterminado, ou queremos que o request seja retornado em, no máximo, um tempo específico:

```
extern crate iron;

use std::time::Duration; //Novo

use iron::prelude::*;
use iron::status;
use iron::Timeouts; //Novo

fn main() {
    let mut iron = Iron::new(|_: &mut Request| {
        Ok(Response::with((status::Ok, "Ola! Eu sou o Goku!!!")))
    });
    iron.threads = 8;
    iron.timeouts = Timeouts {
        keep_alive: Some(Duration::from_secs(10)),
        read: Some(Duration::from_secs(10)),
        write: Some(Duration::from_secs(10))
    };
    iron.http("localhost:3000").unwrap();
}
```

No código anterior, fizemos duas modificações. Com `iron.threads = 8;`, definimos que o nosso servidor atenderá requests em 8 threads diferentes e, pela struct `Timeouts`, definimos os timeouts do nosso servidor Iron, `iron.timeouts = Timeouts {...}`. A struct `Timeouts` conta com 3 campos cujos valores são `Option: keep_alive`, `read` e `write`.

Mas creio que esta implementação pode ter alguns problemas. Primeiro, estamos definindo uma variável mutável no centro de nosso servidor e alterando diversas vezes suas propriedades. Segundo, o código não me parece muito legível e limpo. O terceiro, e último ponto, é que a função `http` poderia ser chamada diretamente em `Iron`. Uma abordagem mais "funcional" e com um melhor estilo de código seria a seguinte:

```
fn main() {
    Iron {
        handler: |_: &mut Request| {Ok(Response::with((status::Ok
, "Ola! Eu sou o Goku!!!")))},
        threads: 8,
        timeouts: Timeouts {
            keep_alive: Some(Duration::from_secs(10)),
            read: Some(Duration::from_secs(10)),
            write: Some(Duration::from_secs(10))
        }}.http("localhost:3000").unwrap();
}
```

Ou, de forma mais limpa, poderíamos definir a função `handler eu_sou_goku` fora do escopo da função `main`:

```
fn main() {
    Iron {
        handler: eu_sou_goku,
        threads: 8,
        timeouts: Timeouts {
            keep_alive: Some(Duration::from_secs(10)),
            read: Some(Duration::from_secs(10)),
            write: Some(Duration::from_secs(10))
        }}.http("localhost:3000").unwrap();
}

fn eu_sou_goku(_: &mut Request) -> IronResult<Response> {
    Ok(Response::with((status::Ok, "Oi, eu sou o Goku!")))
}
```

Claramente, nossa estrutura de dados define o servidor, em que o `handler` recebe uma closure, e logo invocamos a função

`http/unwrap` . Outra possibilidade que temos é retornar o corpo do `request` . Por exemplo, no código a seguir, obteremos `Olá Goku` se enviarmos a mesma frase:

```
use std::io::Read;

...

fn main() {
    Iron {
        handler: |request: &mut Request| {
            let mut body = Vec::new();
            request
                .body
                .read_to_end(&mut body)
                .map_err(|e| IronError::new(e, (status::InternalServerError, "Error ao ler o request"))?);
            Ok(Response::with((status::Ok, body))),
        threads: 8,
        timeouts: Timeouts {
            keep_alive: Some(Duration::from_secs(10)),
            read: Some(Duration::from_secs(10)),
            write: Some(Duration::from_secs(10))
        }}.http("localhost:3000").unwrap();
}
```

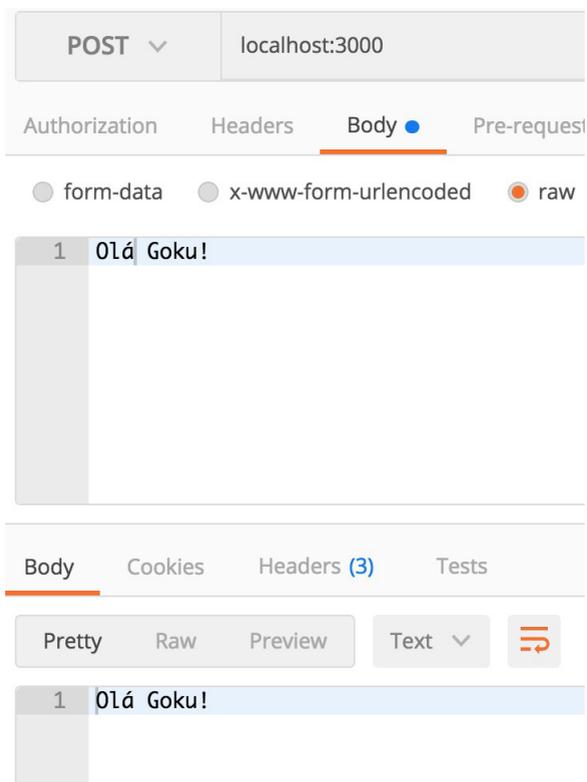


Figura 13.2: Resposta Olá Goku

Difícilmente existe um servidor que possui um único *endpoint*. Para podermos disponibilizar diferentes endpoints em nosso servidor, devemos aplicar estratégias de roteamento de chamadas, assim, vamos apresentar um pouco de routing com Iron.

Para termos acesso à biblioteca de *routing*, é preciso adicionar uma nova crate, a *router*, ao `cargo.toml` :

```
[dependencies]
iron = "*"
router = "*"
```

No nosso `main.rs`, devemos adicionar as seguintes informações:

```
extern crate router;
...
use router::Router;
```

Aplicando as duas funções que apresentamos anteriormente, em um GET e um POST, obtemos:

```
use router::Router;

fn main() {
    let mut router = Router::new();
    router.get("/", eu_sou_goku, "index")
        .post("/", retrucar, "post");

    Iron {
        handler: router,
        threads: 8,
        timeouts: Timeouts {
            keep_alive: Some(Duration::from_secs(10)),
            read: Some(Duration::from_secs(10)),
            write: Some(Duration::from_secs(10))
        }
    }.http("localhost:3000").unwrap();
}

fn eu_sou_goku(_: &mut Request) -> IronResult<Response> {
    Ok(Response::with((status::Ok, "Oi, eu sou o Goku!")))
}

fn retrucar(request: &mut Request) -> IronResult<Response>{
    let mut body = Vec::new();
    request
        .body
        .read_to_end(&mut body)
        .map_err(|e| IronError::new(e, (status::InternalServerError, "Error ao ler o request")))?;
    Ok(Response::with((status::Ok, body)))
}
```

Veja que a modificação ocorreu nas primeiras linhas de código

com a adição de um `Router` e as definições de caminhos `GET` e `POST` :

```
let mut router = Router::new();
    router.get("/", eu_sou_goku, "index")
        .post("/", retrucar, "post");

Iron {
    handler: router,
    ...
}
```

Muitas vezes precisamos que nosso `GET` receba parâmetros como nomes, datas, rotas e conteúdos. Para podermos passar algum parâmetro para o `GET`, basta adicionar uma especificação de parâmetro em sua rota, como a linha `.get("/:query", ola_vc, "query")`, e implementar a função `ola_vc`, que se responsabilizará por tratar esse parâmetro.

```
fn ola_vc(request: &mut Request) -> IronResult<Response>{
    let ref query = request.extensions.get:::<Router>().unwrap().find(
        "query").unwrap_or("/");
    Ok(Response::with((status::Ok, *query)))
}
```

O `query` é o parâmetro passado para a `Router::get`. Ele procura pela tag `query` e é o nome do parâmetro que definimos na URL. Como não queremos tomar ownership do `request`, temos de passá-lo para a função `ola_vc` por meio de um borrow `&mut Request`.

Devido à forma como passamos o `request`, precisamos lidar com seus valores utilizando conceitos de referência e aplicando a palavra `ref`, diferente do que o `Response::with` espera. Assim, ao passar o resultado de `query` para `response`, vamos dereferenciar utilizando `*` (asterisco) para passarmos um valor do tipo `&str`. Mas, e se quisermos usar o `query` para montar uma

frase?

```
fn ola_vc(request: &mut Request) -> IronResult<Response>{
    let ref query = request.extensions.get::
```

Nosso resultado será:

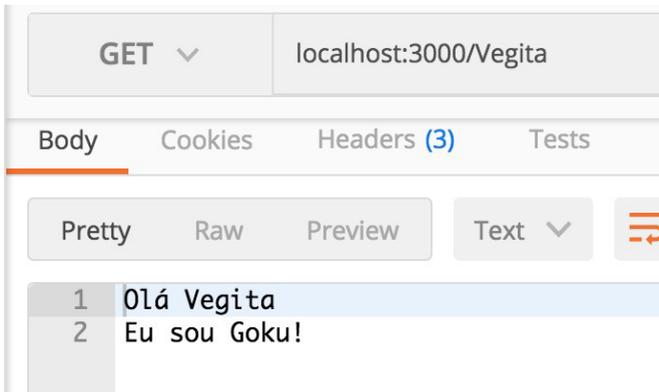


Figura 13.3: Olá Vegita

13.2 MAIS DETALHES DO IRON

Agora que exploramos melhor algumas das qualidades e opções que o Iron nos dá, podemos fazer uma pausa para analisar melhor o que temos até agora. Vamos dissecar o primeiro código,

pois passamos muito rápido por diversos pontos, como os módulos disponibilizados:

```
extern crate iron;

use iron::prelude::*;
use iron::status;

fn main() {
    Iron::new(|_: &mut Request| {
        Ok(Response::with((status::Ok, "Ola Goku!!!")))
    }).http("localhost:3000").unwrap();
}
```

Surpreendentemente, existe um monte de pontos para dissecar em um trecho tão curto de código. O Iron é composto por vários componentes modulares e tem um design muito extensível. Assim, seu código resultante é muitas vezes denso e bastante flexível, proporcionando muito significado semântico, com pouca sintaxe.

Considerando somente as 3 primeiras linhas do código, temos as seguintes informações:

- A crate Iron foi invocada para uso.
- Ela disponibilizou todos (`prelude::*`) os símbolos disponíveis no módulo `prelude` .
- E disponibilizou o módulo `status`, (`status::Ok`, `"Ola Goku!!!"`) .
- O módulo prelúdio disponibiliza diversas structs que foram usadas como `Request` , `Response` e `Iron` .
- Iron é a struct base do módulo e, talvez, de toda lib, pois ela implementa um handler genérico `<H>` que gerencia os `requests` dos clientes, define o número de threads com

um `usize` e usa a struct `Timeouts` para definir os timeouts do servidor.

A implementação do método `new` somente necessita de um handler do tipo `H` para nos retornar um servidor base. Sua definição é dada por `fn new(handler: H) -> Iron<H>`, cujo número default de threads é `8 * num_cpus` (número de CPUs).

- `Response` é a struct responsável por criar a entidade de resposta do servidor. Ela possui 4 campos: `status: Option<Status>`, `headers: Headers`, `extensions: TypeMap`, `body: Option<Box<WriteBody>>`, sendo respectivamente o status da resposta (um `enum` com diversos tipos, como `status::Ok` ou `status::NotFound`); a struct responsável pelos headers da resposta (definida como `data: VecMap<HeaderName, Item>`); `extensions`, que é um mapa chaveado por tipos; e `body`, que corresponde ao corpo da mensagem enviada.
- `Request` é a struct responsável por receber a requisição feita pelo cliente. É constituída por diversos campos, como os da `Response`, e por mais alguns de gerenciamento de socket e URL, além do campo `methods`, definido pelos tipos constituintes da RFC 7231 (<https://tools.ietf.org/html/rfc7231>).

A segunda parte de interesse no código é:

```
fn main() {  
    Iron::new(/*...*/).http("localhost:3000").unwrap();  
}
```

Essa parte é responsável por inicializar o servidor Iron.

Podemos ver que estamos ouvindo o protocolo HTTP em localhost na porta 3000, em `.http("localhost:3000").O` método `http` do Iron aceita qualquer coisa que possa ser analisada por um `std::net::SocketAddr`. Mas a parte mais interessante é entender o `Iron::New()`:

```
Iron::new(|_: &mut Request| {  
    Ok(Response::with((status::Ok, "Ola Goku!!!")))  
})
```

Vamos começar olhando a assinatura do método `new`. Como já explicamos anteriormente, ele recebe um `handler` e retorna um servidor Iron com a implementação do `Handler` de tipo `H`:

```
impl<H> Iron<H> where H: Handler {  
    pub fn new(handler: H) -> Iron<H>;  
}
```

E dela surge uma `trait` especial chamada `Handler`:

```
pub trait Handler: Send + Sync + Any {  
    fn handle(&self, &mut Request) -> IronResult<Response>;  
}
```

Os `Handlers` são os componentes centrais do Iron, e responsáveis por receber um `request` e gerar uma `response`. É válido fazer uma comparação com os *controllers* do framework MVC, porém, os `handlers` possuem muito menos limitações de como podem ser usados, especialmente por poderem carregar parte do que seria o "controller" ao servidor.

A instância do tipo `Handler`, passada para o `Iron::new()`, será a principal forma de manipular o nosso servidor. No nosso caso, o `Handler` passado é uma closure (lembrando de que o tipo do retorno não é necessário):

```
|_: &mut Request| -> IronResult<Response> {
```

```
Ok(Response::with((status::Ok, "Ola Goku!!!")))
}
```

Mas a parte que parece se destacar é a `Response::with()`. Ela na verdade constrói a `Response` que enviaremos de volta ao cliente. Observe sua implementação:

```
pub fn with<M: Modifier<Response>>(modifier: M) -> Response {
    Response::new().set(modifier)
}
```

Definitivamente a quantidade de `Modifier` chama a atenção, por isso vamos ver o `Modifier<Response>` com mais calma.

