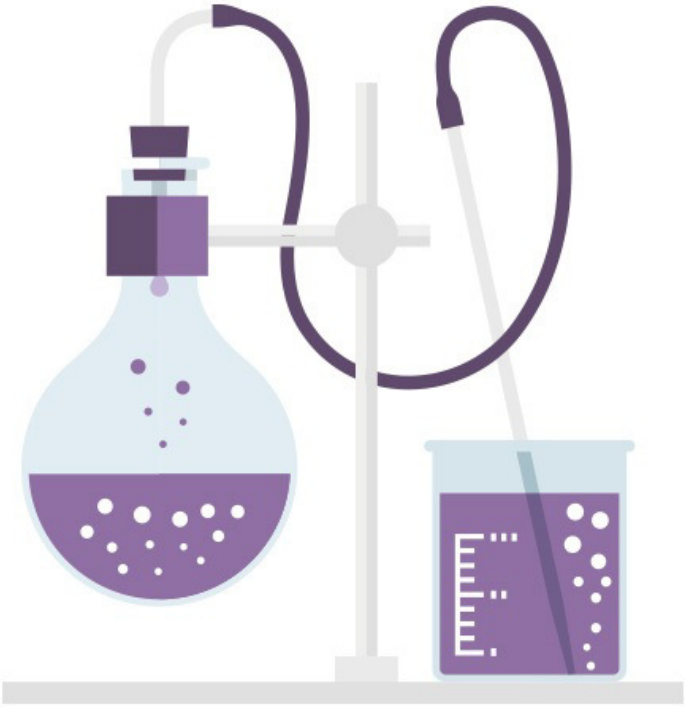
Elixir

Do zero à concorrência



Casa do Código

TIAGO DAVI

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: 978-85-5519-261-6

EPUB: 978-85-5519-262-3

MOBI: 978-85-5519-263-0

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

SOBRE O AUTOR

Comecei a desenvolver minhas primeiras linhas de código a mais ou menos doze anos atrás, em 2004, enquanto estava, por incrível que pareça, em um curso de Web Design com Photoshop e outras ferramentas visuais. Realmente achava que eu seria um designer ou coisa do tipo. Mas assim que coloquei os olhos no código, que na época era apenas JavaScript com HTML, fiquei muito entusiasmado.

Daquele tempo para cá, me aprofundei mais e mais nesta área, e trabalhei em diversas organizações como programador, em que tive a oportunidade de desenvolver vários projetos nas mais diversas tecnologias. Hoje, sou formado em gestão de TI, com curso de pós-graduação em segurança da informação e trabalho para uma empresa no Reino Unido.

Eu diria que, após todos esses anos e experiências, se existe uma coisa que gosto neste setor, esta seria escrever código. Não sou preso a nenhuma linguagem específica — apesar de ter uma delas tatuada em meu braço (risos) —, gosto de observar a sintaxe, os problemas que elas podem resolver e os diferentes paradigmas, principalmente quando se trata do paradigma funcional.

Neste livro, gostaria de lhe convidar a explorar comigo os fundamentos de uma das mais promissoras linguagens funcionais do momento, a linguagem Elixir.

Fique à vontade para me encontrar no Twitter ou GitHub, caso queira fazer um elogio ou mesmo uma crítica sobre o livro.

- https://twitter.com/tiagodavibr
- https://github.com/tiagodavi

PREFÁCIO

Décadas atrás, os processadores eram projetados com apenas um núcleo. Então, naquela época, fazia sentido programar sem pensar em distribuição de tarefas por núcleos de processamento.

A maioria dos programadores não se preocupava se estava utilizando todo o potencial do processador para executar tarefas em paralelo. Entretanto, hoje, depois de várias pesquisas neste setor, os processadores evoluíram e passaram a ter não apenas um núcleo, mas diversos! Hoje estes processadores são conhecidos como *multi-core* e são muito comuns, até mesmo em aparelhos celulares.

No momento, vivemos neste cenário moderno e em constante evolução, mas muitos programadores continuam escrevendo código de maneira clássica, sem considerar que hoje podemos extrair muito mais dos processadores no sentido de podermos processar diversas tarefas em paralelo, a fim de termos um resultado muito mais eficiente.

Diversas linguagens tentam resolver esse problema. Mas a implementação desses recursos em geral é muito complicada e, no final, eles não conseguem resolver o problema de fato.

Sistemas orientados a objeto têm de lidar com mutação de estado em processos paralelos. E quando você tem diversos núcleos disputando o mesmo espaço de memória, para checar esse estado, o resultado final pode ser um desastre.

Neste livro, vamos estudar a linguagem de programação Elixir

que resolve muito bem este problema. Por ser uma linguagem imutável e utilizar o paradigma funcional, Elixir nos permite pensar em termos de funções e transformação de dados.

Elixir também é baseado em processos que se comunicam isoladamente e, por isso, não sofre efeitos colaterais indesejados como outras linguagens. Em vez disso, ele potencialmente cria uma nova versão limpa dos dados a cada possível mutação.

Elixir executa código em pequenos processos e cada um com seu próprio estado, o que faz com que a construção de sistemas distribuídos e concorrentes seja feita de forma natural, transparente e fácil.

PARA QUEM ESTE LIVRO FOI ESCRITO

Se você é recém-chegado no universo da programação, creio que este livro não lhe cairá muito bem devido aos diversos aspectos técnicos que são comparados e explicados aqui. Mas se por outro lado, você já possui alguma experiência com programação, seja qual for a linguagem, e tem vontade de aprender um pouco mais sobre a forma funcional de pensar, utilizando como base os diversos aspectos da linguagem de programação Elixir, então este livro é para você. Espero que sirva bem ao seu propósito.

É requerido que você entenda um pouco sobre Orientação a Objetos, estruturas de dados, funções, variáveis e constantes. Você terá de lidar com terminal, editores de código, Elixir e Erlang. Logo, conhecer estes assuntos previamente pode ajudar.

Como ler este livro

Sugiro que leia em ordem, do início ao fim, pois os assuntos são abordados pouco a pouco e o conhecimento é exigido conforme os capítulos vão avançando. Porém, nada o impede de visitar somente o assunto que lhe interessa, desde que você já saiba do que se trata.

Palavras em destaque são escritas desta maneira: `destaque` . Código elixir que deve ser digitado no terminal aparece em blocos:

```
iex> IO.puts []
:ok
```

```
#exemplo
defmodule Modulo do
  def funcao do
    "resultado"
  end
end
```

Sumário

Versão: 21.7.7

Capítulo 1

INTRODUÇÃO

Elixir é uma linguagem de programação moderna, dinâmica e funcional, que foi desenvolvida por um brasileiro conhecido como José Valim. Ela serve para resolver problemas de escalabilidade, tolerância a falhas e alta concorrência, sem abrir mão de desempenho e produtividade.

Por ser executada sob a máquina virtual do Erlang, originalmente projetada para este propósito, Elixir nativamente nos traz estes e outros recursos. Ele estende a capacidade do Erlang, acrescentando novas características que abstraem conceitos complexos, e funciona como uma camada que usufrui do poder do Erlang, mas com uma curva de aprendizado mais baixa e uma série de outros recursos úteis.

Erlang foi projetado para lidar com sistemas altamente concorrentes e que devem funcionar próximo de 100%, sem falhas. Porém, sua sintaxe é complexa e possui uma curva de aprendizado alta e difícil de estender para outros domínios, então é aí que entra o Elixir, justamente para preencher estas deficiências do Erlang.

Existem alguns cases interessantes que tiram proveito do Erlang, como é o caso do Facebook — que escreveu seu sistema de chat nesta tecnologia — e do WhatsApp — que consegue processar

dezenas de milhares de mensagens com apenas um servidor, usando menos da metade de sua capacidade.

Erlang é uma linguagem de propósito geral, apesar de ser geralmente classificada como uma linguagem exclusiva para uso em sistemas distribuídos. Duas das suas principais características são o *hot swapping*, que permite que o código seja alterado sem parar o sistema que o está executando, e a comunicação entre processos, feita por troca de mensagens em vez de variáveis compartilhadas, o que permite que a linguagem não tenha de gerenciar o estado dessas variáveis.

Erlang é diferente de todas as outras linguagens porque consegue fazer com sucesso o que todas as outras tentam e "quase" conseguem. Já o Elixir fornece tudo o que Erlang, com mais de 20 anos no mercado, tem para oferecer. Porém, ele o faz com uma sintaxe mais agradável e conceitos que tornam o desenvolvedor mais produtivo.

Elixir vem crescendo muito nos últimos anos. Hoje já existem diversas conferências sobre o assunto pelo mundo, e diversas empresas têm implementado seus projetos utilizando esta tecnologia. Isso faz com que existam boas oportunidades de trabalho e de projetos inovadores e desafiantes.

Elixir também possui um Framework conhecido como Phoenix. Ele permite criar aplicações web incríveis e sem dor.

1.1 O PARADIGMA FUNCIONAL

O paradigma funcional não é algo novo. Na verdade, já existe há muito tempo e foi geralmente empregado para fins acadêmicos

e científicos, mas hoje seu uso é cada vez mais comum em outros setores, como o desenvolvimento para web, por exemplo.

Pensar funcional é pensar em programar em termos de transformação de dados por funções. A função seria a unidade principal dentro do código e o objetivo seria expressar sua intenção através da composição de diversas funções menores.

O modelo tradicional de desenvolvimento, conhecido como imperativo, frequentemente esconde efeitos colaterais por trás da manutenção do estado das variáveis e dos objetos. E conforme seu sistema cresce, fica cada vez mais complicado entender este comportamento implícito que acontece na comunicação entre estes objetos.

Como poderíamos solucionar estes problemas?

A solução seria pensarmos de maneira funcional ao mesmo tempo em que tiramos vantagem de tecnologias como Elixir. A linguagem nos força a tratar logo no início da manutenção do estado e nos fornece subsídios para lidar com isso de maneira apropriada, coisa que outras linguagens são incapazes de fazer.

Linguagens funcionais não têm apenas vantagens, e é claro que existem possíveis desvantagens em utilizar esta tecnologia em seus projetos, como por exemplo, a curva de aprendizado.

Apesar de Elixir possuir uma curva baixa, você provavelmente vive muito bem programando na tecnologia que você já domina e terá de investir algum tempo para aprender a maneira funcional de programar. Sem contar que algumas linguagens funcionais usam uma sintaxe totalmente diferente do tradicional com que estamos acostumados, como é o caso do Clojure, que é funcional e tem uma

sintaxe derivada do Lisp, então com certeza a curva de aprendizado pode ser um obstáculo.

Porém, o aprendizado contínuo nos torna desenvolvedores melhores, e é sempre interessante investirmos em conhecimento, principalmente quando você dedica tempo a aprender uma tecnologia que muda a sua maneira de pensar. Isso com toda certeza será útil para enriquecer suas habilidades e lhe permitir enxergar as coisas sob outro ponto de vista.

O paradigma funcional pode fazer com que suas aplicações tenham desempenho bem superior a aplicações desenvolvidas de forma tradicional ou orientada a objetos. Isso porque linguagens como Elixir podem usufruir de todo o poder de processadores multi-core que são os mais comuns nos servidores atualmente.

1.2 INSTALANDO ELIXIR

Para instalar Elixir, é muito simples. Siga o link (http://elixir-lang.org/install.html) e faça a instalação correspondente ao seu sistema operacional.

Todos os exemplos deste livro foram testados no `Erlang v19` e no `Elixir v1.4.1`.

1.3 IEX — ELIXIR INTERATIVO

Para testar nossos exemplos, vamos iniciar o *IEx* no terminal. IEx nada mais é do que um *REPL* (*Read–eval–print loop*) que interpreta e executa código Elixir diretamente no terminal. Se você vem de linguagens como Python ou Ruby, já deve ter visto

ferramentas similares.

Para iniciar, basta abrir o terminal e digitar `iex`. Para fechar, basta pressionar `Ctrl + C` duas vezes.

```
$ iex
Erlang/OTP 19 [erts-8.2] [source-fbd2db2] [64-bit] [smp:8:8] [asy
nc-threads:10] [kernel-poll:
false]

Interactive Elixir (1.4.1) - press Ctrl+C to exit (type h() ENTER
for help)
iex(1)>
```

Isso mostra que tudo foi instalado corretamente e agora estamos aptos a inserir código Elixir para ser executado no terminal.

Vamos ao tradicional *Hello World* e uma operação aritmética básica para exemplificar. Neste código, simplesmente lançaremos uma mensagem de texto na tela e multiplicaremos dois números. Perceba que é possível continuar a expressão na linha de baixo, quando esta ainda não foi concluída pelo interpretador.

```
iex(1)> IO.puts "Hello World"
Hello World
:ok
iex(2)> 5 *
...(2)> 4
20
```

O número do IEx é incrementado após cada execução realizada com sucesso. Vou omitir este número na maioria dos exemplos que se seguem.

Perceba que existe um `:ok` abaixo do Hello World. Isso acontece porque `puts` é uma função do módulo `IO` com efeitos colaterais. Em outras palavras, ele escreve o argumento passado na

tela e esse `:ok` é o que chamamos de `atom` dentro de Elixir. Não se preocupe em entendê-los agora, pois entrarei em detalhes sobre eles nos próximos capítulos. O que você precisa saber por enquanto é que é muito comum termos funções retornando `:ok` para representar sucesso e `:error` para representar erros dentro de Elixir.

1.4 ARIDADE DE FUNÇÕES

Aridade em linguagens funcionais corresponde ao número de argumentos que uma determinada função recebe. Dentro de Elixir, podemos sobrecarregar funções apenas alterando o número de argumentos, ou o padrão que a função recebe como argumento. Neste caso, é algo conhecido como *Pattern Matching*.

Abra seu IEx e invoque a função `h(help ou ajuda)` para obter ajuda. Sempre que você precisar obter detalhes sobre determinada função ou módulo, você pode executar a função `h`, passando a função ou módulo como argumento, ou executar `h` sem argumentos para obter uma lista completa do que é possível fazer com esta função.