MODIFIERS

Os *modifiers* (ou modificadores) são um dos principais tipos de extensão no Iron. Eles permitem que você defina novas maneiras de fazer alterações em outros tipos. No Iron, os tipos de `request` e `response` são aqueles que mais aplicamos. Para entender como os `modifiers` funcionam, vamos verificar a sua `trait`:

```
pub trait Modifier<T> {
    fn modify(self, &mut T);
}
```

Bastante simples, né? Tudo que um `modifier` faz é modificar algo (`T`). Contudo, a `crate modifier` nos traz outra `trait` interessante, a `Set`:

```
pub trait Set {
    fn set<M: Modifier<Self>>(mut self, modifier: M) -> Self
    where Self: Sized {
        modifier.modify(&mut self);
        self
    }
}
```

```
fn set_mut<M: Modifier<Self>>(&mut self, modifier: M) ->
&mut Self {
    modifier.modify(self);
    self
}
}
```

No Iron, tanto request quanto response implementam Set . Isso permite que modifiers sejam encadeados de forma bastante simples. Se voltarmos à implementação do Response::with , veremos que tudo o que ela faz é criar uma response vazia a partir do ::new , para então aplicar um modifier, Set .

Agora que entendemos melhor o funcionamento deles e das principais structs do Iron, podemos resumir tudo com pouco mais de um *tweet*: para criar um servidor, basta usar Iron::new , passar um handler (no nosso caso, uma closure), e começar a escutar a porta. Nosso handler retornará uma nova Response com o campo de status setado como status::Ok (200), e com o corpo que definimos no resto da tupla. Isso tudo acontecerá pela aplicação do modifier .

Caso você precise que o modifier seja aplicado em um tipo diferente do que já foi implementado, é necessário implementar a trait para aquele tipo. Veja as implementações disponíveis em: <https://github.com/iron/iron/blob/master/src/modifiers.rs>.

13.3 IRON TESTES

Criar servidores é muito útil e interessante. Porém, se quisermos colocar nossos serviços em produção, devemos ter um

conjunto de regras que nos permitirá garantir uma qualidade de entrega, tanto em nível de negócio quanto de código.

Outro fator importante de testes é que eles garantem que futuras modificações não causem impacto negativo em nosso serviço. Para isso, precisamos entender como podemos testar o Iron. Iniciamos este processo adicionando a crate do `iron-test` :

```
[dependencies]
iron = "*"
iron-test = "*"
```

Primeiramente, devemos habilitar os módulos do `iron-test` , incluindo o `extern crate iron_test;` e posteriormente use `iron_test::{request, response};` . A partir disto, podemos pensar em um primeiro módulo de testes, com um `assert` simples, que se constitui de uma `response` gerada a partir de uma rota simulada ao `handler` (neste caso, o `GokuHandler`). Depois disso, extraímos a resposta para um `result` e fazemos o `assert_eq!` para validar se a resposta de saída corresponde à que esperávamos:

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_goku_handler() {
        let response = ...
        let result_body = response::extract_body_to_bytes(response);

        assert_eq!(result_body, b"Oi, eu sou o Goku!!!");
    }
}
```

Agora devemos pensar no que queremos que aconteça com um

request quando um handler específico é chamado – neste caso, retornar b"Oi, eu sou o Goku!!!".

```
let response = request::get("http://localhost:3000/hello",
                             Headers::new(),
                             &GokuHandler).unwrap();
```

Para que o nosso teste passe, precisamos de um handler chamado `GokuHandler`, implementado desta forma:

```
use iron::prelude::*;
use iron::{Handler, status};
#[allow(unused_imports)]
use iron_test::{request, response};

struct GokuHandler;

impl Handler for GokuHandler {
    fn handle(&self, _: &mut Request) -> IronResult<Response> {
        Ok(Response::with((status::Ok, "Oi, eu sou o Goku!!!")))
    }
}
```

Para que tudo funcione, basta adicionar a função `main`. Ela nos garantirá que o servidor `iron` estará rodando em um arquivo binário, e teremos os testes marcados como `#[test]`:

```
extern crate iron;
extern crate iron_test;

use iron::prelude::*;
use iron::{Handler, Headers, status};
#[allow(unused_imports)]
use iron_test::{request, response};

struct GokuHandler;

impl Handler for GokuHandler {
    fn handle(&self, _: &mut Request) -> IronResult<Response> {
        Ok(Response::with((status::Ok, "Oi, eu sou o Goku!!!")))
    }
}
```

```

fn main() {
    Iron::new(GokuHandler).http("localhost:3000").unwrap();
}

#[test]
fn test_goku_handler() {
    let response = request::get("http://localhost:3000/hello",
                                Headers::new(),
                                &GokuHandler).unwrap();
    let result_body = response::extract_body_to_bytes(response);

    assert_eq!(result_body, b"Oi, eu sou o Goku!!!");
}

```

Agora voltando ao nosso exemplo Olá Vegeta, Eu sou o Goku! , digamos que agora queremos que dois parâmetros sejam passados, como nome (name) e ação (action). Um teste apropriado para o nosso caso seria algo como o apresentado a seguir, no qual definimos uma variável header do tipo application/x-www-form-urlencoded para explicitar como vamos passar o conteúdo da URL e os parâmetros de chamada pelo "name=Goku&action=Morra" . Com isso, nosso objetivo é ter como retorno: b"Goku, Morra!" .

```

[dependencies]
iron = "*"
iron-test = "*"
urlencoded = "*"

#[cfg(test)]
mod test {
    use iron::Headers;
    use iron::headers::ContentType;

    use iron_test::{request, response};

    use super::SpeechHandler;

    #[test]
    fn test_body() {

```

```

    let mut headers = Headers::new();
    headers.set(ContentType("application/x-www-form-u
rleencoded".parse().unwrap()));
    let response = request::post("http://localhost:3000/",
                                headers,
                                "name=Goku&action=Morra",
                                &SpeechHandler);
    let result = response::extract_body_to_string(response.un
wrap());

    assert_eq!(result, b"Goku, Morra!");
}
}

```

Uma possível solução para este caso é a apresentada a seguir, que extrai da URL os parâmetros que esperamos. Caso algum erro aconteça, um `panic!` é gerado com a mensagem `"UrlEncoding is not correct"`.

```

extern crate iron;
extern crate iron_test;
extern crate urlencoded;

use iron::{Handler, status};
use iron::prelude::*;

use urlencoded::UrlEncodedBody;

struct SpeechHandler;

impl Handler for SpeechHandler {
    fn handle(&self, req: &mut Request) -> IronResult<Response> {
        let body = req.get_ref:::<UrlEncodedBody>()
            .expect("UrlEncoding is not correct");
        let name = body.get("name").unwrap()[0].to_owned();
        let action = body.get("action").unwrap()[0].to_owned();

        Ok(Response::with((status::Ok, name + ", " + &action + "!
)))
    }
}

```

O `SpeechHandler` define um *parser* com dois parâmetros,

name e action , no qual somente o primeiro de cada é solicitado e a ownership é passada para a função handler . Esta retorna uma &str dentro de Response . A implementação de get_ref é a seguinte:

```
fn get_ref<P>(&mut self)
-> Result<&<P as Key>::Value, <P as Plugin<Self>>::Error>
where
    P: Plugin<Self>,
    Self: Extensible,
    <P as Key>::Value: Any,
{}
```

Ao aplicarmos esse código em main , obtemos:

```
fn main() {
    Iron::new(SpeechHandler).http("localhost:3000").unwrap();
}
```

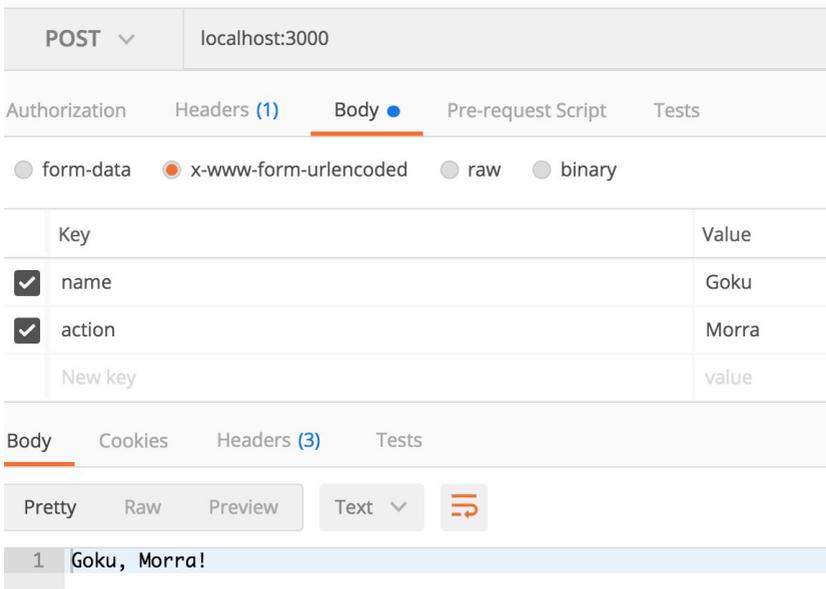


Figura 13.4: Resultado do post/urlencoded

Como utilizamos o `UrlEncodedBody` , nosso handler automaticamente inferiu que estávamos usando um método `POST` . Portanto, para utilizar o método `GET` , teríamos de usar o `UrlEncodedQuery` com uma `query` , como `localhost:3000/?name=xxx&action=yyy` .

Como já testamos os handlers, devemos nos aprofundar mais em outros tipos de estruturas para testar, como os routers, já que eles mesmos podem gerenciar alguns de nossos handlers. É um teste quase de integração, já que eles testam suas funcionalidades com valores de fácil compreensão.

A primeira coisa que queremos é entender o que um teste de router precisa fazer. Basicamente temos de garantir que uma chamada a uma URL específica deve retornar um valor específico. Para este exemplo, é legal termos duas rotas `GET` , uma que exhibe o `ki` e outra que responde alguma frase com o nome da pessoa. Assim, teremos as rotas:

- `GET /valor/:ki` .
- `GET /nome/:name` .

Podemos começar testando a rota `ki` . Comparando-a com os testes anteriores, a diferença é que estamos utilizando o `app_router` em vez de um handler para testar:

```
#[cfg(test)]
mod test {
    use iron::Headers;

    use iron_test::{request, response};

    use super::{app_router};

    #[test]
```

```

fn test_router() {
    let response = request::get("http://localhost:3000/valor/
8000",
                                Headers::new(),
                                &app_router());
    let result = response::extract_body_to_bytes(response.unw
rap());

    assert_eq!(result, b"Eh mais de 8000");
}
}

```

O que nos interessa nesse teste é principalmente o caminho `/valor/8000` e o `&app_router`, pois é isso que nos permite testar um `router`. Agora precisamos apresentar uma solução para `app_router`. Ela consistirá em retornar a variável `router`, que foi criada pelo `Router::new`, teve o caminho `valor/:ki` setado e seu gerenciamento feito pelo handler `KiHandler`.

```

fn app_router() -> Router {
    let mut router = Router::new();
    router.get("valor/:ki", KiHandler, "router");
    router
}

```

Vamos à implementação de `KiHandler`. Como precisamos dos parâmetros enviados (neste caso, `:ki`), recuperamos os parâmetros do `request` com `req.extensions.get::<router::Router>()`, e aplicamos a função `.find("ki")` para obter o valor associado a `ki`. No caso do teste anterior, será `8000`.

```

struct KiHandler;

impl Handler for KiHandler {
    fn handle(&self, req: &mut Request) -> IronResult<Response> {
        let params = req.extensions
            .get::<router::Router>()
            .expect("Router não encontrado a partir da extensão d

```

```

o Request.");
    let ki = params.find("ki").unwrap();

    Ok(Response::with((status::Ok, "Eh mais de ".to_string()
+ ki)))
    }
}

```

Agora precisamos adicionar ao `app_router` uma nova rota, a `/nome/:name`. Para isso, escrevemos o seguinte teste que cumpre as expectativas setadas no início do teste de router:

```

#[test]
fn test_router_name() {
    let response = request::get("http://localhost:3000/nome/Vegit
a",
                                Headers::new(),
                                &app_router());
    let result = response::extract_body_to_bytes(response.unwrap(
));

    assert_eq!(result, b"Vegita\nEu sou Goku!");
}

```

Assim, é preciso implementar a nova rota ao router adicionando: `.get("nome/:name", NameHandler, "name router")`.

```

fn app_router() -> Router {
    let mut router = Router::new();
    router.get("valor/:ki", KiHandler, "ki router")
        .get("nome/:name", NameHandler, "name router");
    router
}

```

Também precisamos criar o handler `NameHandler`, que tem o funcionamento muito semelhante ao `KiHandler`, mas atua sobre a `&str name`:

```

impl Handler for NameHandler {
    fn handle(&self, req: &mut Request) -> IronResult<Response> {

```

```

        let params = req.extensions
            .get::()
            .expect("Router não encontrado a partir da extensão d
o Request.");
        let name = params.find("name").unwrap().to_owned();

        Ok(Response::with((status::Ok, name + "\nEu sou Goku!")))
    }
}

```

O resultado dos nossos testes e suas implementações será:

```

extern crate iron;
extern crate iron_test;
extern crate router;

use iron::{Handler, status};
use iron::prelude::*;

use router::Router;

struct KiHandler;
struct NameHandler;

impl Handler for KiHandler {
    fn handle(&self, req: &mut Request) -> IronResult<Response> {
        let params = req.extensions
            .get::()
            .expect("Router não encontrado a partir da extensão d
o Request.");
        let ki = params.find("ki").unwrap();

        Ok(Response::with((status::Ok, "Eh mais de ".to_string()
+ ki)))
    }
}

impl Handler for NameHandler {
    fn handle(&self, req: &mut Request) -> IronResult<Response> {
        let params = req.extensions
            .get::()
            .expect("Router não encontrado a partir da extensão d
o Request.");
        let name = params.find("name").unwrap().to_owned();

```

```

        Ok(Response::with((status::Ok, name + "\nEu sou Goku!")))
    }
}

fn app_router() -> Router {
    let mut router = Router::new();
    router.get("valor/:ki", KiHandler, "ki router")
        .get("nome/:name", NameHandler, "name router");
    router
}

fn main() {
    Iron::new(app_router()).http("localhost:3000").unwrap();
}

#[cfg(test)]
mod test {
    use iron::Headers;

    use iron_test::{request, response};

    use super::{app_router};

    #[test]
    fn test_router_ki() {
        let response = request::get("http://localhost:3000/valor/
8000",
                                   Headers::new(),
                                   &app_router());
        let result = response::extract_body_to_bytes(response.unw
rap());

        assert_eq!(result, b"Eh mais de 8000");
    }

    #[test]
    fn test_router_name() {
        let response = request::get("http://localhost:3000/nome/V
egita",
                                   Headers::new(),
                                   &app_router());
        let result = response::extract_body_to_bytes(response.unw
rap());
    }
}

```

```
    assert_eq!(result, b"Vegita\nEu sou Goku!");  
  }  
}
```

Assim como todas bibliotecas Rust, Iron encontra-se em estado de desenvolvimento. Felizmente, estes exemplos cobrem grande parte dos problemas básicos que enfrentamos no desenvolvimento de serviços, e suas implementações não devem variar significativamente quando a versão estável for alcançada.

No próximo capítulo, veremos brevemente como desenvolver um pequeno servidor com Nickel, outro famoso framework.

BRINCANDO COM NICKEL

O Nickel é um framework muito parecido com o Iron, porém, ele explora algumas formas diferentes de desenvolvimento. Um aspecto interessante é como ele explora macros e alguns aspectos funcionais, como o conceito de que todas as funções devem retornar valores que possam ser operados por novas funções.

Infelizmente, o pacote mínimo de Nickel possui centenas de dependências que não necessariamente serão usadas. E isso incorpora muito ruído.

Como dito no capítulo anterior, a documentação destes frameworks ainda é muito superficial. Com isso em mente, vale fazer uma pequena demonstração do Nickel, começando com o clássico Hello Goku !

Para isso, precisamos criar um novo projeto cargo e adicionar a dependência nickel :

```
$ cargo new nickel-hello-goku --bin
```

```
[dependencies]  
nickel = ""
```

Agora temos de adicionar o código para o nosso Hello Goku! :

```

#[macro_use]
extern crate nickel;

use nickel::Nickel;

fn main() {
    let mut server = Nickel::new();

    server.utilize(router! {
        get "*" => |_req, _res| {
            "Hello Goku!"
        }
    });

    server.listen("127.0.0.1:3000");
}

```

Nossa primeira linha é um pouco diferente desta vez. A presença do `#[macro_use]` é uma anotação para a `crate nickel`, pois utilizamos a macro `router!`. Depois declaramos o servidor mutável com `let mut server = Nickel::new();`. No servidor, aplicamos a função `utilize` (explicada a seguir) e fazemos o servidor escutar em `localhost:3000`.

UTILIZE()

Essa função registra um handler de middleware, que será invocado entre outros handlers de middleware, antes de cada Request . O middleware pode ser empilhado e é invocado na mesma ordem em que foi registrado.

Um handler de middleware é quase idêntico a um handler de routes regular, com a única diferença de que ele espera um resultado de Action ou de NickelError . Isso se dá para indicar a outros handlers de middleware (se existirem), mais abaixo na pilha, que devem continuar, ou se a invocação do middleware deve ser interrompida após o handler atual.

14.1 ROUTING

O próximo passo é entender como podemos criar diferentes rotas. Vamos criar duas: uma que responde Hello Goku , e outra que responde o nome do usuário.

```
#[macro_use]
extern crate nickel;

use nickel::{Nickel, HttpRouter};

fn main() {
    let mut server = Nickel::new();

    server.get("/hello", middleware!("Hello Goku"))
        .get("/user/:usuario", middleware! { |request|
            format!("ola {:?}!", request.param("usuario").unwrap()
        ))
    });
```

```
server.listen("127.0.0.1:3000");
}
```

A única mudança aqui é que substituímos o `utilize` por métodos `get` empilhados com `middlewares` para controlarem os `requests` e `responses`.

14.2 LIDANDO COM JSON

Grande parte das comunicações via APIs Rest é feita por meio de `GETs` e `POSTs`. Já falamos bastante sobre `GETs` no capítulo anterior e neste, mas muitos dos `request` são feitos com `POSTs`.

O `POST` é feito geralmente em dois formatos, `XML` e `JSON`. A maior parte das bibliotecas mais modernas implementa primeiro soluções `JSON` (creio eu), por ser um formato mais claro. Assim, queremos receber um `POST` de `JSON` para retornar uma `string`, com algum texto específico.

Para isso, devemos adicionar a dependência `rustc-serialize = "*" ao arquivo toml.`

```
#[macro_use]
extern crate nickel;
extern crate rustc_serialize;

use nickel::{Nickel, HttpRouter, JsonBody};

#[derive(RustcDecodable, RustcEncodable)]
struct Pessoa {
    nome: String,
    sobrenome: String,
}

fn main() {
    let mut server = Nickel::new();
```

```

server.post("/ola", middleware! { |request, response|
  let person = request.json_as::<Pessoa>().unwrap();
  format!("Ola sra. {}, {}", person.sobrenome, person.nome)
});

server.listen("127.0.0.1:3000");
}

```

A única novidade aqui é a forma como o `request` está sendo lido, pois tentamos entender o JSON que está sendo passado como uma `struct Pessoa`. Isso se dá pela função `.json_as::<Pessoa>()`. Uma outra vantagem do Nickel é a facilidade para se trabalhar com dados em templates.

14.3 TEMPLATES EM NICKEL

Pela breve experiência que tive com Rails, era bastante simples gerar templates a partir do servidor — algo que muitos frameworks não consideram uma prioridade, especialmente depois do surgimento do React, que facilitou bastante esse serviço. Mas apresentar templating do ponto de vista do Nickel pode ser bastante interessante devido às características fortes de performance e de concorrência do Rust. Elas garantem velocidade de renderização e funcionamento.

Para começar com o sistema de templates, podemos criar algo bem simples e salvar como `template.tpl`, fora da pasta `src/`. Nossa ideia aqui é bem simples, gerar um template que diz `Olá {{Nome para exibir}}`.

```

<html>
  <body>
    <h1>
      Olá {{ nome }}!
    </h1>
  </body>

```

```
</html>
```

Isso deverá ser passado para a função `response.render(template, data)`:

```
#[macro_use]
extern crate nickel;

use std::collections::HashMap;
use nickel::{Nickel, HttpRouter};

fn main() {
    let mut server = Nickel::new();

    server.get("/", middleware! { |_, response|
        let mut data = HashMap::new();
        data.insert("nome", "Julia");
        return response.render("template.tpl", &data);
    });

    server.listen("127.0.0.1:3000");
}
```

Neste caso, o middleware cria uma relação da `HashMap` entre o valor encontrado no template e o valor que deve ser inserido pelo código. A variável `data` recebe dois parâmetros na função `insert(template_key, server_value)`: o primeiro é a chave existente no template que será substituída pelo valor do segundo template.

```
let mut data = HashMap::new();
data.insert("nome", "Julia");
```

E se quiséssemos inserir o nome do usuário no nosso template? Teríamos de recebê-lo como parâmetro. Para isso, poderíamos usar o seguinte código para receber o parâmetro `usuario` pela rota `GET` e devolvê-lo no template:

```
server.get("/:usuario", middleware! { |request, response|
    let mut data = HashMap::new();
```

```
data.insert("nome", request.param("usuario").unwrap());  
return response.render("template.tpl", &data);  
});
```

É interessante dar uma olhada na macro `middleware!`, em <http://nickel.org.github.io/nickel.rs/nickel/macro.middleware!.html>. Ela é apresentada em routing, template e JSON, pois é uma das ferramentas mais fortes do Nickel.

Essa macro nos permite definir em uma closure o que é o `request` e o que é o `response`, e definir a relação entre eles por meio de regras mais livres.

Como já falamos, Iron e Nickel são frameworks diferentes, mas com aspectos importantes: a alta capacidade de concorrência do Iron, e o aspecto mais funcional de Nickel. Agora seguiremos para um exemplo com Hyper, que nos traz uma grande liberdade para implementar aspectos de concorrência e funcionais.

HYPER: SERVIDORES DESENVOLVIDOS NO MAIS BAIXO NÍVEL

Hyper é uma biblioteca de implementação HTTP moderna e rápida, escrita em Rust para o Rust. É uma linguagem de baixo nível com abstração de tipos, segura sobre o HTTP. Ela provê implementações para cliente e servidor, e permite desenvolver aplicações complexas em Rust.

Um fato curioso sobre a Hyper é que ela utiliza o `async IO` (sockets não bloqueantes) por meio das bibliotecas Tokio e Futures. A ideia é que a biblioteca está avançando rapidamente em direção à sua versão 1.0.

O interessante do Hyper é que ele nos permitirá explorar bem alguns conceitos de concorrência, mesmo que muitos deles sejam nativos ao próprio Hyper. Agora, sabendo do que se tratam nossos experimentos, vamos começar com um *Hello World*:

```
[dependencies]
hyper = "*"
service-fn = {git = "https://github.com/tokio-rs/service-fn"}

extern crate hyper;
extern crate service_fn;
```

```

use hyper::header::{ContentLength, ContentType};
use hyper::server::{Http, Response};
use service_fn::service_fn;

static TEXTO: &'static str = "Olá, Mundo do Hyper!";

fn run() -> Result<(), hyper::Error> {
    let addr = ([127, 0, 0, 1], 3000).into();

    let ola = || Ok(service_fn(|_req|{
        Ok(Response::<hyper::Body>::new()
            .with_header(ContentLength(TEXTO.len()) as u64))
            .with_header(ContentType::plaintext())
            .with_body(TEXTO))
    }));

    let server = Http::new().bind(&addr, ola)?;
    server.run()
}

fn main() {
    run();
}

```

O que podemos ver de diferente no arquivo `toml` ? Dessa vez, importamos uma biblioteca que não se encontra no `crates.io` . Inclusive, esse sistema nos permite definir a *branch* que queremos importar, caso não seja a *master*, algo como: `service-fn = {git = "https://github.com/tokio-rs/service-fn", branch = "outra"}` . É também a primeira vez que mostramos um valor global, `TEXTO` , cuja vida perdura durante todo o programa por causa do `&'static` .

Além disso, definimos nosso endereço ao usarmos uma estrutura de dados do tipo *tupla*, com um array de inteiros e um inteiro. Outro fato interessante é a declaração do tipo `ola` como uma função anônima que não recebe parâmetros, mas responde um `OK` que recebe outra função anônima.

Por fim, criamos um servidor unindo a estrutura de dados à nossa função `ola`. Creio que este seja o desenvolvimento mais funcional até o momento.

15.1 CRIANDO UM SERVIDOR HELLO WORLD MAIS ROBUSTO

Agora podemos nos aprofundar em como o `Hyper` faz algumas coisas, sem a abstração do `service-fn`, pois assim poderemos entender como essa biblioteca lida com a concorrência.

```
[dependencies]
futures = "0.1.14"
hyper = "0.11.2"
```

No nosso arquivo `main`, podemos adicionar as crates que vamos utilizar, aplicando:

```
extern crate hyper;
extern crate futures;
```

Nosso primeiro passo é implementar o serviço (`service`) que atenderá nosso `request`. Neste caso, nossa relação entre `request` e `response` será representada pela `struct` `Ola`.

Então, precisamos implementar `Service` para a `struct` `Ola`, mas isso gera um certo boilerplate para definir tipos de `Request`, `Response` e erro, como a seguir:

```
use hyper::server::{Http, Request, Response, Service};
//...
struct Ola;

impl Service for Ola {

    type Request = Request;
    type Response = Response;
```

```

    type Error = hyper::Error;
    type Future = Box<Future<Item=Self::Response, Error=Self::Error>>;

    //...
}

```

Note a presença do tipo `Future` ; isso se deve ao fato de que ele representa uma promessa de resposta da chamada, que será resolvida pela função `call()` . Já que falamos de `call()` , temos o próximo passo, que será implementar a própria chamada.