Vejamos um exemplo de uso dela:

```
iex> h

                                    IEx.Helpers

Welcome to Interactive Elixir. You are currently seeing the docum
entation for
the module IEx.Helpers which provides many helpers to make Elixir
s shell more
joyful to work with.
```

```
This message was triggered by invoking the helper h(), usually re
ferred to as
h/0 (since it expects 0 arguments).

You can use the h function to invoke the documentation for any El
ixir module or
function:

 h  Enum
 h  Enum.map
 h  Enum.reverse/1

You can also use the i function to introspect any value you have
in the shell:

 i  "hello"

There are many other helpers available:

  • b/1           - prints callbacks info and docs for a given mo
dule
  • c/2           - compiles a file at the given path
  • cd/1          - changes the current directory
  • clear/0       - clears the screen
  • flush/0       - flushes all messages sent to the shell
  • h/0           - prints this help message
  • h/1           - prints help for the given module, function or
macro
  • i/1           - prints information about the given data type
  • import_file/1 - evaluates the given file in the shell's conte
xt
  • l/1           - loads the given module's beam code
  • ls/0          - lists the contents of the current directory
  • ls/1          - lists the contents of the specified directory
  • pid/3         – creates a PID with the 3 integer arguments pa
ssed
  • pwd/0         – prints the current working directory
  • r/1           – recompiles and reloads the given module's sou
rce file
  • respawn/0     – respawns the current shell
  • s/1           – prints spec information
  • t/1           – prints type information
  • v/0           – retrieves the last value from the history
  • v/1           – retrieves the nth value from the history
```

```
Help for all of those functions can be consulted directly from th
e command line
using the h helper itself. Try:

 h(v/ 0)

To learn more about IEx as a whole, just type h(IEx).
```

Os números depois das funções, como `b/1` e `clear/0`, indicam a aridade delas, que, como já explicado, significa o número de argumentos que essas funções podem receber. Quando existe aridade zero, significa que a função pode ser executada sem nenhum argumento.

Experimente buscar ajuda para módulos ou funções com aridades específicas.

```
iex> h(c/2)
iex> h Enum.reverse
iex> h Enum.reverse/1
iex> h Enum.reverse/2
```

Para descobrir todas as funções disponíveis em um módulo, você pode digitar o nome do módulo seguido de um `ponto` e um `tab` para autocompletar.

```
iex> Enum.
```

Para descobrir mais sobre o IEx, apenas execute a função `h` passando `IEx`.

```
iex> h(IEx)
```

1.5 EXERCÍCIOS

- Experimente abrir o IEx e executar algumas operações matemáticas;

- Experimente utilizar o comando de ajuda para outros módulos;
- Leia a documentação no site oficial do Elixir.

1.6 VOCÊ APRENDEU

- O que é Elixir, suas principais características e como instalá-lo;
- As particularidades do paradigma funcional;
- Aridade de funções;
- Comandos básicos do IEx.

Capítulo 2

FUNDAMENTOS

Neste capítulo, vamos aprofundar sobre as estruturas de dados, tipos básicos e chamadas de funções úteis. Entenderemos a diferença entre *Atoms* e *Strings*, e estudaremos como o código Elixir é interpretado pela máquina virtual do Erlang.

Antes de ser executado pela máquina virtual, todo o código Elixir é compilado. Isso significa que o compilador verifica a presença de possíveis erros e realiza diversas otimizações antes de produzir o código final que será de fato executado.

Elixir compila para a mesma estrutura que Erlang, então é totalmente possível usar funções do Erlang dentro do Elixir sem problema algum. Este seria um caso no qual Erlang fornece algo que Elixir não fornece, e você só precisaria pegar e utilizar sem necessidade de reinventar a roda.

Elixir é funcional e não orientada a objetos, o que significa que não existem classes ou herança. Elixir resolve estes problemas de maneira muito mais simples.

Outra coisa interessante é que Elixir é construído com Elixir! A maioria dos recursos presentes na linguagem foram desenvolvidos utilizando a própria linguagem através de metaprogramação e macros.

2.1 TUDO É UMA EXPRESSÃO

Elixir sempre retorna um resultado. Isso acontece porque, dentro dele, tudo é considerado uma expressão e toda expressão deve retornar um valor. Se você executar `IO.puts` com o número `100`, você verá que o número 100 surgirá na tela.

Nesta linguagem, até mesmo um item vazio ou nulo é considerado uma expressão. Logo, se você passar `[]`, que é uma lista vazia, para `IO.puts`, Elixir entenderá que você passou uma lista vazia de caracteres e certamente ela pode ser impressa como uma string vazia.

```
iex> IO.puts []
:ok
```

Mesmo ao somar dois números para uma variável, você ainda terá um valor retornado na tela.

```
iex> a = 2 + 2
4
```

Em Elixir, não há declarações sem retorno de valor. Declarações são todas as instruções que um programador pode especificar para um computador executar. Essas instruções podem ser somar dois números, conectar em outro computador, processar um arquivo externo etc.

Essas ações podem ter os mais diversos resultados, porém o código por trás delas, por ele mesmo, geralmente não retorna nada a não ser que você declare explicitamente que você deseja um retorno. Isso é muito comum em linguagens imperativas como Java.

Nesta linguagem, você também pode fazer todas essas tarefas,

porém cada instrução sempre retornará alguma coisa por si mesma. Essa sutil diferença faz com que componentes Elixir sejam muito mais legíveis e naturais, porque o fluxo de execução do programa torna-se mais evidente.

2.2 INSPECIONANDO CÓDIGO

Existe uma função muito útil que é interessante que você a aprenda desde já, para que possa inspecionar seus exemplos de código e analisar melhor o que você está fazendo em seu programa.

A função `inspect/2` é responsável por converter qualquer estrutura Elixir em uma string amigavelmente formatada. Essencialmente esta função nos permite enxergar dentro de estruturas de dados de maneira legível e será útil para debugar nosso código.

Observe um exemplo de como podemos inspecionar elementos dentro de uma lista, ou mesmo o conteúdo de uma variável.

```
iex> inspect ["Livro de Elixir", 1, 2, 3, 4, 5]
"[\"Livro de Elixir\", 1, 2, 3, 4, 5]"

iex> a = 2 + 2
4

iex> inspect a
"4"
```

Você pode pensar que essa função é uma espécie de `ToString` de outras linguagens, contudo, jamais a use com este propósito.

2.3 INTRODUÇÃO A TIPOS

Como a maioria das linguagens, Elixir também tem tipos numéricos, booleanos e coleções. Além disso, nos traz alguns tipos extras como atoms, binários e funções. Isso mesmo, funções também são um tipo dentro de Elixir. Nesta seção, estudaremos os diversos tipos e como aplicá-los na prática.

Tipos numéricos

Nesta categoria, podemos incluir os inteiros e pontos flutuantes. Vamos dar uma olhada neles utilizando o Iex.

```
iex(1)> 59
59
```

Os números inteiros também podem ser escritos com underscores para facilitar a leitura de números grandes, principalmente quando há presença de muitos zeros.

```
iex(1)> 1_00
100

iex(2)> 1_000
1000

iex(3)> 10_000
10000

iex(4)> 100_000
100000

iex(5)> 1_000_000
1000000
```

Os pontos flutuantes são escritos com um ponto. Eles devem ter, no mínimo, um dígito antes e depois do ponto decimal.

```
iex(6)> 1.0
1.0
```

```
iex(7)> 7.2
7.2

iex(8)> 49.7
49.7
```

Operações matemáticas, como adição, subtração e multiplicação, funcionam como esperado, contudo uma coisa curiosa acontece na divisão:

```
iex(9)> 4+2
6

iex(10)> 8-3
5

iex(11)> 7*7
49

iex(12)> 4/2
2.0
```

Na divisão, acontece uma transformação implícita que recebe os inteiros como argumento e retorna um ponto flutuante como resultado. Neste caso, o operador `/` sempre vai retornar um ponto flutuante como resultado.

Se você quer resultados inteiros de uma divisão, você deve utilizar funções próprias para obter isso: `div` para obter o resultado da divisão, e `rem` para obter o resto.

```
iex(13)> div(4,2)
2

iex(14)> rem(4,2)
0

iex(15)> div(9,2)
4

iex(16)> rem(9,2)
```

```
1
```

Binários, hexadecimais e octais

Elixir possui diferentes prefixos para converter determinadas bases numéricas em outras. Para converter de binário para decimal, nós podemos utilizar o prefixo `0b`.

```
iex(1)> 0b1111
15
```

Para converter números hexadecimais em decimais, temos o prefixo `0x`.

```
iex(2)> 0xF
15
```

De números octais para decimais, temos o `0o`.

```
iex(3)> 0o17
15
```

Atoms

Atoms são constantes e seu nome é seu valor em si mesmo. Eles são classificados como símbolos em outras linguagens e podemos representá-los acrescentando `:` (dois pontos) seguido de um nome que pode conter underscores, arrobas e outros sinais, como exclamação e interrogação. Eles são bastante utilizados para sinalizar sucesso, erro e chaves de dicionários.

Atoms com o mesmo nome são sempre iguais mesmo que eles tenham sido criados por aplicações diferentes em computadores diferentes, porém eles não são mutáveis e também não são coletados pelo *garbage collector* (coletor de lixo). Isso significa que a memória consumida por atoms jamais vai expirar até o término

da execução do programa.

```
iex(1)> :ok
:ok

iex(2)> :error
:error

iex(3)> :tiago
:tiago

iex(4)> :funciona?
:funciona?

iex(5)> :exclua!
:exclua!

iex(6)> :nome_com_sobrenome
:nome_com_sobrenome
```

Existem algumas funções úteis que podemos usar para lidar com atoms. As funções `Atom.to_string/1` e `String.to_atom/1` com seus nomes bem sugestivos permitem converter de atom para string e de string para atom, respectivamente.

```
iex(1)> Atom.to_string(:exclua!)
"exclua!"

iex(2)> String.to_atom("funciona?")
:funciona?
```

Booleanos

Como em outras linguagens, Elixir também possui sua maneira de expressar booleanos e tem três valores para representá-los: `true`, `false` e `nil` — que é avaliado como `false` em contextos booleanos. Todos eles são atalhos para `atoms` de mesmo nome, então `true` é o mesmo que `:true`, e assim por

diante.

Na maioria dos contextos, qualquer valor diferente de `false` ou `nil` é tratado como `true`, inclusive o `zero`, que em algumas linguagens é tratado como `false`.

```
iex(1)> 0 || false
0

iex(2)> 0 || nil
0
```

Estes operadores esperam apenas `true` ou `false` como primeiro argumento. Se tentar passar `nil` como se fosse `false`, ou outros argumentos, você terá um erro classificado como `ArgumentError`.

- *a* `or` *b* — `true` se *a* é `true`; caso contrário, *b*.
- *a* `and` *b* — `false` se *a* é `false`; caso contrário, *b*.
- `not` *a* — `false` se *a* é `true`; caso contrário, `true`.

Estes operadores esperam argumentos de qualquer tipo e qualquer valor diferente de `nil` ou `false` é interpretado como `true`.

- *a* `||` *b* — *a* se *a* é `true`; caso contrário, *b*.
- *a* `&&` *b* — *b* se *a* é `true`; caso contrário, *a*.
- `!a` — `false` se *a* é `true`; caso contrário, `true`.

Finalmente, veremos alguns operadores de comparação:

- *a* `===` *b* — `1 === 1.0` é `false`.
- *a* `!==` *b* — `1 !== 1.0` é `true`.
- *a* `==` *b* — `1 == 1.0` é `true`.
- *a* `!=` *b* — `1 != 1.0` é `false`.

- *a* `>` *b* — se *a* é maior do que *b*.
- *a* `>=` *b* — se *a* é maior ou igual a *b*.
- *a* `<` *b* — se *a* é menor do que *b*.
- *a* `<=` *b* — se *a* é menor ou igual a *b*.

Existe uma função em Elixir para checar se um argumento é ou não booleano. A função `is_boolean/1` nos permite fazer esta checagem. E lembre-se de que `true`, `false` e `nil` são atalhos para `atoms` de mesmo nome, e é justamente por isso que estes `atoms` são considerados booleanos.

```
iex(1)> is_boolean(true)
true

iex(2)> is_boolean(:true)
true

iex(3)> is_boolean(1)
false

iex(4)> is_boolean(false)
true

iex(4)> is_boolean(:false)
true

iex(4)> is_boolean(:nil)
false
```

Strings

Strings são caracteres UTF-8 envolvidos por aspas duplas. UTF-8, como você já deve saber, significa que pode ser utilizado qualquer caractere da tabela UTF-8. Além disso, strings têm poderes extras, elas podem ser escritas diretamente com quebras de linha, aceitam *heredoc* (*documentação entre caracteres de abertura e fechamento*) e interpolação — que vai além da

interpolação comum porque permite realizar operações diretamente dentro dela.

```
iex(1)> "Livro de Elixir"
"Livro de Elixir"

iex(2)> "Livro
...(2)> de Elixir"
"Livro\nde Elixir"

iex(3)> IO.puts("Livro de \nElixir")
Livro de
Elixir
:ok

iex(4)> a = "Elixir"
"Elixir"

iex(5)> "Livro de #{a}"
"Livro de Elixir"

iex(6)> "2 + 2 é igual a #{2 + 2}"
"2 + 2 é igual a 4 "
```

Heredocs são mais comuns para escrever documentações e podemos usá-los com aspas triplas. Você encontrará bastante documentação escrita dessa maneira dentro dos módulos de Elixir.

```
iex(7)> IO.puts """
...(7)> Minha primeira
...(7)> documentação de
...(7)> exemplo
...(7)> """
Minha primeira
documentação de
exemplo
:ok
```

Elixir também possui diversas funções para manipular strings. Mostrarei alguns exemplos do que pode ser feito com elas.

Perceba que, no decorrer do livro, aprendemos algumas

funções úteis, contudo, os exemplos aqui listados são apenas uma pequena parcela do que existe dentro do universo de Elixir. Sugiro que você leia a documentação oficial sempre que precisar complementar seus estudos além do que aprendeu com este livro.

A função `reverse` do módulo `String`, com o nome bem sugestivo, pode inverter seus caracteres:

```
iex(10)> String.reverse("Minha String")
"gnirtS ahniM"

iex(11)> String.reverse("ANA")
"ANA"
```

A função `length` pode nos dar a quantidade de caracteres ou tamanho da string:

```
iex(12)> String.length("ANA")
3
```

Também podemos aumentar a caixa do primeiro caractere de uma string.

```
iex(13)> String.capitalize("ana")
Ana
```

Listas

Listas são um tipo de coleção de elementos e podem conter diversos outros tipos dentro dela. Por exemplo, podemos ter listas de números com atoms.

Listas não devem ser comparadas com arrays de outras linguagens, porque não o são. Na verdade, o conceito aqui é que listas podem ter `head` (*cabeça*) e `tail` (*cauda*). A cabeça contém o valor e a cauda por si mesma é a lista inteira. Por causa dessa

implementação, elas podem ser percorridas facilmente.

Elas também são imutáveis, ou seja, podem ser criadas, mas nunca alteradas. Então, você nunca terá uma cópia de uma lista ao remover ou adicionar elementos. O que você terá, na verdade, é uma nova lista.

```
iex(2)> ["Livro", 2, 1.5, true, nil, :ok]
["Livro", 2, 1.5, true, nil, :ok]
```

Podemos facilmente concatenar listas utilizando a função `++/2`:

```
iex(3)> [1,2,3,4,5] ++ [6,7,8,9,10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Porém, adicionar elementos no fim da lista é um processo lento, já que o Elixir precisa copiar a lista inteira e criar uma nova com o novo elemento. Neste caso, prefira concatenar na frente da lista em vez de no fim, uma vez que neste caso o processamento será muito mais eficiente.

```
iex(11)> lista = [1,2,3,4,5]
[1, 2, 3, 4, 5]

iex(12)> nova_lista = [6,7,8,9,10 | lista] |> Enum.sort
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Podemos também subtrair listas com a função `--/2`. Mas, ao subtrair elementos, a função remove o primeiro elemento que casa, e não todos os repetidos.

```
iex(4)> [1,2,3,4,5] -- [2,4]
[1, 3, 5]

iex(5)> [1,2,3,4,4] -- [2,4]
[1, 3, 4]
```

Uma maneira de pegarmos a cabeça da lista, que na verdade é o primeiro elemento, é utilizando a função `hd/1` :

```
iex(6)> hd([1,2,3])
1
```

Já para pegarmos a cauda, que é o restante da lista sem a cabeça, nós usamos a função `tl/1` :

```
iex(7)> tl([1,2,3])
[2, 3]
```

Existem muitos conceitos diferentes aqui, mas não se preocupe, porque vamos aprofundá-los nos próximos capítulos, em que perceberemos o verdadeiro poder dessas listas.

Tuplas

Tuplas são similares a listas e podemos armazenar elementos de qualquer tipo dentro delas. Contudo, escrevemos tuplas entre chaves em vez de parênteses.

A principal diferença entre tuplas e listas é que acessar elementos individuais de uma tupla é mais rápido. Porém, crescer e adicionar mais elementos pode ser custoso com o tempo, coisa que não ocorre sempre com listas.

```
iex(1)> {"livro", 10, true, :funciona?, 1.5}
{"livro", 10, true, :funciona?, 1.5}
```

Existem diversas funções para trabalhar com tuplas e você deve consultar a documentação oficial a cada atualização da linguagem para ficar por dentro das novidades.

Através da função `tuple_size/1` , podemos acessar o tamanho de qualquer tupla. E com a função `elem/2` , podemos

acessar qualquer elemento de uma tupla através de seu índice que sempre começa em 0.

```
iex(2)> tupla = {"livro", 10, true, :funciona?, 1.5}
{"livro", 10, true, :funciona?, 1.5}

iex(3)> tuple_size tupla
5

iex(4)> elem tupla, 2
true

iex(5)> elem tupla, 0
"livro"
```

A função `put_elem/3` permite atualizar um elemento dentro de uma tupla, então vamos supor que temos uma tupla com nome e idade e precisamos atualizar a idade.

```
iex(2)> tupla = {:jhon, 20}
{:jhon, 20}

iex(3)> put_elem tupla, 1, 25
{:jhon, 25}
```

Uma coisa que você deve ter notado nesses exemplos é que podemos executar funções Elixir sem o uso de parênteses. Isso significa que nem sempre precisamos ser tão verbosos, porque Elixir permite que eliminemos coisas desnecessárias, privilegiando a legibilidade do código.

Mapas

Mapas são uma coleção de chave/valor e são similares a tuplas, exceto quando se acrescenta um sinal de `%` (porcentagem) na frente das chaves.

Os valores de mapas podem ser acessados através de suas

chaves, e isso é algo bem similar a um Array em outras linguagens.

Geralmente, as chaves do mapa são do mesmo tipo, mas isso não é obrigatório. Se você usar atoms como chaves de um mapa, você ganha a habilidade de acessar os mapas através de `mapa.chave` , mas terá um erro caso a chave não exista. Além disso, você não pode ter chaves repetidas utilizando mapas.

```
iex(1)> mapa1 = %{"chave1" => "valor1", :chave2 => "valor 2"}
%{:chave2 => "valor 2", "chave1" => "valor1"}

iex(2)> mapa2 = %{chave1: 1, chave2: 2, chave3: "valor 3"}
%{chave1: 1, chave2: 2, chave3: "valor 3"}

iex(3)> mapa3 = %{"repetida" => 1, "repetida" => 2}
%{"repetida" => 2}

iex(4)> mapa1["chave1"]
"valor1"

iex(5)> mapa1["chave2"]
nil

iex(6)> mapa1[:chave2]
"valor 2"

iex(7)> mapa1.chave2
"valor 2"

iex(8)> mapa1.chave3
** (KeyError) key :chave3 not found in: %{:chave2 => "valor 2", "
chave1" => "valor1"}

iex(8)> mapa2[:chave1]
1

iex(9)> mapa2[:chave2]
2

iex(10)> mapa2[:chave3]
"valor 3"
```

```
iex(11)> mapa2.chave1
1

iex(12)> mapa2.chave2
2

iex(13)> mapa2.chave3
"valor 3"

iex(14)> mapa2.chave4
** (KeyError) key :chave4 not found in: %{chave1: 1, chave2: 2, c
have3: "valor 3"}
```

2.4 EXERCÍCIOS

- Experimente abrir o IEx e executar algumas operações com os conceitos estudados;
- Leia a documentação oficial sobre os tipos básicos.

2.5 VOCÊ APRENDEU

- Os fundamentos básicos de Elixir;
- Os tipos mais importantes;
- Algumas funções úteis.

Capítulo 3

ORGANIZAÇÃO

Praticamente todo código escrito em Elixir vive dentro de módulos. Eles são como classes em outras linguagens, e atuam como repositórios de funcionalidades.

Quando você precisar resolver um problema utilizando Elixir, você provavelmente criará um módulo e, dentro dele, criará as funções necessárias para a solução do problema. Então, é bastante comum que tenhamos módulos que solucionam problemas específicos e se comunicam entre si.

Em Elixir, podemos escrever múltiplas funções com o mesmo nome, mesmo número de argumentos e do mesmo tipo! Uma linguagem comum não saberia a diferença entre uma função e outra, mas isso não ocorre nessa linguagem. Isso porque cada função pode ter um padrão na sua assinatura e as chamadas de funções podem ser organizadas por padrão em vez de número de argumentos ou tipos. Isso evita uma avalanche de `if / else` que você teria de escrever somente para garantir que está chamando a função correta.

3.1 PATTERN MATCHING

Reconhecimento de padrões é uma funcionalidade que serve

para decompor estruturas complexas em estruturas mais simples. Em Elixir, até mesmo o que parece ser uma atribuição de valor a uma variável realiza Pattern Matching por trás das cenas. Isso pode soar estranho neste momento, mas veremos exemplos que nos ajudarão a entender melhor este aspecto da linguagem.

Quando fazemos uma atribuição de valor a uma variável, por exemplo, `x = 10`, o que acontece na verdade é uma checagem de padrões. Aqui, não dizemos que o operador `=` seja `igual`, mas sim que estamos tentando fazer `match` (procurando casar) o valor à esquerda do igual com o valor à direita dele — neste caso, casar `x` com `10`.

A maneira correta de ler seria perguntar: `x` *match* (casa com) `10 ?` Vamos executar alguns exemplos no `Iex` para que fique mais claro.

No código a seguir, o que acontece é um `match` de x com 10 que Elixir usa para realizar uma espécie de `bind` (ligar o valor 10 a variável x). É justamente por isso que quando fazemos a operação contrária, ela também funciona porque, como `x = 10`, você também poderia dizer que `10 = x`.

```
iex(1)> x = 10
10

iex(2)> 10 = x
10
```

Se tentarmos casar valores que não batem, receberemos um erro. Vamos tentar uma experiência para fazer exatamente isso.

No exemplo, tentamos casar `11` com `x`, cujo valor já tinha sido ligado antes ao `10`. Como eles não batem, recebemos o erro

de que não houve match.

```
iex(1)> x = 10
10

iex(2)> 10 = x
10

iex(3)> 11 = x
** (MatchError) no match of right hand side value: 10
```

Explicado como funciona o básico do Pattern Matching, podemos avançar para entender quais são as vantagens de ter este recurso disponível. Vamos tornar as coisas um pouco mais divertidas a tentar fazer `match` utilizando `tuplas` .

No código de exemplo, casamos duas tuplas com sucesso, porque temos dois `atoms` nomeados `:carro` em ambos os lados. E como o `bind` acontece entre `modelo` e `Honda` , nós temos armazenado na variável `modelo` o conteúdo `Honda` .

```
iex(1)> {:carro, modelo} = {:carro, "Honda"}
{:carro, "Honda"}

iex(2)> modelo
"Honda"
```

Se tentamos fazer `match` com `atoms` diferentes, eles não casam e temos um erro. Isso também aconteceu antes quando tentamos casar variáveis diferentes.

```
iex(5)> {:carro, modelo} = {:moto, "Honda"}
** (MatchError) no match of right hand side value: {:moto, "Honda
}
```

Também podemos fazer match pulando valores que não nos interessam. Vamos supor que queremos pegar sempre o segundo ou terceiro valor, não importando quais sejam o primeiro ou o

segundo. Para isso, podemos usar um simples `_` (underscore).

```
iex(5)> {_, numero} = {"nao interessa", 500}
{"nao interessa", 500}

iex(6)> numero
500

iex(7)> {_, _, numero} = {1, 2, 500}
{1, 2, 500}

iex(8)> numero
500
```

No processo de `match`, existe um caractere especial `^` (circunflexo), que podemos utilizar para impedir um possível `rebind`. Isto é, o processo de religar um valor a uma variável, o que poderia ser o mesmo valor ou outro valor qualquer.

Vamos ver alguns exemplos nos quais o erro ocorre porque antes o `a` estava ligado ao valor `3` e, com o uso do `^`, dizemos ao Elixir para não fazer o `rebind` para outro valor. Sendo assim, o `a` só pode casar com o valor `3` e, com nenhum outro valor diferente disso, o tornamos completamente imutável.

```
iex(9)> [a, b] = [1, 2]
[1, 2]

iex(10)> a + b
3

iex(11)> [a, b] = [3, 4]
[3, 4]

iex(12)> a + b
7

iex(13)> [^a, b] = [5, 6]
** (MatchError) no match of right hand side value: [5, 6]
```

```
iex(14)> [^a, b] = [3, 6]
[3, 6]

iex(15)> a
3

iex(16)> b
6
```

3.2 MÓDULOS

Módulo é o mecanismo usado para agrupar funções e lógica que você deseja reaproveitar. `List` e `String` são exemplos de módulos nativos e, claro, você pode criar seu próprio módulo. Para isso, basta utilizar a palavra reservada `defmodule` seguida do nome do módulo, por exemplo, `defmodule Multiplicador` .

Você pode criar módulos para armazenar funções úteis que poderiam ser reutilizadas por outros módulos. Vamos então criar o módulo `Multiplicador` e executá-lo no `Iex` para exercitar. Crie um arquivo chamado `multiplicador.ex` com o código de exemplo, abra seu terminal, navegue até o local onde está o arquivo e execute o `Iex` passando o nome do arquivo com a extensão.

No código de exemplo, você cria um módulo com uma função dentro dele que multiplica seus argumentos. Para criar uma função, basta usarmos a palavra reserva `def` , seguida do nome da função e seus argumentos.

Lembre-se também de que, em Elixir, não precisamos retornar nada, pois o próprio código sempre retorna a última linha automaticamente — neste caso, o resultado da multiplicação. Você também pode notar que, para inserir comentários no código, basta acrescentar uma tralha `#` , seguida do comentário.

```
#exemplo de módulo
defmodule Multiplicador do
    def multiplique(a, b) do
        a * b
    end
end

iex  multiplicador.ex
iex> Multiplicador.multiplique(2, 2)
iex> 4
```

Você também poderia criar a mesma função em uma única linha e com parâmetros pré-definidos.

```
#exemplo
defmodule Multiplicador do
  def multiplique(a, b \\ 2), do: a * b
end

iex(1)> Multiplicador.multiplique 4
8

iex(2)> Multiplicador.multiplique 5
10

iex(3)> Multiplicador.multiplique 4, 3
12

iex(4)> Multiplicador.multiplique 5, 3
15
```

Existem ainda **funções privadas**, que são funções que só podem ser chamadas de dentro do módulo no qual foram declaradas. Isto é bastante similar ao que acontece em outras linguagens.

Note o exemplo que declaramos uma função com `defp` em vez de `def` . Isso mostra que ela é privada e restrita às chamadas internas. Perceba também que, quando tentamos executá-la de fora do módulo, recebemos um erro informando que ela é inexistente

ou privada.

```
#exemplo de função privada
defmodule Multiplicador do
  def multiplique(a, b \\ 2), do: soma(10, a * b)
  defp soma(a, b), do: a + b
end

iex(1)> Multiplicador.multiplique 2, 3
16

iex(2)> Multiplicador.multiplique 2, 4
18

iex(3)> Multiplicador.soma 2, 2
** (UndefinedFunctionError) function Multiplicador.soma/2 is unde
fined or private
    Multiplicador.soma(2, 2)
```

Também é possível aninhar módulos, e temos duas maneiras de fazer isso: uma é escrevendo literalmente um módulo dentro do outro, e a outra é usando a notação de pontos. Você pode aninhar quantos módulos forem necessários e a maneira de executá-los é a mesma.