```

use futures::future::Future;

use hyper::header::ContentLength;
use hyper::server::{Http, Request, Response, Service};

//...

struct Ola;

const FRASE: &'static str = "Ola Goku!";

impl Service for Ola {

    //...

    fn call(&self, _req: Request) -> Self::Future {
        Box::new(futures::future::ok(
            Response::new()
                .with_header(ContentLength(FRASE.len() as u64))
                .with_body(FRASE)
        ))
    }
}

```

A função `call` recebe um `Request` que não estamos utilizando neste momento, e retorna uma `Future` com *headers* e *body* associados à frase que definimos como "Ola Goku!" . A vantagem desse método aparece na limpeza da função `main` , que agora ficaria assim:

```
fn main() {
    let addr = "127.0.0.1:3000".parse().unwrap();
    let servidor = Http::new().bind(&addr, || Ok(01a)).unwrap();
    servidor.run().unwrap();
}
```

Com o `let addr` definindo o endereço e o `let servidor` definindo o servidor, temos a inicialização do servidor com `servidor.run().unwrap()`.

15.2 FAZENDO NOSSO SERVIDOR RESPONDER UM POST

Difícilmente um servidor será como o que elaboramos, pois geralmente queremos que ele responda contextos específicos a URLs e métodos específicos. O primeiro passo pode ser definir uma rota para retornarmos algo em um método `POST`, e uma rota para retornamos algo em um método `GET`.

Para isso, precisamos adicionar dois módulos `use hyper:: {Method, StatusCode};`, e fazer nosso serviço ter um pouco mais de sentido. Uma boa ideia seria chamar nossa `struct` de `DbzServer` em vez de `01a`, já que o nome `01a` é bastante vago e inapropriado para um nome de servidor.

Assim, o corpo do nosso método `call` em routing seria como a seguir:

```
impl Service for DbzServer {
    // Types aqui

    fn call(&self, req: Request) -> Self::Future {
        let mut response = Response::new();

        match (req.method(), req.path()) {
```

```

        (&Method::Get, "/" ) => {
            response.set_body(FRASE);
        },
        (&Method::Post, "/dbz") => {
            // mais detalhes no futuro
        },
        _ => {
            response.set_status(StatusCode::NotFound);
        },
    };

    Box::new(futures::future::ok(response))
}
}

```

Nosso método `call` ficou bastante interessante agora, mas temos um ponto fraco, a declaração de `response` `let mut response = Response::new();`, por ser mutável. Para entendermos o que aplicar em nossa variável `response`, temos um `match` que analisa dois parâmetros: o método que foi chamado, com `req.method()`, e o caminho que foi usado, com `eq.path()`.

O primeiro é um `GET` com caminho `"/"`, que retorna a frase "Olá Goku!", e o segundo é um `POST` na rota `"/dbz"`, que ainda não vimos em detalhes. Já o terceiro é qualquer outro valor, que retorna um código `StatusCode::NotFound` (*404 Not Found*). A resposta é então introduzida dentro do `Box` de `futures`, `Box::new(...)`, e retornada como um `future::ok`.

O primeiro passo do nosso servidor, e o mais simples, seria simplesmente retornar o corpo do `request`, algo como:

```

(&Method::Post, "/dbz") => {
    response.set_body(req.body());
},

```

Digamos que nosso objetivo aqui seja fazer mais do que

simplesmente retornar o corpo do `request`. Agora é o momento para fazermos uma pequena demonstração de como aplicar alguma função específica ao nosso `request`, e não necessitamos de nada muito complexo.

Nossa meta é transformar as informações do `request` em `UPPERCASE`. Para isso, devemos usar a função de `map` sobre cada byte correspondente à String do `request`, com a closure `|byte| byte.to_ascii_uppercase()`.

Para fazer isso, precisamos de alguns imports extras, como: a crate de gerenciamento de `Ascii`, as crates de gerenciamento de `Streams` em `futures`, e o componente `Chunk` da crate `hyper`:

```
use std::ascii::AsciiExt;
use futures::Stream;
use futures::stream::Map;
use hyper::Chunk;
```

Como temos novos imports, vamos analisá-los um pouco melhor. O `Body` implementa a trait `Stream`, derivada de `futures`, produzindo um amontoado de `Chunk` (pedaços). Um `Chunk` é apenas um tipo derivado do `Hyper` que representa um conjunto de bytes, e é facilmente convertido em outros tipos de containers de bytes.

Desta forma, podemos definir nossa função que transforma tudo em letras maiúsculas `para_maiusculas`:

```
fn para_maiusculas(chunk: Chunk) -> Chunk {
    let maiusculas = chunk.iter()
        .map(|byte| byte.to_ascii_uppercase())
        .collect::<Vec<u8>>();
    Chunk::from(maiusculas)
}
```

Ainda precisamos fazer mais algumas alterações como atualizar o tipo `Stream` que nossa `response` está usando. Por padrão, o tipo é um `hyper::Body`, mas o `response` pode usar qualquer `Stream` que implemente `AsRef<[u8]>`.

Assim, vamos utilizar o tipo `type Response = Response<Map<Body, fn(Chunk) -> Chunk>>`; , que opera em cima de um `Map`. As mudanças ficam dessa forma:

```
impl Service for DbzServer {
    // other types stay the same
    type Response = Response<Map<Body, fn(Chunk) -> Chunk>>;

    fn call(&self, req: Request) -> Self::Future {
        // ...
        (&Method::Post, "/dbz") => {
            response.set_body(req.body().map(para_maiusculas
as _));
        },
        // ...
    }
}
```

O atual estado do código é:

```
extern crate hyper;
extern crate futures;

use std::ascii::AsciiExt;
use futures::Stream;
use futures::stream::Map;
use futures::future::Future;

use hyper::{Chunk, Body};
use hyper::{Method, StatusCode};
use hyper::server::{Http, Request, Response, Service};

fn main() {
    let addr = "127.0.0.1:3000".parse().unwrap();
    let servidor = Http::new().bind(&addr, || Ok(DbzServer)).unwr
```

```

ap();
    servidor.run().unwrap();
}

struct DbzServer;

impl Service for DbzServer {

    type Request = Request;
    type Response = Response<Map<Body, fn(Chunk) -> Chunk>>;
    type Error = hyper::Error;
    type Future = Box<Future<Item=Self::Response, Error=Self::Error>>;

    fn call(&self, req: Request) -> Self::Future {
        let mut response = Response::new();

        match (req.method(), req.path()) {
            (&Method::Post, "/dbz") => {
                response.set_body(req.body().map(para_maiusculas
as _));
            },
            _ => {
                response.set_status(StatusCode::NotFound);
            },
        };

        Box::new(futures::future::ok(response))
    }
}

fn para_maiusculas(chunk: Chunk) -> Chunk {
    let maiusculas = chunk.iter()
        .map(|byte| byte.to_ascii_uppercase())
        .collect::<Vec<u8>>();
    Chunk::from(maiusculas)
}

```

Nosso atual código é bastante simples, mas temos um servidor definido de forma imutável, o que deveria nos garantir certa estabilidade e previsibilidade ao longo do curso de ação. Temos também um serviço que é capaz de aplicar funções combinadas e,

por fim, uma resposta no formato de `future`, que nos permite não bloquear as threads de serviço caso a thread estagne em um loop.

Assim, podemos concluir que o Hyper é uma biblioteca bastante interessante, ainda mais quando combinada com Iron. O uso de `futures` permite maior controle sobre a geração de threads e garante uma boa integridade da performance em momentos de estresse.

15.3 UMA API GRAPHQL COM HYPER E JUNIPER

Já sabemos como lidar com Hyper, e como servir informações. Nosso objetivo agora é fazer nosso servidor processar uma grande quantidade de informações e servir de acordo com o que o cliente quer. Faremos uso de estruturas de dados, macros e um framework GraphQL chamado Juniper. Infelizmente, Juniper ainda não foi estabilizado como o Hyper, mas é uma alternativa bastante promissora e não acredito que virão mudanças significativas na forma de pensar o serviço. Esta seção foi escrita posteriormente ao desenvolvimento do serviço GraphQL e pode ser encontrado no repositório <https://github.com/naomijub/mtgql/>.

Um novo servidor Hyper

O primeiro commit é simples, e consiste dos mesmos passos de antes: `cargo new mtgql --bin` para gerar o código inicial, adicionar a dependência do Hyper no `cargo.toml` e um código base para fazer o serviço responder. Conforme fizemos antes:

```
//main.rs
```

```

extern crate hyper;

use hyper::{Body, Response, Server};
use hyper::rt::Future;
use hyper::service::service_fn_ok;

static TEXT: &str = "pong";

fn main() {
    let addr = ([127, 0, 0, 1], 3000).into();
    let ping_svc = || {
        service_fn_ok(|_req| {
            Response::new(Body::from(TEXT))
        })
    };
    let server = Server::bind(&addr)
        .serve(ping_svc)
        .map_err(|e| eprintln!("server error: {}", e));
    hyper::rt::run(server);
}

#cargo.toml
[package]
name = "mtgql"
version = "0.1.0"
authors = ["Julia Naomi"]

[dependencies]
hyper = "*"

```

Agora precisamos entender como vamos processar um arquivo JSON com todas as informações. O arquivo utilizado pode ser encontrado em <https://github.com/naomijub/mtgql/blob/master/body.json/>. Este JSON possui informações de cards do jogo Magic The Gathering.

Para trabalharmos com JSON, precisamos incorporar uma nova biblioteca, a **serde** (<https://serde.rs/>), e a boa e velha futures:

```

[dependencies]
hyper = "*"
serde = "*"

```

```
serde_json = ""  
serde_derive = ""  
futures = ""
```

A biblioteca `serde` é dividida em um conjunto de bibliotecas que trabalham de forma complementar, mas sua principal função é serializar e desserializar dados. O primeiro passo que gostaria de tratar é a conversão de um arquivo JSON em uma struct Rust, deixando um pouco de lado o `Hyper`. Para isso, o primeiro passo é incorporar as funcionalidades de `serde` que precisaremos consumir:

```
#[macro_use] extern crate serde_derive;  
#[macro_use] extern crate serde_json;  
extern crate serde;
```

Quando escrevo código Rust que consome macros, sempre deixo as crates que possuem a anotação `#[macro_use]` no topo, conforme o exemplo acima, para facilitar a leitura e entendimento das crates utilizadas.

Agora precisamos de uma forma de ler este JSON e transformá-lo em uma estrutura Rust. Para lermos, precisaremos incorporar duas funcionalidades da `std` que não vêm por padrão, que estão relacionadas à leitura de arquivos. Para isso, adicionamos as linhas `use std::fs::File;`, `use std::io::prelude::*;` e `use std::io::Result;` logo abaixo das crates declaradas. A leitura do arquivo `body.json` (que se encontra na raiz do projeto) pode ser feita através da função `read_fake` (dei o nome de `read_fake` devido à ideia de evoluir este método para um `hyper::client` que faça *request* à API do

Magic):

```
//Result<String> corresponde a std::io::Result<String>
fn read_fake() -> Result<String> {
    let mut contents = String::new();
    File::open("./body.json)?.read_to_string(&mut contents)?;
    Ok(contents)
}
```

A função `read_fake` vai nos retornar um enum `Result` com uma `String` contendo as informações presentes no arquivo `body.json`. A primeira linha da função é a declaração de uma `String` mutável chamada `contents`. A `String` deve ser mutável, pois a linha seguinte altera o conteúdo de `String` adicionando o conteúdo do JSON. A segunda linha consiste de duas partes. A primeira é a leitura do arquivo `./body.json`, que é feita pelo método `open` do namespace `std::fs::File`. A segunda parte consiste na leitura do conteúdo de `body.json` para dentro de `contents`. Podemos ver a presença de dois `?`, eles são um operador unário que é utilizado como sufixo a uma função para substituir a função de `unwrap()`. Veja um exemplo mais detalhado a seguir, no qual representamos a mesma função com `unwrap` e com `?`:

```
//Result<String> corresponde a std::io::Result<String>
fn read_fake() -> Result<String> {
    let mut contents = String::new();
    File::open("./body.json").unwrap().read_to_string(&mut contents).unwrap();
    Ok(contents)
}
```

```
//A linha a seguir é uma forma simplificada da forma com match
File::create("foo.txt)?.write_all(b"Olá Mundo!");
```

```
//Caso menos simples de uso
match File::create("foo.txt") {
    Ok(t) => t.write_all(b"Olá Mundo!"),
```

```

    Err(e) => return Err(e.into()),
}

```

O próximo passo é criarmos a função `fake_json`, que retorna uma struct `Card` ou um JSON, através do tipo `serde_json::Value`. Em uma primeira instância, queremos que a função retorne a struct `Card` e, para isso, a escrevemos da seguinte forma:

```

pub fn fake_json() -> Card {
    let v: Card = from_str(read_fake().unwrap().as_str()).unwrap(
);
    v
}

```

Podemos ver que a função lê de um `str`, através da função `from_str`, que recebe como argumento a leitura gerada na função anterior, `read_fake`. Como `read_fake` retorna uma `String` e `from_str` espera um `&str`, devemos utilizar a função `as_str()` para converter a `String` em `&str`. o `v` ao final é o valor de retorno da `Card`. A struct `Card` será mostrada logo. Para convertermos o valor de retorno em JSON, devemos utilizar a macro `json!()`, incluindo macros da crate `serde_json`, `# [macro_use] extern crate serde_json;`. O valor de retorno muda para um `serde_json::Value`:

```

use serde_json::Value;

pub fn fake_json() -> Value {
    let v: Card = from_str(read_fake().unwrap().as_str()).unwrap(
);
    json!(v)
}

```

Agora, para podermos serializar e desserializar a struct `Card` precisamos de duas anotações do `serde_derive`: a `Serialize` e a `Deserialize`. Junto a isso, é sempre interessante termos a

capacidade de entender e copiar nossa struct. Para isso, adicionamos as anotações `Debug` e `Clone`. A estrutura da struct `Card` se assemelha à estrutura do JSON, na qual os campos que podem ser nulos ou podem não existir devem aparecer como `Option<_>`.