Você pode utilizar isso para evitar conflitos de nomes quando sentir necessidade de criar nomes iguais, ou mesmo para sua própria organização a fim de deixar mais claro o que o módulo pode fazer.

Por exemplo, suponhamos que você precise organizar as finanças por entradas e saídas, e queira uma função chamada `calcular` para ambas, então você faria `Financas.Entradas.calcular` e `Financas.Saidas.calcular`.

Note como ficou bem evidente o que esse módulo de finanças é capaz de fazer. Veja agora alguns exemplos de módulos aninhados:

```
#módulo aninhado
defmodule Multiplicador do
    defmodule Multiplique do
        def por_dois(numero) do
            numero * 2
        end
    end
end

#módulo aninhado com ponto
defmodule Multiplicador.Multiplique do
    def por_dois(numero) do
        numero * 2
    end
end
```

```
iex  multiplicador.ex
iex> Multiplicador.Multiplique.por_dois(2)
iex> 4
```

3.3 FUNÇÕES

Na seção anterior, notamos que, para criar uma função **dentro de um módulo**, bastava utilizarmos a palavra reserva `def`, seguida do nome da função e seus argumentos. Além disso, também podemos criar funções sem argumentos e, inclusive, funções sem nome!

Funções anônimas são úteis quando você precisa passar uma função como argumento, e ela é tão específica e autoexplicativa que não há necessidade de criar um nome para ela.

Note que destaquei o trecho dentro de um módulo para que fique claro que funções com nome só podem existir dentro de um módulo. O Iex, por exemplo, não é um local onde você poderia definir uma função utilizando `def`, sem que crie um módulo para isso.

```
iex(1)> def multiplique(a, b) do a * b end
** (ArgumentError) cannot invoke def/2 outside module
    (elixir) lib/kernel.ex:4297: Kernel.assert_module_scope/3
    (elixir) lib/kernel.ex:3299: Kernel.define/4
    (elixir) expanding macro: Kernel.def/2
             iex:1: (file)
```

Vamos seguir a recomendação e colocar a função dentro de um módulo:

```
iex(1)> defmodule Multiplicador do def multiplique(a, b) do a * b
 end end
iex(2)> Multiplicador.multiplique(2,3)
6
```

Claro que existem outras maneiras de declararmos funções, pois elas são cidadãs de primeira classe em linguagens funcionais. No exemplo a seguir, criamos uma função anônima usando a palavra reservada `fn`, acrescentamos os argumentos `a` e `b`, e um `->` que representa o corpo da função.

Ela é ligada por meio de `match` a uma variável que servirá para executá-la mais tarde. Perceba que, desta vez, usamos um `.` (ponto) para chamar a função por meio de uma variável. Os parênteses também são obrigatórios neste caso, pois esta é a diferença entre funções anônimas e funções nomeadas. Ocorre um erro quando executamos sem o ponto, porque o Elixir não consegue encontrar a definição da função.

```
iex> multiplicador = fn a, b -> a * b end
#Function<12.50752066/2 in :erl_eval.expr/5>

iex> multiplicador.(2, 4)
8

iex> multiplicador.(2, 3)
6

iex> multiplicador(2, 4)
```

```
** (CompileError) iex:4: undefined function multiplicador/2
```

Criar funções anônimas é uma tarefa tão comum que os desenvolvedores por trás de Elixir resolveram criar um atalho para isso. O `&` pode ser usado para criar funções anônimas mais facilmente. Parece confuso de início, mas na verdade é bastante simples.

Basta utilizar o `&` seguido de parênteses com o corpo da função. Seus argumentos são nomeados com `&1`, `&2`, `&3`, e assim por diante na respectiva ordem.

```
iex(1)> soma = &(&1 + &2)
&:erlang.+/2

iex(2)> soma.(2,2)
4

iex(2)> multiplica_e_soma = &(&1 * &2 + &3)
#Function<18.50752066/3 in :erl_eval.expr/5>

iex(3)> multiplica_e_soma.(2,2,1)
5
```

3.4 FUNÇÕES E PATTERN MATCHING

Funções também podem ter várias assinaturas diferentes, e o Elixir consegue fazer isso porque utiliza o reconhecimento de padrões que vimos anteriormente. Neste exemplo, definiremos uma função com múltiplas assinaturas, que reconhecerá o padrão do `atom` que será enviado como argumento e retornará o conteúdo de acordo.

Caso não nos importarmos com o padrão, basta usarmos o *underscore* que atua para pegar qualquer coisa em qualquer padrão. É importante saber que o `_` normalmente é utilizado no

final da expressão, porque a leitura do padrão é realizada de cima para baixo e, se você o colocar no início, ele vai aceitar qualquer padrão, impedindo que os outros sejam reconhecidos.

```
iex(5)> escreva_o_nome = fn
...(5)> :tiago -> "Seu nome é Tiago"
...(5)> :davi  -> "Seu nome é Davi"
...(5)> _ -> "Não me importo qual é o seu nome"
...(5)> end
#Function<6.50752066/1 in :erl_eval.expr/5>

iex(6)> escreva_o_nome.(:tiago)
"Seu nome é Tiago"

iex(7)> escreva_o_nome.(:davi)
"Seu nome é Davi"

iex(8)> escreva_o_nome.(:jonas)
"Não me importo qual é o seu nome"
```

3.5 EXERCÍCIOS

Este foi um capítulo bastante extenso, portanto, sugiro que realize os exercícios para fixar bem o conteúdo, já que ele será necessário nos capítulos seguintes.

- Experimente criar funções de diferentes formas, sejam anônimas, nomeadas e com parâmetros predefinidos;
- Crie módulos simples e aninhados;
- Procure exercitar bastante o Pattern Macthing, tentando casar coisas diferentes.

3.6 VOCÊ APRENDEU

- Módulos;
- Funções;

- Pattern Matching.

Capítulo 4

VERIFICAÇÃO

Neste capítulo, aprenderemos como a linguagem é lida com controle de fluxo. Alguns dos mecanismos serão bem familiares para quem já tem experiência em outras linguagens, mas, para outros, daremos um pouco mais de ênfase por não serem tão comuns.

Também avançaremos para um aspecto que os idealizadores de Elixir chamam de *guard clauses* (cláusulas de guarda ou protetoras) e, finalmente, entenderemos um dos operadores que mais facilitam nossa vida no dia a dia de trabalho com Elixir, o operador `|>` pipe.

4.1 CONTROLE DE FLUXO

Para lidar com controle de fluxo, existem mecanismos tradicionais como o `if/else` . Alguns não tão tradicionais como o `unless` , que é o contrário do `if` , e alguns mais modernos, como o `case` , que lembra bastante um `switch` para quem vem de outras linguagens. É importante saber que esses mecanismos não são construções especiais como na maioria das outras linguagens. Em Elixir, eles são conhecidos como `macros` que são exatamente como qualquer outra função básica.

No código exemplo, realizamos uma verificação simples utilizando o `if/else`.

```
iex(1)> x = 10
10

iex(2)> if x do
...(2)> "x tem valor válido"
...(2)> else
...(2)> "x é nil ou false"
...(2)> end
"x tem valor válido"
```

Claro que você não é obrigado a passar o `else`.

```
iex(1)> x = 10
10

iex(2)> if x == 10 do
...(2)> "x é 10"
...(2)> end
"x é 10"
```

O `unless` é conhecido de quem vem de linguagens como Ruby, e funciona da mesma maneira. Ele verifica se a condição é falsa, ou seja, exatamente o contrário do `if`, que verifica se a condição é verdadeira.

```
iex(3)> unless x == 10 do
...(3)> "x não é 10"
...(3)> else
...(3)> "x é 10 sim"
...(3)> end
"x é 10 sim"
```

Para facilitar ainda mais nossa vida e deixar o código mais limpo, também podemos escrever o `if` em uma única linha. Veja os exemplos.

```
iex(4)> if true, do: 1 + 2
3
```

```
iex(5)> if false, do: :tiago, else: :davi
:davi
```

Quando precisamos checar diversas condições e encontrar a primeira condição válida, nós podemos utilizar a macro `cond`. Veja os exemplos.

```
iex(6)> cond do
...(6)> 2 + 2 == 5 -> "Isso não é verdade"
...(6)> 2 + 2 == 6 -> "Isso também não é verdade"
...(6)> 2 + 2 == 4 -> "Ok, você acertou!"
...(6)> end
"Ok, você acertou!"
```

É importante saber que, se nenhuma das condições for verdadeira, você terá um erro.

```
iex(7)> cond do
...(7)> 2 + 2 == 5 -> "Isso não é verdade"
...(7)> 2 + 2 == 6 -> "Isso também não é verdade"
...(7)> 2 + 2 == 7 -> "Você está louco!"
...(7)> end
** (CondClauseError) no cond clause evaluated to a true value
```

A macro `case` nos permite tentar casar um valor com vários padrões até encontrarmos um que corresponde e faça `match` com sucesso. Ela pode ser usada de diversas maneiras diferentes, com listas, com tuplas e tudo mais; use sua criatividade.

```
iex(7)> case :tiago do
...(7)> :davi   -> "Isso não casa com :tiago"
...(7)> 10      -> "Isso muito menos"
...(7)> :jonas  -> "Estou ficando cansado..."
...(7)> :tiago  -> "Ok, você casou :tiago com :tiago!"
...(7)> end
"Ok, você casou :tiago com :tiago!"
```

```
iex(9)> case 10 do
...(9)> 11  -> "10 não é 11"
...(9)> 12  -> "10 não é 12"
```

```
...(9)> _   -> "10 não é underscore, mas underscore é um coringa
que casa com tudo beleza?"
...(9)> end
"10 não é underscore, mas underscore é um coringa que casa com tu
do beleza?"
```

4.2 GUARD CLAUSES

Cláusulas de guarda nada mais são do que expressões que você implementa para garantir, por exemplo, que determinada função só pode receber um tipo de argumento, ou que seu argumento deve passar por determinada condição para ser válido, e assim por diante. Isso evita que tenhamos de digitar uma série de condicionais para verificar estes detalhes, porque a cláusula de guarda literalmente protege a execução.

No código a seguir, note o uso do `when` (quando). Esta expressão, além de validar a correspondência de padrões, também garante que a variável `x` seja maior do que zero, para que sejam satisfeitas todas as condições.

```
iex(1)> case {1,2,3} do
...(1)> {1,x,3} when x > 0 -> "Isso vai casar porque 2 é maior qu
e zero"
...(1)> _ -> "Isso casaria se não tivesse casado antes"
...(1)> end
"Isso vai casar porque 2 é maior que zero"
```

Você também pode utilizar as guard clauses dentro de funções. Note que, se nenhuma das cláusulas bater, você vai receber um erro, por isso é importante escrever seu código de um jeito que verifique as possíveis exceções. Neste caso, é comum usarmos o coringa *underscore* para casar com todas as outras possibilidades.

```
iex(2)> minha_funcao = fn
...(2)> a, b  when a > 0 and b == 10 -> a + b
```

```
...(2)> a, b  when is_atom(a) -> "#{a} é um atom"
...(2)> end
#Function<12.50752066/2 in :erl_eval.expr/5>

iex(3)> minha_funcao.(1, 10)
11

iex(4)> minha_funcao.(1, 11)
** (FunctionClauseError) no function clause matching in :erl_eval
"-inside-an-interpreted-fun-"/2

iex(5)> minha_funcao.(:tiago, 11)
"tiago é um atom"
```

Existem uma série de funções iniciando com `is_` que você pode utilizar em guard clauses, por exemplo: `is_integer/1`, para aceitar somente números inteiros; ou `is_list/1`, para aceitar somente listas. Você pode usá-las para garantir que sua implementação aceita apenas estes tipos de argumentos, evitando ter de validar esses detalhes de maneira verbosa.

Abra o Iex e digite `is_ + tab` para ver uma lista dessas funções:

```
iex(1)> is_
is_atom/1         is_binary/1       is_bitstring/1    is_boolean/1

is_float/1        is_function/1     is_function/2     is_integer/1

is_list/1         is_map/1          is_nil/1          is_number/1
is_pid/1          is_port/1         is_reference/1    is_tuple/1
```

4.3 OPERADOR PIPE

O operador pipe `|>` é algo bem simples, mas que facilita demais na hora de escrever código Elixir. Basicamente ele pega a saída da expressão da esquerda e passa como primeiro argumento de uma função à direita, e assim por diante. Vamos ver exemplos

sem o operador pipe, e depois com o operador, para que você entenda as claras vantagens de utilizá-lo.

O módulo `Enum` possui as funções `map/2` e `filter/2` que recebem uma coleção como argumento e uma função que é aplicada a cada elemento da coleção. No caso do `map`, ele retorna uma nova coleção depois de aplicar a função a todos os elementos dela, e o `filter` filtra os elementos e retorna na coleção somente aqueles cujas funções passadas para ele retornaram `true` — ou seja, filtrou com sucesso.

```
iex(11)> colecao = [1,2,3]
[1, 2, 3]

iex(12)> Enum.map(colecao, &(&1 * 2))
[2, 4, 6]

iex(13)> Enum.filter(colecao, &(&1 > 2))
[3]

iex(14)> Enum.filter(Enum.map(colecao, &(&1 * 2)), &(&1>4))
[6]
```

Neste exemplo, utilizando o Pipe, perceba que estamos passando somente a função anônima como segundo argumento para o `filter`, porque o operador Pipe automaticamente passou o retorno da função anterior `map` como primeiro argumento para ele.

```
iex(15)> Enum.map(colecao, &(&1 * 2)) |>
...(15)> Enum.filter(&(&1 > 4))
[6]
```

E podemos usar o Pipe quantas vezes forem necessárias, fazendo com que o código fique bem mais limpo e organizado.

```
iex(16)> Enum.map(colecao, &(&1 * 2)) |>
...(16)> Enum.map(&(&1 * 2)) |>
```

```
...(16)> Enum.map(&(&1 * 2))
[8, 16, 24]
```

Suponhamos que você precise mapear, filtrar e somar itens de um carrinho de compras. Vamos utilizar o pipe para facilitar nossa vida.