```
#[derive(Serialize, Deserialize, Debug, Clone)]
struct Card {
    cards: Vec<CardBody>,
}

#[derive(Serialize, Deserialize, Debug, Clone)]
struct CardBody {
    name: String,
    mana_cost: Option<String>,
    cmc: i32,
    colors: Vec<String>,
    color_identity: Option<Vec<String>>,
    types: Vec<String>,
    subtypes: Option<Vec<String>>,
    rarity: String,
    set: String,
    set_name: Option<String>,
    text: String,
    artist: String,
    number: String,
    power: Option<String>,
    toughness: Option<String>,
    layout: String,
    multiverseid: i32,
    image_url: Option<String>,
    rulings: Option<Vec<Rulings>>,
    printings: Vec<String>,
    original_text: Option<String>,
    original_type: Option<String>,
    id: String
}

#[derive(Serialize, Deserialize, Debug, Clone)]
struct Rulings {
    date: String,
```

```

    text: String,
}

```

Agora, voltando um pouco a nos preocupar com o Hyper, precisamos que a função `fake_json` receba um `Request` e retorne um `Response` que o Hyper possa entender. Isso pode ser feito adicionando `Response` e `Body` à função:

```

use hyper::{Request, Body, Response};

fn fake_json(_req: Request<Body>) -> Response<Body> {
    let v: Card = serde_json::from_str(read_fake().unwrap().as_str())
        .unwrap();
    Response::new(Body::from(json!(v.clone()).to_string()))
}

```

Para servirmos esse `Response` com o corpo que queremos, basta utilizar a função do Hyper `service_fn_ok`:

```

use hyper::rt::Future;
use hyper::{Server};
use hyper::service::{service_fn_ok};

fn main() {
    let addr = ([127, 0, 0, 1], 3000).into();

    let ping_svc = move || {
        service_fn_ok(fake_json)
    };

    let server = Server::bind(&addr)
        .serve(ping_svc)
        .map_err(|e| eprintln!("server error: {}", e));

    hyper::rt::run(server);
}

```

Deixando nosso servidor responder via GraphQL

Primeira coisa de que precisamos é adicionar a crate que disponibiliza GraphQL em Rust. Ela é chamada de **Juniper**

(<https://crates.io/crates/juniper>). Além disso, precisamos adicionar sua implementação para Hyper, `juniper_hyper`, e mais algumas dependências, conforme a seguir:

```
[dependencies]
hyper = "*"
serde = "*"
serde_json = "*"
serde_derive = "*"
gqllog = "*"
futures = "*"
futures-cpupool = "*"
juniper = "*"
juniper_hyper = "*"
pretty_env_logger = "*"
```

Tendo disponíveis as informações fornecidas por nosso contexto (`body.json`), ou "contrato", no caso de uma API, podemos criar o *schema* da query do GraphQL, que representa o esquema que estará disponível para consulta via requisição. Ao ler o arquivo `body.json`, percebemos que ele responde um JSON com o parâmetro "cards", que será representado pelo parâmetro `cards: Vec<CardBody>`, na `struct Card`. O parâmetro `cards` da `struct Card` é um vetor de "corpos" para cada `card`, como definidos na `struct CardBody`. Com esta estrutura serializada, podemos fazer o GraphQL entender os campos que ela representa, adicionando a declaração de uso do macro da crate Juniper, `#[macro_use] extern crate juniper;`, e adicionando a derivação `GraphQLObject` às structs `Card`, `CardBody` e `Rulings`:

```
#[derive(Serialize, Deserialize, Debug, Clone, GraphQLObject)]
#[graphql(description="Card Fields")]
struct CardBody {
    name: String,
    manaCost: Option<String>,
    ...
}
```

Antes de entrarmos no Hyper, podemos definir uma primeira query para o GraphQL. Essa primeira query será bem simples, e consistirá de retornar todas as cards presentes e com todos os campos serializados, conforme a imagem a seguir:



Figura 15.1: Query allCards, com resultados e campos disponíveis

Para implementarmos a query GraphQL precisaremos de uma `struct Query` e de implementar suas subqueries através da macro `graphql_object!`. Esta macro implementa a `trait GraphQLType`. Ao utilizá-la, em vez de implementar o tipo

manualmente, obtemos segurança de tipos e reduzimos declarações repetitivas. Além disso, a macro nos dá o poder de autorresolver os campos definidos como `field` através da sua função interna `resolve_field`.

A struct `Query` poderia ser uma struct com campos e estados, especialmente se fôssemos utilizar a funcionalidade de *mutation*. O exemplo a seguir demonstra como declarar uma struct de `Query` com estado interno:

```
//Exemplo da documentação
#[macro_use] extern crate juniper;
struct User { id: String, name: String, group_ids: Vec<String> }
graphql_object!(User: () |&self| {
    field id() -> &String {
        &self.id
    }
    field name() -> &String {
        &self.name
    }
    field member_of_group(group_id: String) -> bool {
        self.group_ids.iter().any(|gid| gid == &group_id)
    }
});
```

Neste exemplo, a struct `User` possui 3 campos, e quando acessamos as queries de `id` ou de `name`, vemos a palavra `self` antes, que retorna o estado contido na struct. Na última query, `member_of_group`, vemos se a `String` recebida está contida dentro do vetor iterável `self.group_ids.iter()` através da função `any`.

Como nossa `Query` não terá estados, não precisamos nos

preocupar com campos internos da struct. Assim, podemos partir diretamente para a implementação da query `allCards` :

```
use juniper::{FieldResult};

struct Query;

graphql_object!(Query: Card |&self| {
    field allCards(&executor) -> FieldResult<Vec<CardBody>> {
        Ok(executor.context().cards.to_owned())
    }
});
```

A primeira diferença que vemos é que no nosso exemplo definimos o tipo da `Query` como `Card` . A segunda é que nossas funções recebem o argumento `&executor` , responsável por acessar o contexto que será passado para o GraphQL. Veremos isso logo adiante. Agora vamos ao que acontece nesta query.

Quando a query contendo `allCards` é feita, entendemos que o tipo do contexto é do tipo `Card` , que será o conjunto de campos que poderemos consultar no contexto do GraphQL. Quanto à resposta, imagine que não queremos que todo o contrato de `CardBody` seja serializado via GraphQL. Para isso, poderíamos declarar uma struct de retorno `CardQL` que contém somente os campos que queremos resolver. Algo neste formato:

```
#[derive(Serialize, Deserialize, Debug, Clone, GraphQLObject)]
#[graphql(description="Card Query Fields")]
struct CardQL {
    name: String,
    mana_cost: Option<String>,
    cmc: i32,
    colors: Vec<String>,
    types: Vec<String>,
    subtypes: Option<Vec<String>>,
    rarity: String,
    number: String,
    power: Option<String>,
```

```

    toughness: Option<String>,
}

```

Neste caso, `CardQL` conteria os campos possíveis de consultar.

Além disso, temos um tipo de resposta específico para o campo, que foi declarado no `use juniper::{FieldResult}`. Este campo é basicamente uma struct em volta de um `Result` contendo um `FieldError`. No caso do nosso exemplo, utilizamos o `<Vec<CardBody>>` como resposta, mas poderia ter sido o `<Vec<CardQL>>`, conforme discutimos anteriormente. Por último, temos o `Ok()`, que retorna todas as cards.

Estamos falando muito de contexto, mas não explicamos bem o que é isso. Para isso, vou mostrar como `juniper_hyper` integra `Juniper` com `Hyper`. Agora precisamos de algumas coisas mais, como o tamanho das threads de CPU disponíveis na `pool` que vamos utilizar. Precisamos do nóculo de origem (`root_node`) e do contexto. Dentro de `main` adicionaremos alguns destes parâmetros, como o exemplo a seguir:

```

use futures_cpupool::CpuPool;
use juniper::RootNode;
use juniper::{EmptyMutation};
use std::sync::Arc;

fn main() {
    pretty_env_logger::init();
    let pool = CpuPool::new(4);
    let addr = ([127, 0, 0, 1], 3000).into();

    let root_node = Arc::new(RootNode::new(Arc::new(Query), EmptyMutation::<Card>::new()));
    ...
}

```

O valor `pool` consiste do número de threads que vamos

consumir e o `root_node` é do tipo *atomic reference count* e conterá um `RootNode` com a `Query` e uma `EmptyMutation`, que representa uma mutação vazia, pois nosso código não alterará o estado do contexto. Agora, antes de entrar no serviço `Hyper` podemos detalhar o último parâmetro que usaremos para construir o GraphQL, o `Context`, ou contexto. Nosso contexto do tipo `Query` está definido como `Card`. Então, devemos ler o arquivo `body.json`, como fazemos com a função `fake_read`, mas vamos alterar a função `fake_json` para algo que retorne um contexto falso, e para que seu tipo seja `Card`, como a função `fake_ctx`:

```
fn fake_ctx() -> Card {
    let v: Card = serde_json::from_str(read_fake().unwrap().as_str())
        .unwrap();

    v
}
```

Utilizamos o `serde_json` para serializar o resultado de `read_fake` em uma struct `Card`. A última edição para termos o GraphQL funcionando é substituir o serviço `ping_svc` por um serviço que implemente os tipos do GraphQL. Faremos isso dentro da closure `service` declarada no bloco a seguir:

```
#[macro_use] extern crate serde_json;
#[macro_use] extern crate serde_derive;
#[macro_use] extern crate juniper;
extern crate serde;
extern crate hyper;
extern crate juniper_hyper;
extern crate futures;
extern crate futures_cpupool;
extern crate pretty_env_logger;

use futures::future;
```

```

use futures_cpupool::CpuPool;
use hyper::rt::{Future};
use hyper::{Body, Method, Response, Server, StatusCode};
use hyper::service::{service_fn};
use juniper::{FieldResult, EmptyMutation};
use juniper::RootNode;
use std::sync::Arc;

fn main() {
    pretty_env_logger::init();
    let pool = CpuPool::new(4);
    let addr = ([127, 0, 0, 1], 3000).into();

    let root_node = Arc::new(RootNode::new(Arc::new(Query), EmptyMutation::<Card>::new()));

    let service = move || {...}

    let server = Server::bind(&addr)
        .serve(service)
        .map_err(|e| eprintln!("server error: {}", e));

    hyper::rt::run(server);
}

```

A implementação do service fica assim:

```

let service = move || {
    let root_node = root_node.clone();
    let pool = pool.clone();
    let ctx = Arc::new(fake_ctx());
    service_fn(move |req| -> Box<Future<Item = _, Error = _> + Send> {
        let root_node = root_node.clone();
        let ctx = ctx.clone();
        let pool = pool.clone();
        match (req.method(), req.uri().path()) {
            (&Method::GET, "/") => Box::new(juniper_hyper::graphql("/graphql")),
            (&Method::GET, "/graphql") => Box::new(juniper_hyper::graphql(pool, root_node, ctx, req)),
            (&Method::POST, "/graphql") => {
                Box::new(juniper_hyper::graphql(pool, root_node, ctx, req))
            }
        }
    })
}

```

```

    }
    _ => {
        let mut response = Response::new(Body::empty());
        *response.status_mut() = StatusCode::NOT_FOUND;
        Box::new(future::ok(response))
    }
}
}))
};

```

Observe as duas palavras reservadas `move`. A função de `move` é absorver o contexto dos valores externos através de uma cópia para dentro de seu contexto e tomando *ownership* da cópia. Depois disso, perceba uma grande quantidade de `clone()`, o que se deve à nossa necessidade de criar instâncias novas para os valores que temos, com o objetivo de os passarmos para outras funções. Além disso, a função `service_fn` é responsável por criar este serviço GraphQL, respondendo uma `Future` dentro de uma `Box`. Essa `Box` é criada através de um *pattern matching* de dois argumentos presentes na requisição `|req|`: `req.method()`, `req.uri().path()`, o método da requisição e a URL da requisição, respectivamente. O padrão `(MÉTODO, "URL")` funciona como uma desestruturação no formato de uma tupla, na qual os dois argumentos devem ser satisfeitos.

Refatorando para utilizar módulos

Primeiramente, o que vamos fazer são as declarações dos módulos no arquivo `main`, logo após as declarações das crates e a criação de um arquivo `lib.rs`, que conterà o contexto dos módulos a serem utilizados. As anotações para consumo de macros devem ser declaradas na raiz do módulo, `lib.rs`, pela anotação `#[macro_use]`. E os módulos para serem consumidos fora da `lib` devem ser declarados com a palavra-chave `pub`

antes, conforme a seguir.

Declaração dos módulos e consumo:

```
mod client;
mod schema;

...
use client::fake_ctx;
use schema::{Card, Query};
```

Lib.rs com as crates consumidas e módulos públicos declarados:

```
#[macro_use] extern crate serde_derive;
#[macro_use] extern crate juniper;
extern crate serde_json;
extern crate serde;
extern crate rayon;

pub mod client;
pub mod schema;
```

O módulo `client` é basicamente igual a toda a parte relacionada ao consumo do arquivo `body.json`, e ficará da seguinte maneira:

```
use std::fs::File;
use std::io::prelude::*;
use std::io::Result;
use serde_json::from_str;
use super::schema::Card;

fn read_fake() -> Result<String> {
    let mut file = File::open("./body.json")?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    Ok(contents)
}

pub fn fake_ctx() -> Card {
    let v: Card = from_str(read_fake().unwrap().as_str()).unwrap(
```

```
);
    v
}
```

Já para o módulo `schema` vamos extrair tudo o que se relacionava à parte de schemas e queries do GraphQL, mas adicionar uma nova query, que chamaremos de `card_by_name`. A estrutura do módulo sem a nova query está conforme a seguir:

```
use juniper::{FieldResult};
use rayon::prelude::*;

pub struct Query;

graphql_object!(Query: Card |&self| {
    field all_cards(&executor) -> FieldResult<Vec<CardBody>> {
        Ok(executor.context().cards.to_owned())
    }
    ...
});

#[derive(Serialize, Deserialize, Debug, Clone, GraphQLObject)]
#[graphql(description="Cards Vector")]
pub struct Card {
    pub cards: Vec<CardBody>,
}

#[derive(Serialize, Deserialize, Debug, Clone, GraphQLObject)]
#[graphql(description="Card Fields")]
#[allow(non_snake_case)]
pub struct CardBody {
    ...
}

#[derive(Serialize, Deserialize, Debug, Clone, GraphQLObject)]
#[graphql(description="Card Rulings")]
struct Rulings {
    ...
}
```

No arquivo `lib.rs`, possuímos a declaração de uma crate

através de `extern crate rayon`, e no módulo `schema` possuímos a seguinte declaração `use rayon::prelude::*`; `.` A `crate Rayon` serve para fazermos processamento em paralelo de *adapters* e *consumers*.

Neste momento, o que precisamos saber sobre a `crate Rayon` é que basta substituir o iterador comum por um iterador em paralelo. Essa substituição já nos permitirá fazer o processamento em paralelo dos dados passados para o iterador. A tabela a seguir mostra como são renomeados estes iteradores:

Sequencial	Paralelo
<code>.iter()</code>	<code>.par_iter()</code>
<code>.iter_mut()</code>	<code>.par_iter_mut()</code>
<code>.into_iter()</code>	<code>.into_par_iter()</code>

Após declararmos os iteradores em paralelo, podemos utilizar *consumers* e *adapters* da mesma forma explicada no capítulo 8. *Iterators, adapters e consumers*. Assim, agora queremos declarar a query `card_by_name`, e pesquisar em todas as cards de forma paralela. Para isso, na macro `graphql_object!` declaramos um novo `field` com nome de `card_by_name`:

```
graphql_object!(Query: Card |&self| {
    ...

    field card_by_name(&executor, name: String) -> FieldResult<Vec<CardBody>> {
        let cards = executor.context().cards.clone();
        Ok(cards.into_par_iter().filter(|card| card.name.contains(
            name.as_str())).collect:::<Vec<CardBody>>())
    }
});
```

O `field card_by_name` recebe uma `String name` e extrai

do executor o contexto de cards, retornando um `Field Result` com todas as cards que possuem no nome a String passada, `name`. O processamento disto é feito através do iterador `into_par_iter` e do adapter `filter`, que verifica se o nome da card corrente possui a `&str name`, para depois consumir todas as cards que retornaram verdadeiro da iteração.

Um pouco sobre Rayon

Rayon é uma crate cujo objetivo é transformar cálculos sequenciais em cálculos paralelos. Para a utilizarmos, basta adicioná-la ao `cargo.toml`, declará-la por meio do `extern crate rayon` e consumi-la com `use rayon::prelude::*`; . A única mudança efetiva que será feita no seu código é a demonstrada na tabela de *Sequencial e Paralelo*, que acabamos de ver. Rayon possui algumas outras funções interessantes, como as da família de `par_sort`, que fazem processamento em paralelo das mais diferentes formas:

```
let mut v = [-5, 4, 1, -3, 2];

v.par_sort();
assert_eq!(v, [-5, -3, 1, 2, 4]);
```

E a função `join`, que, quando associada à `struct ThreadPool`, nos permite chamar várias closures em paralelo e definir a quantidade de threads que vamos utilizar. Para isso, basta fazer a seguinte declaração de `pool`: `let pool = rayon::ThreadPoolBuilder::new().num_threads(8).build().unwrap()`; . Com isso, criamos uma pool de 8 threads para resolver as funções que serão chamadas em paralelo com `join`. No exemplo a seguir declaramos a função `fib`, que retorna `1` ou `0` para valores de `n` iguais a 1 ou 0. Caso os valores sejam

maiores, dispara duas funções `fib` dentro de `rayon::join`. O resultado destas funções é consumido através de uma desestruturação em tupla chamada de `let (a, b)`, que é retornada como uma soma. `install()` executa uma closure em uma das threads de `ThreadPool`. Além disso, quaisquer outras operações de `rayon` chamadas dentro de `install()` também serão executadas no contexto do `ThreadPool`.

```
fn main() {
    let pool = rayon::ThreadPoolBuilder::new().num_threads(8).build
    ().unwrap();
    let n = pool.install(|| fib(20));
    println!("{}", n);
}

fn fib(n: usize) -> usize {
    if n == 0 || n == 1 {
        return n;
    }
    let (a, b) = rayon::join(|| fib(n - 1), || fib(n - 2));
    return a + b;
}
```

Pattern matching *versus* If Let

A próxima etapa será criar mais um `field` na macro `graphql_object!`. Ele será responsável por buscar cards que contenham, ou uma cor de *mana* específica, ou um conjunto de cores de *manas*. Nossa declaração da query ficará assim:

```
field mana_type_cards(&executor, color: Option<String>, colors: Option<Vec<String>>)
    -> FieldResult<Vec<CardBody>> {
    let cards = executor.context().cards.clone();
    ...
}
```

Pattern matching

Como recebemos dois argumentos do tipo `Option`, devemos atuar de forma a identificar quais são as situações prioritárias. Da forma como eu implementei, pensei primeiro na situação de encontrar uma cor específica, pois acredito que seja mais relevante e versátil. Para resolver isso, fiz um pattern matching com `color`:

```
field mana_type_cards(&executor, color: Option<String>, colors: Option<Vec<String>>)
    -> FieldResult<Vec<CardBody>> {
    let cards = executor.context().cards.clone();
    match color {
        Some(c) => Ok(cards.into_par_iter()
            .filter(|card| card.colors.contains(&c.to_owned()))
            .collect::<Vec<CardBody>>()),
        None => Ok(vec![]),
    }
}
```

O padrão criado verifica primeiro se `color` é um `Option` contendo um valor `Some` e retorna um `FieldResult` que consiste da iteração em paralelo de `cards`. O filtro verifica se o campo `colors` de `card` possui a `color` definida como argumento, retornando um vetor com todos os resultados positivos do `filter`. O caso de `None` retorna um vetor vazio. Mas e quanto ao argumento `colors`? Não podemos simplesmente fazer um novo `match` depois do primeiro? Não seria possível, pois seria um código inatingível, já que a função está retornando o resultado do `match` já existente. Para evitar isso e para termos um código atingível, precisamos inserir um novo `match` dentro da cláusula `None`. Faremos isso da seguinte forma:

```
field mana_type_cards(&executor, color: Option<String>, colors: Option<Vec<String>>)
    -> FieldResult<Vec<CardBody>> {
    let cards = executor.context().cards.clone();
```

```

match color {
  Some(c) => Ok(cards.into_par_iter()
    .filter(|card| card.colors.contains(&c.to_
wned()))
    .collect:::<Vec<CardBody>>()),
  None => match colors {
    Some(cs) => Ok(cards.into_par_iter()
    .filter(|card|
ue
      cs.iter().fold(true, |value, x| val
wned()))
      && card.colors.contains(&x.to_
wned()))
    .collect:::<Vec<CardBody>>()),
    None => Ok(vec![]),
  }
}
}
}

```

Podemos ver que o resultado vazio `Ok(vec![])` foi passado para o `None` interno. No cenário atual, temos o seguinte fluxo:

1. No primeiro `match`, caso uma cor de *mana* única, `color`, resulte no valor `Some(c)`, retornaremos um `Result` com o vetor de `cards` que contenham as cores `color`.
2. Caso `color` seja um `None`, entraremos em um novo `match` para `colors`.
3. Caso `colors` retorne `Some(cs)` retornaremos um `Result` com o vetor de `cards` que contenham as cores definidas em `colors`.
4. Caso o `match` de `colors` seja um `None`, retornaremos um `Result` com um vetor vazio.

Nossa estratégia para o caso 3, ou `Some(cs)`, é aplicar um `filter` em cada `card` e retornar um vetor com todas as `cards` que possuem as cores de `colors`. Para termos uma closure que retorna valores verdadeiros no `filter` aplicamos um `iter` a `cs` seguido por um `fold`, `fold(true, |value, x| value &&`

```
card.colors.contains(&x.to_owned()))).
```

Este `fold` tem um valor inicial, `value == true`, e a cor corrente da iteração, `x`. Através da função `contains`, verifica-se se ela está presente nas cores de `card`, `card.colors.contains(&x.to_owned())`. Os resultados da cor corrente e de `value` são testados em um `and`, ou `&&` (e lógico), e retornados ao `filter` como um argumento booleano. Agora vamos entender como a lógica do *pattern matching* ficaria com `if lets`.

If Let

Assim como com os matches anteriores, temos duas situações diferentes e um resultado padrão caso os dois matches falhem. Para isso, utilizaremos a estrutura `if let Some(c) = color` seguido por `else if let Some(cs) = colors` seguido por `else`. No primeiro `if let` verificamos se `c` é, de fato, um valor `Some` de `color`. Caso for verdadeiro, retornamos o caso 1 do *pattern matching*. Se `color` não retornar `Some(c)`, cairemos no `else`, que verifica se `cs` é um valor do tipo `Some` de `colors`, e caso isso seja verdadeiro retornamos o caso 3 do *pattern matching*. Se nenhum dos dois casos for verdadeiro, retornamos o caso `else` com `Ok(vec![])`. A estrutura é a seguinte:

```
field mana_type_cards(&executor, color: Option<String>, colors: Option<Vec<String>>)
-> FieldResult<Vec<CardBody>> {
    let cards = executor.context().cards.clone();
    if let Some(c) = color {
        Ok(cards.into_par_iter()
            .filter(|card| card.colors.contains(&c.to_owned()))
            .collect::<Vec<CardBody>>())
    } else if let Some(cs) = colors {
```

```

Ok(cards.into_par_iter()
    .filter(|card|
        cs.iter().fold(true, |value, x| value
            && card.colors.contains(&x.to_owned()))
    .collect::

```

Ao meu ver, temos uma estrutura de *pattern matching* mais adequada com `if let else` do que com `match`.

Pensando em erros controlados

Já temos bastantes queries em `graphql_object!`, mas temos pouco controle caso algo dê errado. É muito comum recebermos valores inválidos, que podem interromper o funcionamento da API, ou valores maliciosos que servem para explorar alguma falha de segurança. Por isso, devemos nos preparar e criar estruturas mais controladas e seguras. Um exemplo disso seria criar a constante `COLORS`, que contém as cores possíveis para as *manas*.

Para isso, basta fazermos a seguinte declaração em `schema.rs`:

```

const COLORS: [&str; 5] =
["White", "Red", "Black", "Green", "Blue"];

```

Agora podemos adicionar um `if` que testa se o argumento `color` está contido na constante `COLORS`, por meio de:

```

field mana_type_cards(&executor, color: Option<String>, colors: Option<Vec<String>>)
    -> FieldResult<Vec<CardBody>> {
    if !COLORS.contains(&color.clone().unwrap_or(String::from("WRONG")).as_str()) {
        return Ok(vec![]);
    }
    ...
}

```

Podemos ver que, além do teste para ver se `COLORS` contém `&color`, ao extrairmos o valor de `color` criamos uma cláusula que retorna `"WRONG"` através de `unwrap_or`, que representa o caso de `color` ser do tipo `Option.None`.

Agora, e se decidirmos criar outra query para testar a raridade das cards? Podemos criar uma constante que contenha todas as possíveis raridades, como `const RARITY: [&str; 3] = ["Common", "Uncommon", "Rare"];`. Com ela criada, podemos criar um novo `field` que busca cards por raridade, com a query `cards_by_rarity`:

```
field cards_by_rarity(&executor, rarity: String) -> FieldResult<Vec<CardBody>> {
    if !RARITY.contains(&rarity.as_str()) {
        return Ok(vec![]);
    }
    ...
}
```

Para o valor de retorno, precisamos somente de um filtro que verifica se a raridade da card é igual ao argumento `rarity`:

```
field cards_by_rarity(&executor, rarity: String) -> FieldResult<Vec<CardBody>> {
    if !RARITY.contains(&rarity.as_str()) {
        return Ok(vec![]);
    }
    let cards = executor.context().cards.clone();
    Ok(cards.into_par_iter()
        .filter(|card| card.rarity == rarity)
        .collect:::<Vec<CardBody>>())
}
```

Nosso próximo passo é, em vez de retornar `Ok(vec![])`, retornarmos um erro para o GraphQL, conforme o que podemos ver na imagem a seguir para a query descrita:

```
{
  cardsByRarity(rarity: "Teste"){
    name
  }
}
```

Erro para raridade não encontrada



Figura 15.2: Erro para raridade não encontrada

Para que essa funcionalidade se concretize, precisamos mudar o tipo de resultado de `FieldResult<Vec<CardBody>>` para `Result<Vec<CardBody>, InputError>`, em que `InputError` será definido no módulo `gqlerror`. Para começarmos,

precisamos declarar o módulo e consumir o erro que estamos criando:

```
//main.rs
...
mod gqlerror;

...
use super::gqlerror::{InputError};
```

Ainda devemos adicionar a declaração do módulo no `lib.rs` :
`pub mod gqlerror;` . Com isso pronto, podemos começar a entender a implementação de `gqlerror.rs` :

```
//gqlerror.rs
use juniper::{FieldError, IntoFieldError};

pub enum InputError {
    ColorValidationError,
    RarityValidationError,
}

impl IntoFieldError for InputError {
    fn into_field_error(self) -> FieldError {
        match self {
            InputError::ColorValidationError => FieldError::new(
                "The input color is not a possible value",
                graphql_value!({
                    "type": "COLOR NOT FOUND"
                }),
            ),
            InputError::RarityValidationError => FieldError::new(
                "The input rarity is not a possible value",
                graphql_value!({
                    "type": "RARITY NOT FOUND"
                }),
            ),
        }
    }
}
```

O primeiro a se fazer é consumir os tipos de dados que

precisamos utilizar para criar nossos erros customizados por meio de `FieldError`, que representa um tipo `Err` no campo `field` do GraphQL, e a `trait IntoFieldError`, que nos permite implementar o que este erro responderá ao GraphQL. Além disso, declaramos o `enum` público `InputError`, que contém dois estados: `ColorValidationError` e `RarityValidationError`.

Agora podemos partir para a implementação da `trait IntoFieldError`. A função que devemos implementar se chama `into_field_error`, e possui um tipo de retorno `FieldError`, `fn into_field_error(self) -> FieldError`. Com um `match`, podemos determinar qual o tipo de erro que obtivemos e criar um novo `FieldError`:

```
FieldError::new(
    "The input rarity is not a possible value",
    graphql_value!({
        "type": "RARITY NOT FOUND"
    }))
```

Por final, devemos mudar o valor de retorno contido dentro dos `ifs` de validação:

```
graphql_object!(Query: Card |&self| {
    ...
    field cards_by_rarity(&executor, rarity: String)
        -> Result<Vec<CardBody>, InputError> {
            if !RARITY.contains(&rarity.as_str()) {
                return Err(InputError::RarityValidationError);
            }
        }
    ...
}

    field mana_type_cards(&executor, color: Option<String>, colors:
Option<Vec<String>>)
        -> Result<Vec<CardBody>, InputError> {
            if !COLORS.contains(&color.clone().unwrap_or(String::from("
WRONG"))).as_str()) {
```