Criamos uma lista de itens que, supostamente, vieram de uma lista de compras, então aplicamos desconto de 20% em cada um dos produtos, e depois filtramos para pegar somente aqueles que têm valor superior a R$10,00. Em seguida, somamos todos estes valores e mostramos o resultado na tela, tudo utilizando o que aprendemos junto com o pipe.

```
iex(1)> itens = [%{produto: "Tv LG 32 polegadas", valor: 935.50},
%{produto: "Notebook Samsung 1TB", valor: 1599.00}, %{produto: "B
arbeador Gilette", valor: 9.99}]
[%{produto: "Tv LG 32 polegadas", valor: 935.5},
 %{produto: "Notebook Samsung 1TB", valor: 1599.0},
 %{produto: "Barbeador Gilette", valor: 9.99}]

iex(2)> Enum.map(itens, &(Float.round(&1.valor - &1.valor * 0.2, 
))) |>
...(2)> Enum.filter(&(&1 > 10.00)) |>
...(2)> Enum.sum
2027.6
```

4.4 EXERCÍCIOS

- Exercite bastante o uso do operador Pipe e dos condicionais;
- Crie módulos com suas próprias funções e condicionais abordados;
- Leia a documentação oficial sobre os condicionais.

4.5 VOCÊ APRENDEU

- Controle de fluxo;
- Guard clauses;
- Pattern Matching.

Capítulo 5

LISTAS

Neste capítulo, veremos como os loops podem ser implementados de maneira funcional com *list comprehension*. Também aprenderemos como carregar coleções enormes, ou mesmo infinitas, através de *lazy evaluation*. Por fim, vamos abordar um aspecto muito comum em linguagens funcionais, que é a capacidade que uma função tem de executar a si mesma por meio de recursividade.

5.1 HEAD E TAIL (CABEÇA E CAUDA)

Uma lista em Elixir consiste de `head` e `tail`. Logo, `head` seria a cabeça ou o primeiro valor encontrado na lista, e `tail`, a cauda ou a própria lista em si. Isso significa que até mesmo uma lista de um único elemento contém head/tail, sendo que, neste caso, o `head` é o elemento que está na lista e `tail` seria uma lista vazia, já que não existem outros elementos dentro dela.

Uma lista vazia pode ser representada por colchetes vazios, `[]`, como vimos nos capítulos anteriores. E uma lista de um único elemento, como você deve imaginar, seria uma lista contendo um único elemento dentro dela.

Suponhamos que temos uma lista de um único elemento, cujo

elemento é o valor 1, `[1]` . Este único elemento dentro da lista, na verdade, pode ser representado como head/tail, desta maneira, `[ 1 | [] ]` , em que o número 1 seria a cabeça e a cauda seria a lista vazia.

Experimente tentar casar estes valores no Iex para confirmar o que acabou de aprender.

```
iex> [1] = [ 1 | [] ]
[1]

iex> [ 1 | [] ]
[1]
```

Vou deixar seu cérebro ligeiramente confuso acrescentando mais um elemento dentro da lista, mas prometo que, ao final deste capítulo, você entenderá a razão disso tudo.

Você gosta de desafios? Vamos então acrescentar o número 2 na lista, ficando deste modo: `[1,2]` . Logo, sabemos que neste caso a cabeça da lista é o número 1, e a cauda é o restante da lista; portanto, isso pode ser representado como `[ 1 | [ 2 | [] ] ]` .

Note que o que antes era uma lista vazia, depois do número 1, agora é uma lista com o número 2 seguida de uma lista vazia.

```
iex> [1,2] = [ 1 | [ 2 | [] ] ]
[1, 2]

iex> [ 1 | [ 2 | [] ] ]
[1, 2]
```

Acrescentando mais um elemento, podemos notar o padrão head/tail. Logo, `[1,2,3]` poderia ser representado como: `[ 1 | [ 2 | [ 3 | [] ] ] ]` .

Agora, quando você ver uma simples lista `[1,2,3]` , você saberá que na verdade isso significa lista com 1, seguida de pipe lista com 2, seguida de pipe lista com 3, seguida de pipe lista vazia, e assim por diante, dependendo da quantidade de elementos na lista.

```
iex> [1,2,3] = [ 1 | [ 2 | [ 3 | [] ] ] ]
[1, 2, 3]

iex>  [ 1 | [ 2 | [ 3 | [] ] ] ]
[1, 2, 3]
```

Podemos usar o operador pipe para fazer Pattern Matching do head e do tail da lista, sendo que o que fica à esquerda do operado pipe é o head, e o que fica à direita dele é o tail.

```
iex> [head | tail] = [1,2,3,4,5,6,7,8,9,10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

iex> head
1

iex> tail
[2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Você não precisa chamá-los de head/tail, porém é uma convenção e uma boa prática chamar deste modo; a não ser que você chame de algo que represente melhor o contexto onde está sendo aplicado.

```
iex> [nome | jogos] = ["Tiago", "Dragon Age Inquisition", "GTA V"
, "Devil My Cry"]
["Tiago", "Dragon Age Inquisition", "GTA V", "Devil My Cry"]

iex> nome
"Tiago"

iex> jogos
["Dragon Age Inquisition", "GTA V", "Devil My Cry"]
```

5.2 LIST COMPREHENSION

Em Elixir, existe uma maneira funcional de percorrer coleções que denominamos List comprehension. Ela pode ser classificada como a capacidade de gerar coleções a partir de outras coleções, porque é exatamente isso que faz.

Este mecanismo é usado onde você normalmente utilizaria loops ou laços em outras linguagens. Há uma macro chamada `for` que podemos usar para este propósito.

No código a seguir, dado que você entre com uma coleção, a macro percorre os elementos dela multiplicando cada um deles por dois, e cria outra coleção com o resultado.

```
iex> for x <- [1,2,3], do: x * 2
[2, 4, 6]
```

Também podemos aplicar um filtro na macro, fazendo com que somente os elementos que passem no filtro sejam colocados na nova lista que será criada. Nos exemplos, somente atoms e números inteiros estarão presentes na nova coleção.

```
iex> for x <- [1,2,3, :tiago, :davi], is_atom(x), do: x
[:tiago, :davi]

iex> for x <- [1,2,3, :tiago, :davi], is_integer(x), do: x
[1, 2, 3]
```

Você também poderia criar sua própria função customizada para filtrar elementos na macro `for` . Nos exemplos, vamos filtrar somente os estudantes que têm idade igual ou superior a dezoito anos.

```
iex(1)> estudantes = [{"Daniel", 15}, {"Philip", 27}, {"Joana", 1
}]
```

```
[{"Daniel", 15}, {"Philip", 27}, {"Joana", 19}]

iex(2)> maior_de_idade = fn {nome, idade} -> idade >= 18 end
#Function<6.52032458/1 in :erl_eval.expr/5>

iex(4)> for estudante <- estudantes, maior_de_idade.(estudante),
do: estudante
[{"Philip", 27}, {"Joana", 19}]
```

Podemos ainda criar novas estruturas e fazer Pattern Matching. Note que, no lugar do x, realizamos o match com o nome e o aparelho de videogame que está presente na tupla, e ainda conseguimos transformá-lo em um mapa de chave/valor.

```
iex> for x <- [1,2,3, :tiago, :davi], is_integer(x), do: {x, x* 2
}
[{1, 2}, {2, 4}, {3, 6}]

iex> for {nome, aparelho} <- [{"tiago", "PS4"}, {"Davi", "PS3"},
"Jonas", "PS2"}],  do: nome
["tiago", "Davi", "Jonas"]

iex> for {nome, aparelho} <- [{"tiago", "PS4"}, {"Davi", "PS3"},
"Jonas", "PS2"}],  do: aparelho
["PS4", "PS3", "PS2"]

iex> for {nome, aparelho} <- [{"tiago", "PS4"}, {"Davi", "PS3"},
"Jonas", "PS2"}],  do: %{nome: nome, aparelho: aparelho}
[%{aparelho: "PS4", nome: "tiago"}, %{aparelho: "PS3", nome: "Dav
i"}, %{aparelho: "PS2", nome: "Jonas"}]
```

Você pode ainda ser mais ousado e percorrer duas ou mais coleções ao mesmo tempo. Dado que eu tenha duas ou mais coleções, as operações serão aninhadas.

Suponhamos que temos `a <- [1,2], b <- [3,4]` , então `a=1 , b=3 ; a=1 , b=4 ; a=2 , b=3 ; a=2 , b=4 .`

```
iex> for a <- [1,2], b <- [3,4], do: {a,b}
[{1, 3}, {1, 4}, {2, 3}, {2, 4}]
```

```
iex> for a <- [1,2], b <- [3,4], do: a + b
[4, 5, 5, 6]
```

Em todos os nossos exemplos, sempre retornamos uma lista depois da execução do comprehension, mas podemos mudar este resultado com o uso da palavra reservada `into`. Perceba como podemos retornar um mapa em vez de uma lista utilizando este recurso:

```
iex> for {nome, aparelho} <- [{"tiago", "PS4"}, {"Davi", "PS3"},
"Jonas", "PS2"}], into: %{},  do: {nome, aparelho}
%{"Davi" => "PS3", "Jonas" => "PS2", "tiago" => "PS4"}
```

A nova estrutura não precisa estar vazia. Neste caso, a macro vai incluir o que estiver presente na estrutura no resultado final.

```
iex> for {nome, aparelho} <- [{"tiago", "PS4"}, {"Davi", "PS3"},
"Jonas", "PS2"}], into: %{"Pedro" => "PS1"},  do: {nome, aparelho
}
%{"Davi" => "PS3", "Jonas" => "PS2", "Pedro" => "PS1", "tiago" =>
 "PS4"}
```

E podemos utilizar todas as funções que já conhecemos para nos auxiliar:

```
iex> for {nome, aparelho} <- [{"tiago", "PS4"}, {"Davi", "PS3"},
"Jonas", "PS2"}], into: %{},  do: {nome |> String.downcase |> Str
ing.to_atom, aparelho}
%{davi: "PS3", jonas: "PS2", tiago: "PS4"}
```

5.3 LAZY EVALUATION

Carregamento lento (*lazy evaluation*) é um recurso presente na linguagem para evitar carregarmos dados desnecessários em memória. Isso é muito útil para lidar com coleções cujo tamanho não sabemos, ou algum arquivo pesado que não pode ser carregado todo de uma vez.

Já vimos que, em Elixir, existe o módulo `Enum` que é útil para lidar com coleções, porém, apesar de ele ser incrível, ele potencialmente carrega toda a coleção em memória e, consequentemente, devolve uma nova coleção.

Imagine usar este módulo para carregar milhares de registros ou um arquivo de 50GB! Isso provavelmente demorará bastante, ou até mesmo travará a máquina, porque este módulo tem de alocar tudo em memória antes de usar.

Para resolver este problema, existe o módulo `Stream`, que faz o carregamento das informações somente quando elas são necessárias. Ele não devolve uma nova coleção; diferente disso, ele devolve um Stream de dados que representa a informação contida nela. Em Elixir, este recurso é chamado de *lazy evaluation*.

Nada melhor do que utilizar exemplos para representar a diferença entre carregamento rápido e lento. Então, vamos carregar 20 milhões de registros usando o módulo `Enum`, e depois carregar os mesmos 20 milhões utilizando o módulo `Stream`, para observarmos a diferença.

No exemplo, criamos uma coleção que vai de 1 até 20 milhões usando o range do Elixir, basta colocarmos `inicio..fim`, e será criada a coleção que vai do início ao fim que desejamos. Os underscores são permitidos para facilitar a visualização de números grandes.

Execute o exemplo para ver quanto tempo demora para mapear 20 milhões de registros, e depois pegarmos apenas dez deles para mostrar na tela.

```
iex> Enum.map(1..20_000_000, &(&1)) |> Enum.take(10)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Agora realizaremos a mesma operação usando `Stream`. Execute e veja a diferença no tempo de execução que é praticamente imediato.

```
iex> Stream.map(1..20_000_000, &(&1)) |> Enum.take(10)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Vamos entender o que aconteceu. O `Enum.take(10)` pediu somente dez elementos, então a diferença foi que mapear a coleção com `Enum` carregou tudo em memória para depois utilizar somente 10. Já o `Stream` carregou em memória somente o que de fato precisou e utilizou conforme foi exigido dele. Neste caso ele carregou apenas 10 e não os 20 milhões. Além disso, `Enum` devolve uma nova lista e `Stream` devolve um stream de dados.

```
iex> Enum.map([1,2,3], &(&1))
[1, 2, 3]

iex> Stream.map([1,2,3], &(&1))
#Stream<[enum: [1, 2, 3], funs: [#Function<46.89908360/1 in Strea
m.map/2>]]>
```

Você deve se perguntar, já que Stream é melhor do que Enum, então vou utilizar Stream em tudo! Vá com calma gafanhoto, pois ambos os recursos não existem sem motivo. Nem toda iteração precisa de Stream, então considere utilizá-lo somente quando precisar atrasar o carregamento de informações, quando precisar de parte de algo muito maior, ou quando tiver de lidar com arquivos muito grandes.

O uso de Stream consome mais recursos que o uso de Enum, então usar Stream para situações nas quais ele não deveria ser aplicado pode ter o efeito contrário e reduzir em vez de ganhar performance. Não use Stream quando lidar com coleções

pequenas, ou precisar mapear uma lista simples de elementos. Em vez disso, procure utilizar o bom senso antes de decidir por um ou por outro.

5.4 RECURSIVIDADE

Recursividade é a capacidade que uma função tem de executar a si mesma múltiplas vezes, até resolver um determinado problema. Isso parece um pouco confuso, mas na verdade é algo bastante simples que, se bem aplicado, pode ajudar a resolver questões complexas de modo muito mais simples do que seria se fossem feitas de outra forma.

Elixir ajuda bastante na implementação de funções recursivas, pois podemos usar Pattern Matching em conjunto para eliminar boa parte do trabalho pesado. Isso torna as funções muito mais expressivas, fáceis de entender e reutilizar.

Vamos implementar um contador de elementos, coisa que já existe no módulo `Enum`. Contudo, vamos implementar nosso próprio de maneira recursiva. Observe o uso da função `Enum.count/1`, ela serve exatamente para contar quantos elementos existem dentro da coleção passada para ela como argumento.

```
iex> Enum.count [1,2,3]
3
```

Faremos o mesmo. Crie um módulo chamado `Contador` em um arquivo chamado `contador.ex`, com duas versões de uma função chamada `contar` que recebe uma lista como argumento. Na primeira versão, ela reconhece o padrão de uma lista vazia e, se

o padrão for reconhecido, ela retorna `0`.

Na segunda versão, ela reconhece o padrão de uma lista com elementos e utiliza head/tail para pegar o primeiro elemento no head e o restante da lista no tail. Em seguida, esta função executa a si mesma passando como argumento `1 + contar(tail)`.

```
defmodule Contador do
  def contar([]), do: 0
  def contar([head | tail]), do: 1 + contar(tail)
end
```

```
iex contador.ex
iex> Contador.contar []
0

iex> Contador.contar [1,2,3]
3

iex> Contador.contar [1,2,3,4,5]
5
```

A mágica acontece porque, na primeira vez que você chama a função `contar`, você passa para ela `1 + tail (restante da lista)`, sendo que o primeiro item da lista sempre fica armazenado no head. Desta maneira, você remove o primeiro elemento e passa o restante até que a lista esteja completamente vazia, o que executa o primeiro padrão que retorna zero. Logo, ela soma de 1 em 1, até não existirem mais elementos a serem contabilizados.

```
Contador.contar([1,2,3])
= 1 + contar([2,3])
= 1 + 1 + contar([3])
= 1 + 1 + 1 + contar([])
= 1 + 1 + 1 + 0
= 3
```

5.5 EXERCÍCIOS

- Crie um contador recursivo que possa contar listas, mapas e tuplas;
- Itere sobre múltiplas coleções com e sem filtros;
- Carregue as primeiras 100 linhas de um arquivo `txt` com Stream.

5.6 VOCÊ APRENDEU

- O que as listas Elixir são na realidade;
- List comprehension;
- Lazy evaluation;
- Recursividade.

Capítulo 6

CALCULADORA DE MÉDIAS

Neste capítulo, colocaremos a mão na massa para criar uma aplicação do zero e ajudá-lo a fixar o que foi aprendido nos capítulos anteriores. Então, sugiro que tire um tempo para construir esta aplicação sem copiar e colar, pois isso o ajudará a entender melhor os conceitos estudados.

Aqui você também vai aprender, de uma maneira bastante prática, conceitos novos como `defstruct`, por exemplo.

O objetivo desta aplicação será bastante simples. Dado que eu passe uma lista de estudantes com suas notas, ela deve ser capaz de me retornar uma lista dos estudantes com suas médias. Além disso, ela também precisa me informar qual foi a melhor média da classe.

Precisamos de um módulo chamado `Student` com as funções `first_name` e `last_name`, que servirão para retornar respectivamente o nome e o sobrenome do estudante.

Como queremos registrar o resultado das médias, o módulo `Student` conta com uma estrutura padrão que definimos utilizando a palavra-chave `defstruct`, seguida das propriedades que essa estrutura vai conter.

Neste caso, a estrutura contém as propriedades `nome` e `resultados`, que serão criadas por padrão com o valor `nil` e `[]`, respectivamente. Isso significa que, toda vez que criarmos um elemento baseado nesta estrutura, este conterá essas propriedades com esses valores por padrão.

Você também pode especificar outros valores predefinidos para suas estruturas. Isso funcionará como uma classe em outras linguagens, e o elemento que contém a estrutura funcionará como uma instância desta classe.

Para que você entenda o que são valores predefinidos, vamos imaginar que precisamos de um módulo para produtos de uma loja de R$1,99. O valor mínimo de cada produto será predefinido em R$1,99 e, toda vez que criarmos um produto baseado nesta estrutura sem informar o que foi predefinido, ele será preenchido com este valor padrão.

```
// produto.ex

defmodule Produto do
  defstruct nome: nil, valor: 1.99
end

$ iex produto.ex

iex> sabonete = %Produto{nome: "Dove"}
%Produto{nome: "Dove", valor: 1.99}

iex> sabonete
%Produto{nome: "Dove", valor: 1.99}

iex> shampoo = %Produto{nome: "Nivia", valor: 5.50}
%Produto{nome: "Nivia", valor: 5.5}
```

Isso parece muito similar a Orientação a Objetos (OO), mas não fique tentado a sair criando estruturas para tudo quanto é

lado. Lembre-se de que Elixir é uma linguagem funcional e não orientada a objetos, logo, não é pensando em OO que você vai tirar proveito de uma linguagem como essa. Você deve sempre procurar utilizar o bom senso e, se algo parece muito complicado de fazer, é porque provavelmente você deve estar pensando de maneira errada.

Dito isso, vamos em frente para o código do módulo `Student`.

A função `first_name` recebe um estudante, captura seu nome e cria uma lista dividindo-a onde antes havia espaços em branco. A função `split/1` do módulo `String` recebe por padrão uma `String`.

Se você usar a função `h String.split`, vai poder ler a documentação que mostra exemplos de como ela funciona. Neste caso, estou passando uma `String` com duas palavras separadas por um espaço, e o retorno da função é uma lista com as duas palavras dentro.

```
iex> String.split("foo bar")
    ["foo", "bar"]
```

Precisamos capturar o primeiro nome do estudante, então vamos utilizar a função `first/1` do módulo `List`, que recebe uma lista como argumento e retorna o primeiro elemento dessa lista.

```
iex> String.split("foo bar") |> List.first
"foo"
```

Também precisamos capturar o último nome do estudante, então usaremos a função `last/1` do módulo `List`, que recebe

uma lista como argumento e retorna o último elemento dela.

```
iex> String.split("foo bar") |> List.last
"bar"
```

Para escrever a documentação do módulo, podemos utilizar a macro `@moduledoc`. Neste caso, ela espera três aspas duplas com o texto dentro delas. Isso será útil quando executarmos este módulo com a função `h` nativa do Elixir para retornar a documentação dos módulos.

Já a macro `@doc` serve para escrevermos a documentação específica de cada função do módulo e, neste caso, informamos o que a função faz, quais são os argumentos que ela espera, de que tipo eles são e exemplos de uso. Esta documentação também é lida pela função `h`, explicada anteriormente.

Vamos ao código completo do módulo `Student`:

```
// student.ex
defmodule Student do
  @moduledoc """
  Define a Student struct and functions to handle a student.
  """
  defstruct name: nil, results: []

  @doc """
  Get the first name of a Student.

  ## Parameters

  - `student` - A Student struct.

  ## Examples

  *  joao = %Student{name: "João Joaquim"}
  *  Student.first_name(joao)
  *  "João"
  """
  def first_name(student) do
```

```
    student.name
    |> String.split
    |> List.first
  end

  @doc """
  Get the last name of a Student.

  ## Parameters

  - `student` - A Student struct.

  ## Examples

    *  joao = %Student{name: "João Joaquim"}
    *  Student.last_name(joao)
    *  "Joaquim"
  """
  def last_name(student) do
    student.name
    |> String.split
    |> List.last
  end

end
```

Execute o módulo e observe que podemos criar e armazenar a estrutura dentro de uma variável. Neste caso, podemos informar os valores que queremos passar para as propriedades da estrutura utilizando a base previamente criada. Estas estruturas são como `mapas`, a diferença é que precisamos informar o nome do módulo na estrutura.

```
iex student.ex
iex> carlos = %Student{name: "Carlos Joaquim"}
%Student{name: "Carlos Joaquim", results: []}

iex> Student.first_name carlos
"Carlos"

iex> Student.last_name carlos
"Joaquim"
```

O Elixir vai compilar a estrutura para um simples mapa com uma propriedade extra, chamada `__structure__`, por debaixo dos panos. Então, ficaria algo assim:

```
%{
  __structure__: Student,
  name: nil,
  results: []
}
```

Como estamos pensando no contexto de estudantes e médias, e vamos precisar de possíveis matérias, então implementaremos o módulo `Subject` que tem apenas a documentação e uma estrutura que usaremos como base para criarmos estas matérias.

É bastante comum criarmos módulos simples como este em Elixir para armazenar estrutura de dados, pois isso seria muito melhor do que termos uma estrutura sem referência alguma dentro do código.

Vamos ao código completo do módulo `Subject`:

```
// subject.ex
defmodule Subject do
  @moduledoc """
  Define a Subject struct.
  """
  defstruct name: nil, result: nil
end
```

Pronto, já temos os estudantes e as matérias. Agora podemos trabalhar na nossa calculadora.

O módulo `Calculator` é o mais complexo de todos. Ele servirá para iniciar os outros módulos, efetuar todos os devidos cálculos e mostrar o resultado na tela. Vamos aos detalhes do passo a passo.

Este módulo possui também sua devida documentação, apenas uma função pública e algumas funções privadas. Usamos a função `build_subjects` para definir quatro matérias de exemplo. Esta mapeia essas matérias criando um `Subject` para cada uma delas com seu nome e um resultado aleatório que é fornecido por meio da função `Random/1` do módulo `Enum`, passando como argumento uma lista de resultados que está em `results`.

Vamos utilizar resultados aleatórios para que o resultado seja dinâmico e diferente em cada execução. Mas fique à vontade para brincar com estas funções criando mais matérias e diferentes resultados.

Vamos precisar da função `average` para calcular a média. Ela recebe um estudante com suas matérias, conta o total de matérias utilizando `Enum.count`, e armazena o resultado na variável `total`. Depois, a função mapeia os resultados, soma todos eles e reduz a um único resultado, que é a soma de todas as notas de todas as matérias.

A função `Float.ceil` recebe a divisão da soma das notas pelo total de matérias e retorna um ponto flutuante de precisão de duas casas. Isso é útil para quando queremos garantir a precisão das casas decimais que, neste caso, precisa ter no máximo duas depois do ponto.

Precisamos da função `calculate` para transformar o resultado. Ela recebe uma lista de estudantes, mapeia-os para pegar o nome e o sobrenome de cada um deles, e executa a função `average` para calcular a média.

Vamos utilizar a função `start` sem nenhum argumento. Ela

será o ponto de partida para usarmos todas as outras funções e módulos. Nela simplesmente criamos quatro estudantes com notas aleatórias para serem usadas na criação das matérias. Então, passamos uma lista com os quatro estudantes para a função `calculate` e armazenamos o resultado na variável `result`.

Como queremos exibir a melhor média, precisamos de uma função que recebe uma lista de estudantes com suas médias, mapeia para uma lista de médias e armazena a maior na variável `best` utilizando `Enum.max`. Vamos chamá-la de `best_result`

No final, ela filtra os estudantes que possuem a melhor média encontrada e retorna uma lista com eles. Isso é útil pois, se dois ou mais estudantes tiverem médias iguais e ela, por acaso, for a maior, então todos eles serão retornados na lista.

Passamos então o resultado da função `calculate/1` para a função `best_result/1`, para encontrar a lista com os melhores. No fim, retornamos uma tupla com o resultado geral na esquerda e os melhores resultados na direita.

Vamos ao código completo do módulo `Calculator`:

```
// calculator.ex
defmodule Calculator do
  @moduledoc """
  Define a Calculator module and functions to execute it.
  """

  @doc """
  Start the system to process some result.
  """
  def start() do
    joao  = %Student{name: "João Joaquim",   results: build_subje
cts([5.2, 3.9, 2.7, 9.8])}
    maria = %Student{name: "Maria da Silva", results: build_subje
cts([8.4, 7.5, 9.4, 2.8])}
```

```
    pedro = %Student{name: "Pedro Pedrada",  results: build_subje
cts([1.3, 8.5, 8.9, 7.6])}
    kaua  = %Student{name: "Kauã Camboinhas",results: build_subje
cts([5.4, 4.9, 2.2, 3.8])}

    result = calculate([joao, maria, pedro, kaua])
    best = best_result(result)

    {result, best}
  end

  @doc """
  Get all students with their averages.

  ## Parameters

  - `students` - A list of Student structs.
  """
  defp calculate(students) do
    students
    |> Enum.map(&(
     %{
      first_name: Student.first_name(&1),
      last_name:  Student.last_name(&1),
      average: average(&1)
      }
    ))
  end

  @doc """
  Calculate the average of a Student.

  ## Parameters

  - `student` - A Student struct.
  """
  defp average(student) do
    total = Enum.count(student.results)

    result = student.results
    |> Enum.map(&(&1.result))
    |> Enum.reduce(&(&1 + &2))

    Float.ceil(result / total, 2)
  end
```

```
  @doc """
  Get the greater average.

  ## Parameters

  - `students` - A list of students with their averages
  """
  defp best_result(students) do
    best = students
    |> Enum.map(&(&1.average))
    |> Enum.max

    students
    |> Enum.filter(&(&1.average === best))
  end

  @doc """
  Build some subjects and some random results.

  ## Parameters

  - `results` - A list of results
  """
  defp build_subjects(results) do
    subjects = ["Matemática", "Português", "Geografia", "História
"]

    subjects
    |> Enum.map(&(%Subject{name: &1, result: Enum.random(results)
}))
  end

end
```

Para compilar todos os módulos, organize-os dentro de uma pasta e execute `elixirc`, passando os nomes dos módulos para compilar todos de uma vez. Neste caso, seria algo como:

```
// compilando todos os módulos
$ elixirc calculator.ex student.ex subject.ex
```

Depois de compilar todos os módulos, alguns arquivos com

extensão `.beam` serão gerados no diretório. Estes são arquivos de bytecode que a máquina virtual vai executar.

Como todos os módulos estão compilados dentro da mesma pasta, o IEx pode ser chamado dentro desta pasta e ele automaticamente terá acesso a todos eles.

Vamos então executar o *start* principal para vermos o resultado. Faremos Pattern Matching do resultado para uma variável `result` com o resultado geral, e `best` para os melhores.

```
$ iex
$ iex> {result, best} = Calculator.start()

{[%{average: 6.55, first_name: "João", last_name: "Joaquim"},
  %{average: 5.16, first_name: "Maria", last_name: "Silva"},
  %{average: 4.78, first_name: "Pedro", last_name: "Pedrada"},
  %{average: 4.61, first_name: "Kauã", last_name: "Camboinhas"}],
 [%{average: 6.55, first_name: "João", last_name: "Joaquim"}]}

iex> result
[%{average: 6.55, first_name: "João", last_name: "Joaquim"},
 %{average: 5.16, first_name: "Maria", last_name: "Silva"},
 %{average: 4.78, first_name: "Pedro", last_name: "Pedrada"},
 %{average: 4.61, first_name: "Kauã", last_name: "Camboinhas"}]

iex> best
[%{average: 6.55, first_name: "João", last_name: "Joaquim"}]
```

Vamos agora ler a documentação que escrevemos para o módulo `Student`, utilizando a função `h` nativa do Elixir.