```

        return Err(ValidationError::ColorValidationError);
    }
    ...
}
})

```

Última função

O último caso que quero apresentar com este exemplo é utilizando um `filter` com contexto interno, algo como uma lógica interna que vá além de uma closure simples, e posteriormente aplicando um *pattern matching* a ele. Faremos isso no último `field`, `cards_by_power_and_toughness`, do nosso `graphql_object!`. Em um primeiro momento, precisamos retornar todas as `cards` que possuem poder (*power*) e resistência (*toughness*) maiores que as enviadas como argumento. Caso o valor da `card` não faça sentido, como em `cards` de terreno, que não costumam ter poder e resistência, retornamos `-1`, para que o resultado da lógica `>=` seja falso e, portanto, a `card` seja filtrada da iteração. Veja o código a seguir:

```

field cards_by_power_and_toughness(&executor, min_power: i32, min_toughness: i32)
-> Result<Vec<CardBody>, InputError> {
    let cards = executor.context().cards.clone();
    Ok(cards.into_par_iter()
        .filter(|card| {
            let card_power = card.power.clone();
            let card_tough = card.toughness.clone();
            card_power.unwrap_or("-1".to_string()).parse:::<i32>()
            .unwrap_or(-1) >= min_power
            && card_tough.unwrap_or("-1".to_string()).parse:::<i32>()
            >= min_toughness})
        .collect:::<Vec<CardBody>>())
}

```

O que temos de dentro do `filter` é um pouco diferente de

uma closure comum, pois através do uso de chaves conseguimos incluir um contexto. Digamos que o código ficaria extremamente confuso se fosse feito da seguinte maneira:

```
card.power.clone().unwrap_or("-1".to_string()).parse::i32>().unwrap_or(-1) >= min_power
&& card.toughness.clone().unwrap_or("-1".to_string()).parse::i32>().unwrap_or(-1) >= min_toughness})
```

Assim, se extrairmos o contexto de `card.<atributo>.clone()` para um valor como `let card_<atributo> = card.<atributo>.clone()`, teremos um contexto melhor para compararmos. Além disso, estamos utilizando dois `unwrap_or` em contextos diferentes. O primeiro se refere ao fato de que o atributo pode não existir e precisamos passar algum valor coerente para a função `parse::i32>()`, na qual encontramos o segundo `unwrap_or`, que retorna um valor negativo. Creio que seja melhor não retornar cards caso falhe algo, do que dar um `panic` desconhecido, já que um `panic` poderia informar ao usuário dados sensíveis.

Nosso último passo agora é salvar esse valor para podermos aplicar mais um filtro adicional, como custo máximo de *mana*. Para isso, vamos salvar o valor de `Ok` em um campo chamado `cards`. O custo de *mana*, `max_mana_cost`, será um valor opcional, conforme o código a seguir:

```
field cards_by_power_and_toughness(&executor, min_power: i32, min_toughness: i32, max_mana_cost: Option<i32>)
-> Result<Vec<CardBody>, InputError> {
  let cards = executor.context().cards.clone()
    .into_par_iter()
    .filter(|card| {
      let inner_power = card.power.clone();
      let inner_tough = card.toughness.clone();
      inner_power.unwrap_or("-1".to_string()).parse::i32>
```

```

).unwrap_or(-1) >= min_power
    && inner_tough.unwrap_or("-1".to_string()).parse::i32:
>().unwrap_or(-1) >= min_toughness})
    .collect::

```

Para extrairmos o valor de `max_mana_cost`, vamos utilizar a estratégia do `if let`. Assim, no caso de `if let` retornar um valor `Some`, aplicamos um novo filtro; caso contrário, retornamos `Ok(cards)` na cláusula `else`. O novo filtro consiste basicamente em filtrar as `cards` que tenham um custo de *mana* maiores que `max_mana_cost`:

```

field cards_by_power_and_toughness(&executor, min_power: i32, min
_toughness: i32, , max_mana_cost: Option<i32>)
-> Result<Vec<CardBody>, InputError> {
    let cards = executor.context().cards.clone()
        .into_par_iter()
        .filter(|card| {
            let inner_power = card.power.clone();
            let inner_tough = card.toughness.clone();
            inner_power.unwrap_or("-1".to_string()).parse::i32>(
).unwrap_or(-1) >= min_power
                && inner_tough.unwrap_or("-1".to_string()).parse::i32:
>().unwrap_or(-1) >= min_toughness})
            .collect::

```

Agora nosso servidor GraphQL retorna várias queries interessantes com diferentes estratégias de Programação Funcional para servir os dados de `cards` de Magic. No próximo capítulo,

apresentarei uma última implementação de um servidor Web, dessa vez de mais baixo nível e com maior ênfase nas questões de assincronicidade.

TOKIO E A ASSINCRONICIDADE

Resumidamente, Tokio é o ponto central para tudo que se relaciona com assincronicidade em Rust. O núcleo da Tokio baseia-se em `futures` que são a base Rust para a computação assíncrona, e ela nos disponibiliza diversas crates para nos auxiliar em aspectos específicos.

A Tokio possui algumas características interessantes: é rápida, pois possui baixo custo para abstrações, o que permite uma performance incrível; tem alta produtividade, pois é bastante trivial de implementar; confiabilidade, já que garante alta segurança no uso de threads; e por último, é escalável, conseguindo suportar grande pressão no gerenciamento de threads.

Algumas das principais bibliotecas associadas à Tokio são:

- `tokio-proto` : camada mais simples para criar servidores e clientes. O trabalho é principalmente lidar com a serialização de mensagens, para o `proto` cuidar do resto. A biblioteca inclui uma grande variedade de sabores de protocolos, como streamings e protocolos *multiplex*.
- `tokio-core` : tem como foco escrever código assíncrono

de baixo nível, permitindo lidar diretamente com I/O de estruturas de dados e eventos, mas com uma forma ergonômica de alto nível para lidar com os códigos. Core será importante quando precisarmos de total controle do interior do servidor ou do cliente. Felizmente, nosso exemplo de `tokio-proto` é detalhado o suficiente para abranger o `tokio-core`.

- `futures` : provê abstrações como `futures`, `streams` e `sinks`, usadas por toda Tokio.

MULTIPLEX

Uma conexão de *socket multiplex* é aquela que permite que muitos `requests` simultâneos sejam feitos, com as `responses` voltando na ordem em que estão prontas em vez de na ordem solicitada. A multiplexação permite que um servidor comece a processar `request` assim que ele for recebido, e permite responder ao cliente assim que for processado.

16.1 TOKIO-PROTO E TOKIO-CORE

Vamos começar com a Tokio usando a biblioteca padrão, a `proto`. Faremos o exemplo clássico de um servidor que retorna o que foi passado. Para iniciar, rodamos o comando `cargo new servidor-proto --bin` e adicionamos as seguintes dependências ao `cargo.toml`, em que usaremos versões específicas:

```
[dependencies]
```

```
bytes = "0.4"
futures = "0.1"
tokio-io = "0.1"
tokio-core = "0.1"
tokio-proto = "0.1"
tokio-service = "0.1"
```

Além disso, precisamos disponibilizar nossas dependências no `main.rs` :

```
extern crate bytes;
extern crate futures;
extern crate tokio_io;
extern crate tokio_proto;
extern crate tokio_service;
```

Um servidor `proto` é formado por 3 partes principais:

1. Um *transport*, responsável pela serialização dos `requests` e `responses` para o `socket`.
2. A especificação do protocolo, que elabora um *codec* com informações básicas sobre o protocolo (como *multiplexed* ou *streaming*).
3. Um serviço que é responsável por determinar como uma `response` é gerada para um determinado `request` . Neste caso, o serviço é uma função assíncrona.

Implementando o codec

O primeiro passo, e o mais simples, é implementar um `codec` de linha que incluiremos em um arquivo `lib.rs` , com o nome de `pub mod codec` , e estará armazenado no arquivo `codec.rs` sob o nome de `pub struct LineCodec` . É importante lembrar de incluir o módulo do `codec` no `main.rs` com `mod codec` .

Para começar, devemos implementar as `traits Encode` e

Decode para nossa struct `LineCodec`. Assim, definimos as implementações de `Decoder` e de `Encoder` da crate `tokio_io`, com as definições de `Item` e `Erro`:

Codecs são programas responsáveis por codificar e decodificar streams de dados.

```
use std::io;
use std::str;
use tokio_io::codec::{Encoder, Decoder};

pub struct LineCodec;

impl Decoder for LineCodec {
    type Item = String;
    type Error = io::Error;

    // ...
}

impl Encoder for LineCodec {
    type Item = String;
    type Error = io::Error;

    // ...
}
```

Usaremos `String` para a representação das linhas, o que significa que nosso `Encoder` e `Decoder` precisarão implementar a codificação UTF-8 para o protocolo de linha. A forma mais fácil de se começar é implementando a função `encode`:

```
use bytes::BytesMut;

impl Encoder for LineCodec {
    type Item = String;
    type Error = io::Error;
```

```

    fn encode(&mut self, msg: String, buf: &mut BytesMut) -> io::
Result<()> {
        buf.extend(msg.as_bytes());
        buf.extend(b"\n");
        Ok(())
    }
}

```

O código é simples, mas precisa de algumas explicações. A primeira delas é o `BytesMut`, que nos provê uma forma simples, mas eficiente, de gerenciar buffers – algo como um `Arc<[u8]>`. O método `extend` estende uma coleção de acordo com o tipo a ser iterado (no nosso caso, bytes). O `Ok(())` refere-se ao tipo de retorno `io::Result<()>`. Agora vamos implementar o `decode`:

```

impl Decoder for LineCodec {
    type Item = String;
    type Error = io::Error;

    fn decode(&mut self, buf: &mut BytesMut) -> io::Result<Option
<String>> {
        if let Some(i) = buf.iter().position(|&b| b == b'\n') {
            let line = buf.split_to(i);

            buf.split_to(1);

            match str::from_utf8(&line) {
                Ok(s) => Ok(Some(s.to_string())),
                Err(_) => Err(io::Error::new(io::ErrorKind::Other
,
                    "UTF-8 invalid")),
            }
        } else {
            Ok(None)
        }
    }
}

```

Apesar de um pouco enrolada, esta solução do `decode` tem um aspecto funcional interessante. Nosso valor de resposta pode

ser resumido com a seguinte linha: `if let Some(i) = { ... } else {Ok(None)}` .

O caso do `else` é bastante intuitivo, pois retorna um "nada". Já o `Some(i)` itera sobre o buffer de bytes para encontrar as posições de `b'\n'` , `buf.iter().position(|&b| b == b'\n')` . Nessas posições, quebramos a sequência em duas com o `let line = buf.split_to(i);` , e retiramos o `b'\n'` aplicando o `buf.split_to(1)` .

Após isso, retornamos um `Ok` para o caso de termos conseguido transformar nossos bytes em `str` , com `match str::from_utf8(&line) { Ok(s) => Ok(Some(s.to_string())) , ... }` . Agora vamos codificar e decodificar bytes em strings UTF-8. Nosso resultado é:

```
extern crate tokio_io;
extern crate bytes;

use std::io;
use std::str;
use self::bytes::BytesMut;
use self::tokio_io::codec::{Encoder, Decoder};

pub struct LineCodec;

impl Decoder for LineCodec {
    type Item = String;
    type Error = io::Error;

    fn decode(&mut self, buf: &mut BytesMut) -> io::Result<Option<String>> {
        if let Some(i) = buf.iter().position(|&b| b == b'\n') {
            let line = buf.split_to(i);

            buf.split_to(1);

            match str::from_utf8(&line) {
                Ok(s) => Ok(Some(s.to_string())),
            }
        }
    }
}
```

```

        Err(_) => Err(io::Error::new(io::ErrorKind::Other
,
                                "UTF-8 invalido")),
    }
    } else {
        Ok(None)
    }
}
}

impl Encoder for LineCodec {
    type Item = String;
    type Error = io::Error;

    fn encode(&mut self, msg: String, buf: &mut BytesMut) -> io::
Result<()> {
        buf.extend(msg.as_bytes());
        buf.extend(b"\n");
        Ok(())
    }
}
}

```

Implementando o protocolo

Nosso próximo passo é transformar o codec em um protocolo real. Felizmente, a crate `tokio-proto` já possui uma grande variedade de estilos de protocolo, como o *multiplexed* e o *streaming*. Para continuarmos com o paradigma do `LineCodec`, vamos desenvolver um protocolo baseado em linha, aplicando um protocolo de pipeline sem streaming.

Vamos criar um arquivo Rust com o nome de `protocol.rs`. No arquivo `lib`, vamos incluir `pub mod protocol;` e injetar esse módulo na `main`, com `mod protocol`. Para o tipo de protocolo que escolhermos, devemos incluir `use tokio_proto::pipeline::ServerProto;` e criar uma nova struct, já que não lidaremos com estados de configuração `pub struct LineProto`.

Neste momento, temos de enfrentar um pouco de boilerplate para definirmos corretamente o estilo de protocolo com o codec de forma assíncrona. Isso resultará em:

```
extern crate tokio_proto;
extern crate tokio_io;

use self::tokio_proto::pipeline::ServerProto;
use self::tokio_io::{AsyncRead, AsyncWrite};
use self::tokio_io::codec::Framed;
use std::io;
use super::codec::LineCodec;

pub struct LineProto;

impl<T: AsyncRead + AsyncWrite + 'static> ServerProto<T> for LineProto {
    type Request = String;
    type Response = String;
    type Transport = Framed<T, LineCodec>;
    type BindTransport = Result<Self::Transport, io::Error>;

    fn bind_transport(&self, io: T) -> Self::BindTransport {
        Ok(io.framed(LineCodec))
    }
}
```

Definimos um tipo `T` que deve possuir as traits `AsyncRead` e `AsyncWrite`, para implementar um `ServerProto` (protocolo de serviço) para a nossa struct `LineProto`. O tipo de `Request` deve ser igual ao tipo do `Item` do `Decoder`, e o tipo de `Response` deve ser igual ao tipo `Item` do `Encoder`.

Nosso tipo `Transport` gera um pouco de boilerplate para fazer liga entre nosso estilo de protocolo e nosso codec. E, finalmente, temos a função que faz o bind do tipo `T`, com a saída `BindTransport` a partir do codec `LineCodec`.

Implementando o serviço

Agora geramos um protocolo genérico de linha, mas não temos muito como utilizá-lo. Para isso, precisamos que ele seja gerenciado por um serviço que indicará como responder a um `request`.

A crate `tokio_service` provê uma `trait` para isso. Para usá-la, devemos criar um arquivo `service.rs`, incluir no arquivo `lib.rs` `pub mod service;`, e disponibilizar nossa implementação de serviço na `main` com `mod service`. Para iniciarmos o arquivo `service.rs`, vamos necessitar inicialmente do trecho a seguir:

```
extern crate tokio_service;

use self::tokio_service::Service;

pub struct EchoService;
```

Como princípio central, um serviço da crate `tokio_service` é uma função assíncrona, não bloqueante, que transforma um `request` em uma `response`. A forma como Tokio se propôs a resolver código assíncrono é por meio das `futures`, vindas da `trait Future`. Elas são basicamente a versão assíncrona de `Result` – na verdade, uma promessa de `Result`.

Para incluir as `futures`, basta adicionar o seguinte código:

```
extern crate futures;

use self::futures::{future, Future};
```

Queremos que nosso serviço retorne o que lhe foi passado, o que usualmente chamamos de `echo`. Para isso, queremos que esse nosso serviço aplique a `future::ok`, pois isto produzirá uma `future` que imediatamente retornará uma promessa de valor, neste caso, retornando o `request` como `response`.

Para manter isso mais simples, criaremos uma `box` que transformará a `future` em uma `trait`, o que vai nos permitir utilizar a `trait` para definir nosso serviço. Isso resultará em:

```
extern crate tokio_service;
extern crate futures;

use self::tokio_service::Service;
use self::futures::{future, Future};
use std::io;

pub struct EchoService;

impl Service for EchoService {
    type Request = String;
    type Response = String;
    type Error = io::Error;
    type Future = Box<Future<Item = Self::Response, Error = Self::Error>>;

    fn call(&self, req: Self::Request) -> Self::Future {
        Box::new(future::ok(req))
    }
}
```

Os tipos de `Request` e `Response` devem ser iguais aos do protocolo. O tipo `Error` para protocolos sem streaming deve ser `io::Error`, já o tipo `Future` possui uma `box` para simplicidade, mas tem como objetivo gerar a `response`. A função `call` é responsável por produzir uma `future` capaz de gerar uma `response` para um `request`, que se torna bastante trivial `Box::new(future::ok(req))` pelo uso da `box`.

Rodando nosso servidor

Finalmente, podemos voltar para a `main` e configurar nosso servidor para rodar. Temos um protocolo e um serviço sobre o qual proveremos, e a única coisa que nos falta é configurar o

servidor para rodá-lo. Faremos isso usando a `TcpServer` , basta adicionar `use tokio_proto::TcpServer` à sua `main` :

```
extern crate tokio_proto;

use self::tokio_proto::TcpServer;

mod codec;
mod protocol;
mod service;

fn main() {
    let addr = "0.0.0.0:12345".parse().unwrap();
    let servidor = TcpServer::new(protocol::LineProto, addr);

    servidor.serve(|| Ok(service::EchoService));
}
```

Primeiro, importamos os módulos que definimos anteriormente com a série de `mod` , depois definimos um endereço `localhost` com uma porta em `addr` . Com isso, podemos passar para nosso `TcpServer` o protocolo que vamos utilizar e o endereço em que ele se encontra.

Por fim, provemos uma maneira de instanciar o serviço para cada nova conexão. Para ver o servidor funcionando, basta rodar `cargo run` e, em outra janela do terminal, rodar `telnet localhost 12345` . Vale a pena rodar em diversos terminais para testar os resultados de assincronicidade. Ao final do capítulo, temos os links com os códigos gerados.

16.2 FUTURES

Tokio existe para resolver problemas de I/O de forma assíncrona, mas o interessante é que a lib não exige que você possua muito conhecimento sobre I/O assíncrono. Um exemplo

simples seria ler um número exato de bytes em um `socket` . Podemos ler 4096 bytes para dentro do vetor `my_vec` .

```
socket.read_exact(&mut my_vec[..4096]);
```

Como a biblioteca padrão é síncrona, a chamada a este `socket` torna-se bloqueante, permanecendo dormente enquanto mais bytes não são recebidos. Isso pode se tornar um problema para contextos de servidores escalados, como servir a diferentes clientes de forma concorrente.

Para isso, em vez de bloquearmos a `thread` com o `request` até que ele esteja completo, podemos registrá-lo como *em andamento* e liberar a `thread` para atender a outra demanda, permitindo que ela volte quando o `request` estiver pronto para ser atendido. Isso significa que podemos usar uma única `thread` para gerenciar um número arbitrário de conexões, com cada conexão usando recursos mínimos. É nisso que as `futures` brilham.

Uma `future` é um valor que está no processo de ser computada, mas talvez não esteja pronta para fazer o todo. Geralmente, ela está completa ou disponível devido a um evento que aconteceu em outro lugar. `Futures` podem representar uma série de eventos:

- **Uma query a um banco de dados** – Quando a `query` termina, a `future` é tida como completa, e o valor é o resultado da `query` .
- **Uma invocação RPC a um servidor** – Quando o servidor responde, a `future` se completa e seu valor é a resposta do servidor.
- **Timeouts** – Se o tempo terminar, a resposta da `future` será `()` .

- **Uma tarefa intensa de CPU de longa duração rodando em um pool de threads** – Quando a task terminar, a `future` se completa e o resultado é o retorno da tarefa.
- **Lendo bytes de um socket** – Quando os bytes estão terminados, a `future` se completa, e os bytes podem ser retornados diretamente.

Resumidamente, as `futures` são aplicadas a todos os tipos de eventos assíncronos. A assincronicidade reflete-se no fato de você receber a `future` instantaneamente, sem bloquear a `thread`, apesar de o seu valor representar um que está disponível em algum momento do tempo, o futuro.

Um exemplo com futures

Nosso objetivo nesse exemplo é adicionar timeouts a uma task custosa para o sistema: calcular se um número é primo de forma ineficiente. Para isso, o primeiro passo é iniciar uma app com cargo `new --bin timeout-primos && cd timeout-primos`.

Vamos começar criando nossa função *burra* que verificará se o número é primo percorrendo todos, de 2 até ele mesmo, e verificando se são divisíveis (a ideia é que ela não seja otimizada mesmo):

```
const PRIMO: u64 = 15485867;

fn main() {
    if eh_primo(PRIMO) {
        println!("{}", PRIMO)
    }
}
```

```

}

fn eh_primo(numero: u64) -> bool {
    for i in 2..numero {
        if numero % i == 0 { return false }
    }
    true
}

```

Essa seria a versão síncrona do nosso problema, na qual a thread `main` fica bloqueada até que a função `eh_primo` retorne seu valor.

16.3 A VERSÃO ASSÍNCRONA

Agora vamos utilizar `pools` de threads e `futures` para resolver esse problema. Então, vamos precisar das seguintes crates:

```

[dependencies]
futures = "*"
futures-cppool = "*"

```

Primeiramente, precisamos disponibilizá-las no nosso código, com:

```

extern crate futures;
extern crate futures_cppool;

use futures::Future;
use futures_cppool::CpuPool;

fn main() {...}

```

Com isso, podemos criar nossa pool de threads com `let pool = CpuPool::new_num_cpus()`. Agora precisamos criar nossa função que gera nossa avaliadora de primos e retorna uma `future`, a `let prime_future = pool.spawn_fn(|| {...})`. Dentro da função `pool.spawn_fn`, temos:

```
pool.spawn_fn(|| {
    let primo = eh_primo(PRIMO);

    let res: Result<bool, ()> = Ok(primo);
    res
});
```

Verificamos se o número é primo com `eh_primo(PRIMO)` e, se for, precisamos encapsular em um tipo `Result<bool, ()>`. Nosso resultado final é algo assim:

```
extern crate futures;
extern crate futures_cpupool;

use futures::Future;
use futures_cpupool::CpuPool;

const PRIMO: u64 = 15485867;

fn main() {
    let pool = CpuPool::new_num_cpus();

    let future_primo = pool.spawn_fn(|| {
        let primo = eh_primo(PRIMO);

        let res: Result<bool, ()> = Ok(primo);
        res
    });
    println!("Future Criada");

    if future_primo.wait().unwrap() {
        println!("Primo");
    } else {
        println!("Composto");
    }
}

fn eh_primo(numero: u64) -> bool {
    for i in 2..numero {
        if numero % i == 0 { return false }
    }
    true
}
```

Utilizar `wait` não é algo muito comum, mas nos permite entender a diferença entre a `future` e o valor que ela produz. O `unwrap` serve para garantir a extração do valor dela.

Adicionando timeouts

Observando o exemplo anterior, podemos pensar que tomamos um caminho difícil para trabalhar com `thread pools`, e até meio inútil. Porém, a graça aparece quando podemos limitar ainda mais as condições de assincronicidade. É neste momento que a força das `futures` aparece, por conta de sua habilidade de combinar resultados – o que demonstraremos com a adição de `timeouts` ao nosso código.

Isso nos permitirá perceber como as `futures` podem ser escaláveis e combináveis de forma complexa, delegando à biblioteca a responsabilidade de gerenciar todos os estados e as sincronizações. Antes de iniciar, vale lembrar de que é preciso incorporar uma nova `crate`, a `tokio-timer` = "*", e disponibilizá-la no código:

```
extern crate futures;
extern crate futures_cpupool;
extern crate tokio_timer;

use std::time::Duration;

use futures::Future;
use futures_cpupool::CpuPool;
use tokio_timer::Timer;

const PRIM0: u64 = 15485867;

fn main() {

    let timeout = Timer::default()
        .sleep(Duration::from_millis(500))
```

```

        .then(|_| Err(()));

let primo = CpuPool::new_num_cpus()
    .spawn_fn(|_| {
        Ok(eh_primo(PRIMO))
    });

match timeout
    .select(primo)
    .map(|(ok, _)| ok)
    .wait() {
    Ok(true) => println!("Primo"),
    Ok(false) => println!("Composto"),
    Err(_) => println!("Timeout"),
}
}

fn eh_primo(numero: u64) -> bool {
    for i in 2..numero {
        if numero % i == 0 { return false }
    }
    true
}
}

```

Digamos que nosso código ficou mais pomposo agora que empilhamos funções. Muitas pessoas diriam que é um código mais funcional, mas creio que ele só esteja mais simples e limpo. A primeira diferença agora é a criação do valor de timeout, com `Timer::default()`, com a definição de duração e resultado.

Agora limpamos a variável `primo` de forma que ela simplesmente retorna a correspondência lógica da função `eh_primo`. Assim, aplicamos `timeout` em comparação a `primo`, esperamos pelo primeiro e checamos o resultado.

Para as configurações definidas, teremos `timeout`; mas se aumentarmos o tempo de `timeout` ou utilizarmos um `primo` menor, como 157, teremos o valor de retorno `primo`.

Duas funções diferentes e importantes apareceram agora, a `select` e a `then` :

- `then` : permite sequenciar `futures` para que elas rodem mesmo depois de receber o valor de outra. Neste caso, estamos alterando o valor da `future timeout` para `Err(())` , mesmo depois de outra `thread` encerrar.
- `select` : combina duas `futures` do mesmo tipo para que elas possam competir para se encerrar. Ela entrega um par, em que o primeiro componente é o valor produzido pela primeira `future` e o segundo joga de volta a outra `future` . Aqui, apenas tomamos o valor `OK` .

Códigos

- Código do projeto que realizamos usando `tokio-proto` : <https://github.com/naomijub/tokio/tree/master/tokio-proto/src>
- Código do projeto que realizamos utilizando `futures` : <https://github.com/alexcrichton/futures-rs>

Agora que vimos muitos exemplos de Rust no aspecto funcional e, principalmente, no concorrente, podemos concluir alguns aspectos importantes. Rust não é uma linguagem funcional, mas foi claramente pensada de uma forma a permitir *functional first* (programação funcional em primeiro lugar), o que nos permite um desenvolvimento limpo e claro de seu código.

Além disso, temos fortes aspectos de concorrência, que nos permitem chamar a linguagem de *concurrency first* (concorrência primeiro), já que o seu objetivo inicial era lidar com estes

problemas sem recorrer a soluções complexas em C++, garantindo a performance. Isso se mostra real também quando comparamos aspectos fortes de Rust com Clojure, uma linguagem funcional e com alta capacidade de concorrência.

Alguns desafios interessantes para continuar nossa jornada em Rust podem ser o desenvolvimento de uma GUI que recebe updates assíncronos, um algoritmo genético que as gerações não ocorrem de forma linear e homogênea, ou até mesmo uma calculadora que divide seus blocos de cálculo para retornar mais rápido.

No meu projeto, exploramos a possibilidade de utilizar Rust de forma a criar um servidor que conseguia se comunicar de forma assíncrona com o banco de dados, e gerar threads não bloqueantes de sistema. Uma implementação assim pode ser extremamente performática e mais fácil de manter do que um servidor Spring.

BIBLIOGRAFIA

BLANDY, Jim. *Why Rust?* O'Reilly, 2015.

KAIHLAVIRTA, Vesa. *Mastering Rust*. Packt, 2017.

NICHOLS, Carol. *Increasing Rust's Reach*. Jun. 2017.
Disponível em: <https://blog.rust-lang.org/2017/06/27/Increasing-Rusts-Reach.html>.