```
iex> h Student

                                    Student

Define a Student struct and functions to handle a student.

iex> h Student.first_name

                             def first_name(student)
```

```
Get the first name of a Student.

## Parameters

  • student - A Student struct.

## Examples

  •  joao = %Student{name: "João Joaquim"}
  •  Student.first_name(joao)
  •  "João"
```

Todo o código desta demo pode ser encontrado no meu repositório no GitHub: https://github.com/tiagodavi/livro-elixir-demo-1.

6.1 EXERCÍCIOS

- Experimente melhorar ainda mais estes módulos;
- Crie módulos para resolver um problema qualquer diferente deste;
- Pesquise outras formas de chamar módulos, como `require`, `alias`, `use` e `import`.

6.2 VOCÊ APRENDEU

- Como criar módulos com estruturas predefinidas;
- Como documentar módulos e funções;
- Como compilar e executar módulos separados;
- Como utilizar os recursos aprendidos na prática.

Capítulo 7

MIX

Mix é uma interface de linha de comando (CLI) para construção de projetos, gerenciamento de dependências e execução de tarefas. Com ela, é possível desde criar projetos com toda a estrutura de pastas predefinida até tarefas customizadas.

Creio que não exista maneira melhor de entender Mix do que utilizando-o na prática. Vamos então criar um projeto novo com ele, para que possamos analisar cada arquivo e pasta que ele vai gerar.

O Mix já vem instalado por padrão junto ao Elixir, então apenas digite no terminal `mix new <nome do projeto>`, em que `<nome do projeto>` é o nome que você gostaria de dar ao projeto. Com este comando, o Mix gerará toda a estrutura do projeto para você, com todas as pastas necessárias para executá-lo.

Criar nomes de projetos às vezes é um processo complicado e lento, por isso, costumo simplesmente chamar todos eles de `app` e, futuramente, quando eu souber o nome adequado, coloco o `app` dentro de outra pasta que seria de fato o nome real do projeto. Isso é útil para que você não perca tempo pensando em nomes e comece logo a desenvolver seu projeto.

No exemplo, criarei um projeto chamado `app`. Após executar

o comando, o terminal mostrará as pastas que o Mix criou e informará que é possível utilizar o Mix também para compilar e testar o projeto.

```
$ mix new app

* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/app.ex
* creating test

* creating test/test_helper.exs
* creating test/app_test.exs

Your Mix project was created successfully.
You can use "mix" to compile it, test it, and more:

    cd app
    mix test

Run "mix help" for more commands.
```

Após criá-lo, você pode entrar na pasta do projeto para ver a lista de pastas e arquivos que ele gerou. Vamos fazer um *tour* por esta estrutura para entendê-la melhor. Vamos começar pelo arquivo `README.md` .

Este arquivo descreve o que o projeto faz, como ele o faz e informações necessárias para executá-lo. Este é um típico arquivo usado por sistemas de controle de versão, como `GIT` , para descrever o projeto.

Este arquivo também informa que você pode incluir seu projeto no `HEX` , que é o gerenciador de dependências de Elixir, e ele serviria para mostrar a outros desenvolvedores como colocar

seu projeto como uma dependência externa.

```
//README.md

# App

**TODO: Add description**

## Installation

If [available in Hex](https://hex.pm/docs/publish), the package c
an be installed
by adding `app` to your list of dependencies in `mix.exs`:

def deps do
  [{:app, "~> 0.1.0"}]
end

Documentation can be generated with [ExDoc](https://github.com/el
ixir-lang/ex_doc)
and published on [HexDocs](https://hexdocs.pm). Once published, t
he docs can
be found at [https://hexdocs.pm/app](https://hexdocs.pm/app).
```

O arquivo `mix.exs` gerencia todas as dependências do projeto. É apenas um módulo, e nele é possível incluir dependências que o projeto precisa carregar, além de informar qual a versão do Elixir é necessária para executá-lo.

A função `application` carrega dependências externas necessárias antes de carregar a aplicação principal. Já a função `deps` carrega e faz download de todas as dependências.

```
// mix.exs

defmodule App.Mixfile do
  use Mix.Project

  def project do
    [app: :app,
     version: "0.1.0",
     elixir: "~> 1.4",
```

```
    build_embedded: Mix.env == :prod,
    start_permanent: Mix.env == :prod,
    deps: deps()]
  end

  # Configuration for the OTP application
  #
  # Type "mix help compile.app" for more information
  def application do
    # Specify extra applications you'll use from Erlang/Elixir
    [extra_applications: [:logger]]
  end

  # Dependencies can be Hex packages:
  #
  #   {:my_dep, "~> 0.3.0"}
  #
  # Or git/path repositories:
  #
  #   {:my_dep, git: "https://github.com/elixir-lang/my_dep.git",
 tag: "0.1.0"}
  #
  # Type "mix help deps" for more examples and options
  defp deps do
    []
  end
end
```

A pasta `tests` conterá todos os testes do projeto. Vamos falar sobre testes no próximo capítulo; por enquanto, só esteja ciente que é aqui que você vai criá-los.

```
// app_test.exs

defmodule AppTest do
  use ExUnit.Case
  doctest App

  test "the truth" do
    assert 1 + 1 == 2
  end
end
```

A pasta `lib` conterá todos os arquivos que você vai criar para executar seu projeto. Por enquanto, existe apenas um módulo com o mesmo nome do projeto, porém você customizará este e criará outros módulos conforme desenvolve seu projeto.

```
// app.ex

defmodule App do
  @moduledoc """
  Documentation for App.
  """

  @doc """
  Hello world.

  ## Examples

      iex> App.hello
      :world

  """
  def hello do
    :world
  end
end
```

É uma convenção criar novos módulos dentro de uma pasta com o mesmo nome do projeto, dentro da pasta `lib` . Seus novos módulos ficariam assim: `lib/app/modulo.ex` . Neste exemplo, vamos criar um módulo `Hello` e executá-lo.

Crie uma pasta chamada `app` , dentro da pasta `lib` . Neste caso, chama-se `app` porque seu projeto possui este nome, mas poderia levar qualquer nome; o importante é que seja o nome do seu projeto.

Crie um módulo chamado `hello.ex` , dentro da pasta `app` . Logo, ficaria assim: `lib/app/hello.ex` . O nome do módulo se

chama `App.Hello` para evitar conflitos de nomes, e também porque é uma convenção seguir este padrão.

A função `say` apenas retorna uma string que passamos para ela.

```
// hello.ex

defmodule App.Hello do
  def say(str) do
    str
  end
end
```

Para executar seu projeto, você precisa entrar na pasta dele e digitar `iex -S mix`. Isso vai compilar e executar seu projeto no contexto do IEx.

```
$ iex -S mix

iex> App.Hello.say "Tiago"
"Tiago
```

A pasta `config` contém toda a configuração do projeto. Existe um arquivo previamente criado com informações básicas de configuração.

```
// config.exs

# This file is responsible for configuring your application
# and its dependencies with the aid of the Mix.Config module.
use Mix.Config

# This configuration is loaded before any dependency and is restr
icted
# to this project. If another project depends on this project, th
is
# file won't be loaded nor affect the parent project. For this re
ason,
# if you want to provide default values for your application for
# 3rd-party users, it should be done in your "mix.exs" file.
```

```
# You can configure for your application as:
#
#     config :app, key: :value
#
# And access this configuration in your application as:
#
#     Application.get_env(:app, :key)
#
# Or configure a 3rd-party app:
#
#     config :logger, level: :info
#

# It is also possible to import configuration files, relative to
this
# directory. For example, you can emulate configuration per envir
onment
# by uncommenting the line below and defining dev.exs, test.exs a
nd such.
# Configuration from the imported file will override the ones def
ined
# here (which is why it is important to import them last).
#
#     import_config "#{Mix.env}.exs"
```

Podemos usar este arquivo de configuração para passar argumentos para toda aplicação. Isso seria útil, por exemplo, para verificarmos se a aplicação está em ambiente de teste ou de produção.

Vamos criar uma configuração personalizada e acessar dentro da aplicação que criamos anteriormente. Para isso, basta chamar a macro `config`, seguida dos argumentos que queremos utilizar.

```
// config.exs
use Mix.Config

config :app, prefix: "Hello "
```

Vamos utilizar esta configuração em nossa aplicação,

concatenando a configuração com o argumento da função, para que possamos visualizar o retorno.

```
// hello.ex
defmodule App.Hello do
  def say(str) do
    Application.get_env(:app, :prefix) <> str
  end
end

$ iex -S mix
iex> App.Hello.say "Tiago"
"Hello Tiago"
```

Chamar `Application.get_env(:app, :prefix)` é uma coisa grande e repetitiva demais, então para resolver este problema, podemos criar uma variável no escopo do módulo. Estas variáveis de módulo podem ser criadas em qualquer módulo e são pré-fixadas com `@` .

```
// hello.ex
defmodule App.Hello do

  @prefix Application.get_env(:app, :prefix)

  def say(str) do
    @prefix <> str
  end
end

$ iex -S mix
iex> App.Hello.say "Tiago"
"Hello Tiago"
```

A macro `config` também pode ser usada com três argumentos. O primeiro continua sendo o nome da aplicação, já o segundo é o nome do módulo que queremos configurar, e o terceiro é o prefixo com a configuração que já vimos anteriormente.

```
// config.exs
use Mix.Config

config :app, App.Hello, prefix: "Hello "
```

Neste caso, para chamar a configuração, precisamos informar o nome da aplicação, seguida do nome do módulo e uma lista com o nome da configuração. Podemos utilizar a variável `__MODULE__` que contém o nome do módulo atual, então `__MODULE__ = App.Hello` .

```
// hello.ex
defmodule App.Hello do

  @prefix Application.get_env(:app, __MODULE__)[:prefix]

  def say(str) do
    @prefix <> str
  end
end

$ iex -S mix
iex> App.Hello.say "Tiago"
"Hello Tiago"
```

Geralmente, precisamos executar nossa aplicação em diferentes ambientes. Então, para isso, podemos criar configurações separadas para cada um deles. Vamos criar uma configuração para o ambiente de testes, outra para o ambiente de desenvolvimento e outra para o de produção.

Crie os arquivos `test.exs` , `dev.exs` e `prod.exs` , dentro da pasta `config` , e cada um deles receberá a sua configuração personalizada. Não esqueça de remover a configuração anterior que criamos no arquivo principal para não conflitar com as novas configurações. Também descomente a seguinte linha `import_config "#{Mix.env}.exs"` , para que o Mix possa

importar as configurações personalizadas.

```
// test.exs
use Mix.Config
config :app, App.Hello, prefix: "Test: "

// dev.exs
use Mix.Config
config :app, App.Hello, prefix: "Dev: "

// prod.exs
use Mix.Config
config :app, App.Hello, prefix: "Prod: "
```

Para executar, é bastante simples. Basta executar o Mix, passando como argumento o ambiente em que ele será executado.

```
$ MIX_ENV=test iex -S mix

iex> App.Hello.say "tiago"
"Test: tiago"

$ MIX_ENV=dev iex -S mix

iex> App.Hello.say "tiago"
"Dev: tiago"
```

A pasta `build` contém todos os arquivos compilados que serão executados pela máquina virtual.

7.1 EXERCÍCIOS

- Execute este projeto em ambiente de produção `MIX_ENV=prod` e pesquise sobre o resultado;
- Pesquise como criar tarefas usando Mix;
- Leia a documentação oficial do Mix e do HEX.

7.2 VOCÊ APRENDEU

- O que é Mix;
- Quais arquivos são gerados pelo Mix e para que servem;
- Como criar projetos utilizando Mix;
- Como executar o Mix no contexto do IEX.

Capítulo 8

EXUNIT

Testes automatizados são códigos que verificam se o código de um projeto de software funciona corretamente. Você poderia tentar testar manualmente executando função por função e avaliando os resultados, mas obviamente essa solução é impraticável.

É muito difícil testar manualmente, tendo em vista que o código evolui e muda constantemente. Você perderia um tempo enorme, além de que poderia facilmente esquecer de testar alguma coisa, correndo o risco de introduzir falhas graves.

Hoje não há mais espaço para criar código sem testes. Está mais do que provado que testes são fundamentais para qualquer projeto de software. Os testes são tão importantes que existem diversas variações e até mesmo técnicas de engenharia de software baseada em testes, como por exemplo, o TDD (*Test Driven Development*) e o BDD (*Behavior Driven Development*).

O tipo mais comum de teste é o unitário, e o Elixir automaticamente instala uma biblioteca conhecida como `ExUnit` que podemos utilizar para escrevê-los. Ela é a principal ferramenta para escrever testes automatizados com Elixir.

Vamos construir uma calculadora utilizando BDD, Mix e

ExUnit. Optei por uma calculadora porque será mais fácil identificar o resultado esperado pelos cálculos, e isso trará mais foco para os testes do que para a implementação da solução em si.

Para que possamos testar o projeto mais facilmente, precisamos criá-lo usando o Mix. Então, crie uma pasta chamada `calculadora` e, dentro dela, crie um projeto chamado `app`.

```
$ mkdir calculadora
$ cd calculadora
$ mix new app && cd app
```

A estrutura de pastas do Mix foi explicada no capítulo anterior, então agora focaremos somente nas pastas `test` e `lib`, que são onde criaremos os testes e a implementação do projeto, respectivamente. Ao criar o projeto com o Mix, ele gera automaticamente dois arquivos dentro da pasta `test`: `app_test.exs` e `test_helper.exs`.

Observe que a extensão destes arquivos de teste é `.exs`, e não `.ex`. Isso significa que eles não serão compilados pela máquina virtual. Arquivos `.exs` são uma espécie de arquivo script que são apenas interpretados pela máquina, e isso é útil pois a máquina virtual não precisa recompilar os testes toda vez que eles mudam.

No `test_helper.exs`, temos apenas uma linha que inicia o ExUnit:

```
//test_helper.exs
ExUnit.start()
```

O `app_test.exs` é o teste de exemplo propriamente dito. Ele é apenas um módulo comum que faz uso de outro módulo chamado `ExUnit.Case`. A chamada deste módulo dentro do teste dispara uma macro que permite escrever testes e asserts de forma

amigável.

Na verdade, até mesmo os operadores como o sinal de `=` são macros entendidas como ferramentas de teste no ExUnit. E isso você provavelmente não vai ver em nenhuma outra linguagem.

Para descrever os testes, precisamos de uma maneira de incluir textos com a descrição deles, e isso é feito através da macro `test`. Logo, simplesmente executamos a macro, seguida da descrição do teste.

Para validar o teste, usamos a macro `assert`, seguida de outras macros que, neste caso, são os operadores de `+` e `==`. Existem outras macros que utilizaremos no decorrer deste capítulo, por enquanto procure entender apenas que estes operadores também são macros.

Basicamente, este teste está verificando se a soma de `1 + 1 = 2`. Se isso for verdade, o teste vai passar; caso contrário, falhará.

```
//app_test.exs
defmodule AppTest do
  use ExUnit.Case
  doctest App

  test "the truth" do
    assert 1 + 1 == 2
  end
end
```

Para executar o teste, simplesmente execute `mix test` dentro da pasta do projeto. Observe o resultado que diz que há dois testes com zero falhas.

```
app $mix test

Compiling 1 file (.ex)
```

```
Generated app app
..

Finished in 0.06 seconds
2 tests, 0 failures

Randomized with seed 449098
```

Desenvolveremos algumas funcionalidades nesta calculadora utilizando BDD. Logo, vamos escrever os testes antes do código. BDD significa *Behavior Driven Development* (Desenvolvimento Guiado por Comportamento), e basicamente é um processo de desenvolvimento que serve para guiar nosso código até um nível de qualidade que você talvez não pudesse alcançar sem usar a técnica.

BDD foi originalmente concebido em 2003, por Dan North, como uma resposta ao TDD (desenvolvimento guiado por testes). É uma abordagem que funciona muito bem com uma metodologia ágil, encorajando desenvolvedores, pessoas de qualidade, e não técnicas de negócios em um projeto de software. Em BDD, cada cenário é um exemplo escrito para ilustrar um aspecto específico de comportamento da aplicação.

Sendo assim, remova o código de teste de exemplo para que possamos testar a calculadora. O primeiro teste é bem simples, consiste apenas em verificar se a calculadora pode multiplicar um número por outro.

```
// test/calculator_test.exs
defmodule App.CalculatorTest do
  use ExUnit.Case

  test "should return 4 when multiply 2 by 2" do
    assert App.Calculator.multiply(2, 2) == 4
  end
end
```

```
$ mix test

1) test should return 4 when called with 2 and 2 (AppTest)

     test/calculator_test.exs:4
     ** (UndefinedFunctionError) function App.Calculator.multiply/
 is undefined (module App.Calculator is not available)
     stacktrace:
       App.Calculator.multiply(2, 2)
       test/app_test.exs:5: (test)


Finished in 0.03 seconds
1 test, 1 failure
```

Como você pode notar, o teste falhou porque nem mesmo existe o módulo `Calculator`. Então, precisamos fazer o mínimo necessário para que ele passe. Este é o conceito de fazer com que os testes guiem o desenvolvimento da implementação do código.

Existe um conceito chamado **red - green - refactor**, que significa: crie o teste antes de tudo e o faça falhar, em seguida faça ele passar com o mínimo necessário e depois refatore o código de forma que o teste continue passando. É importante seguir todos os passos para escrever código de qualidade guiado por testes.

O teste informa que não temos o módulo `Calculator`, então precisamos criá-lo, e depois criar a função `multiply` para resolver o teste. Fazer o mínimo necessário para que este teste passe seria simplesmente retornar o número 4. Este é um conceito que os testadores chamam de passos de bebê (*baby steps*), em que você vai incrementando as funcionalidades aos poucos.

```
defmodule App.Calculator do
  def multiply(a, b) do
    4
  end
```

```
end

$ mix test

Finished in 0.02 seconds
1 test, 0 failures

Randomized with seed 573370
```

Vamos então criar outro teste para avaliar se o módulo realmente funciona:

```
test "should return 6 when multiply 2 by 3" do
  assert App.Calculator.multiply(2, 3) == 6
end

$ mix test

1) test should return 6 when multiply 2 by 3 (App.CalculatorTest)
     test/calculator_test.exs:8
     Assertion with == failed
     code:  App.Calculator.multiply(2, 3) == 6
     left:  4
     right: 6
     stacktrace:
       test/calculator_test.exs:9: (test)

Finished in 0.03 seconds
2 tests, 1 failure
```

Desta vez, um teste passou e o outro falhou, porque o resultado da função passando 2 e 3 não é 6, e sim 4. Vamos corrigir a implementação da função e rodar o teste novamente.

```
defmodule App.Calculator do
  def multiply(a, b) do
    a * b
  end
end

$ mix test

Finished in 0.03 seconds
```

```
2 tests, 0 failures
```

Desta vez, ambos os testes passaram, e agora parece que nossa implementação funciona como esperado. Vamos criar outros testes para verificar o que acontece se multiplicarmos um número por vazio.

```
test "should return nil when multiply 2 by nil" do
  assert App.Calculator.multiply(2, nil) == nil
end

test "should return nil when multiply nil by 2" do
  assert App.Calculator.multiply(nil, 2) == nil
end
```

```
$ mix test

1) test should return nil when multiply nil by 2 (App.CalculatorT
est)
   test/calculator_test.exs:16
     ** (ArithmeticError) bad argument in arithmetic expression
     stacktrace:
       (app) lib/app/calculator.ex:3: App.Calculator.multiply/2
       test/calculator_test.exs:17: (test)

2) test should return nil when multiply 2 by nil (App.CalculatorT
est)
   test/calculator_test.exs:12
   ** (ArithmeticError) bad argument in arithmetic expression
   stacktrace:
     (app) lib/app/calculator.ex:3: App.Calculator.multiply/2
     test/calculator_test.exs:13: (test)

Finished in 0.04 seconds
4 tests, 2 failures
```

Neste caso, a implementação não funciona e não é capaz de lidar com argumentos vazios. Faremos então uma refatoração para resolver isso. Se o argumento `a` ou `b` forem vazios, então retorna vazio; caso contrário, faz a operação normalmente.

```
defmodule App.Calculator do
  def multiply(a, b) do
   cond do
      is_nil(a) || is_nil(b) -> nil
      true -> a * b
    end
  end
end

$ mix test

Finished in 0.04 seconds
4 tests, 0 failures
```

Agora sim nosso código funciona até mesmo quando passamos argumentos vazios. Caso precisemos descrever ainda mais o teste, podemos passar uma string para ela ser utilizada em caso de erro no teste.

Vamos escrever um teste errado para exemplificar o uso dela. Observe que a string que usamos aparece no erro para nos ajudar a entender o teste.

```
test "should return 10 when multiply 2 by 4" do
  assert App.Calculator.multiply(2, 4) == 10, "This is an error!!"

end

$ mix test

1) test should return 10 when multiply 2 by 4 (App.CalculatorTest
)
   test/calculator_test.exs:20
   This is an error!!
   stacktrace:
    test/calculator_test.exs:21: (test)
```

Existe uma macro chamada `refute`, que faz exatamente o contrário do `assert`. Ela é útil para testar resultados que são falsos. Vamos utilizá-la para corrigir o teste anterior.

```
test "shouldn't return 10 when multiply 2 by 4" do
  refute App.Calculator.multiply(2, 4) == 10, "This is an error!!"

end

$ mix test

Finished in 0.05 seconds
5 tests, 0 failures
```

Como o `refute` é o contrário do `assert`, ele basicamente seria usado em situações nas quais o resultado negativo ou falso é o que você espera como resultado válido. Ele é uma alternativa para a negação de código com `!`.

Geralmente, precisamos compartilhar variáveis entre diferentes testes para evitar repetição de código. Para fazer isso, basta criar uma função chamada `setup` que retorna uma tupla com um atom `ok` e a variável que você deseja compartilhar.

No teste, podemos extrair a variável usando Pattern Matching no segundo argumento:

```
setup do
  {:ok, tiago: 2, davi: 6}
end

test "should return 4 when multiply 2 by 2", %{tiago: valor} do
  assert App.Calculator.multiply(2, valor) == 4
end

test "should return 6 when multiply 2 by 3", %{davi: valor} do
  assert App.Calculator.multiply(2, 3) == valor
end

$ mix test

Finished in 0.05 seconds
5 tests, 0 failures
```

Para carregar os testes mais rapidamente, é possível executá-los de forma assíncrona em um processo paralelo. Para isso, basta passar o argumento `async: true`.

Esta opção, apesar de deixar os testes mais rápidos, não vem ativada por padrão, porque caso você esteja testando serviços externos com um banco de dados, por exemplo, os resultados podem ser inesperados devido à natureza assíncrona do teste. Então, você deve utilizar esta opção somente quando tiver certeza de que não está testando serviços como estes.

```
defmodule App.CalculatorTest do
  use ExUnit.Case, async: true
```

ExUnit também permite classificar os testes com `tags`. Estas são úteis caso você precise executar um conjunto de testes enquanto mantém outros inativados. Imaginemos que seus testes cresceram e agora você tem dezenas deles, mas quer executar apenas um para ter mais foco, ou somente alguns que são mais importantes. Nestes casos, você faria uso destas tags.

Vamos criar alguns exemplos. A `tag` pode ser colocada em um módulo inteiro ou em funções específicas. Neste exemplo, o `@moduletag` classifica o módulo inteiro como `:math` e, usando esta `tag`, podemos controlar a execução do módulo como um todo.

```
defmodule App.CalculatorTest do
  use ExUnit.Case, async: true

  @moduletag :math
```

Para funções específicas, simplesmente colocamos uma `@tag` em cima da função.

```
@tag :negative
test "should return nil when multiply 2 by nil" do
  assert App.Calculator.multiply(2, nil) == nil
end

@tag :negative
test "should return nil when multiply nil by 2" do
  assert App.Calculator.multiply(nil, 2) == nil
end
```

As `tags` são bem flexíveis e podemos usá-las de várias formas.

```
mix test --exclude tag # Exlui todos que contém esta tag
mix test --include tag # Inclui somente os que contém esta tag
mix test --only tag    # Executa somente os que contém esta tag

$ mix test --only negative

Including tags: [:negative]
Excluding tags: [:test]

Finished in 0.05 seconds
5 tests, 0 failures, 3 skipped
```

8.1 EXERCÍCIOS

- Crie outro projeto e teste-o utilizando outras funções de testes e tags;
- Leia a documentação do ExUnit.

8.2 VOCÊ APRENDEU

- O que são testes automatizados;
- Como criar e executar testes automatizados com Elixir e ExUnit;
- O que é e para que serve a técnica BDD;

- Como classificar testes.

Capítulo 9

INTRODUÇÃO A PROCESSOS

Processos em Elixir são abstrações autocontidas. Isso significa que eles são únicos e isolados. Isto é, as implementações e o controle de estado dentro de cada processo permanecem lá e nada fica exposto, mesmo quando estes processos são executados ao mesmo tempo ou em diferentes máquinas.

O contexto de cada processo é 100% isolado de outros processos, e eles só podem se comunicar através de mensagens. Quando falamos de processos dentro da máquina virtual do Erlang, nos referimos a processos internos geridos pela linguagem e que vivem dentro da máquina virtual, e não a processos do sistema operacional onde ela executa.

Existe uma diferença entre processos do sistema operacional e os da máquina virtual. Os processos do sistema operacional são controlados pelo próprio sistema operacional, seja Windows, Linux, MacOS etc., e geralmente cuidam de alocação de memória e diversas outras tarefas de baixo nível. Estes também costumam ser pesados, difíceis de controlar e não se comunicam entre si.

Por outro lado, os processos da máquina virtual (BEAM) do

Erlang são geridos internamente pela própria máquina. São leves, fáceis de controlar, isolados, se comunicam entre si e não é incomum rodarmos milhares deles ao mesmo tempo em apenas uma máquina virtual, pois ela naturalmente lida com isso de forma bastante eficiente. Podemos dizer que não custa caro levantar processos dentro da VM.

Existem alguns mecanismos nativos que permitem gerenciar processos dentro de Elixir. Eles são tão simples que acho que devem causar até um pouco de inveja em outras linguagens.

Para que você possa criar processos, a maneira mais simples é usando a função `spawn`. Esta tem duas versões, que são `spawn/1` e `spawn/3`. A `spawn/1` recebe apenas um argumento, que é uma outra função para ser executada dentro de um processo e, ao ser chamada, retorna um `PID`, que é o identificador do processo.

A versão `spawn/3` é um pouco mais sofisticada, e pode receber um módulo, um atom que representa uma função dentro do módulo e uma lista de argumentos que essa função pode receber. Ela também retorna o `PID` ao ser executada.

Vamos dar uma olhada na documentação destas funções para que possamos entendê-las melhor. Abra o Iex e digite `h spawn/1` e `h spawn/3`, para que possamos abrir suas documentações, com exemplos de uso.

```
//spawn/1

def spawn(fun)

Spawns the given function and returns its PID.

Check the Process and Node modules for other functions to handle
processes,
```

```
including spawning functions in nodes.

The anonymous function receives 0 arguments, and may return any v
alue.

Inlined by the compiler.

## Examples
current = self()
child   = spawn(fn -> send current, {self(), 1 + 2} end)

receive do
{^child, 3} -> IO.puts "Received 3 back"
end

//spawn/3

def spawn(module, fun, args)

Spawns the given module and function passing the given args and r
eturns its
PID.

Check the Process and Node modules for other functions to handle
processes,
including spawning functions in nodes.

Inlined by the compiler.

## Examples
   spawn(SomeModule, :function, [1, 2, 3])
```

Simplificando tudo, a função `spawn/1` cria um processo para uma função, e `spawn/3` também cria um processo para uma função. Entretanto, a diferença é que esta função está dentro de um módulo que devemos especificar, e você também pode especificar os argumentos que a função dele recebe.

Dentro do contexto do Iex, existe uma função chamada `self`, que nada mais é do que uma referência ao `PID` do próprio Iex. Por meio deste `PID`, poderíamos mandar uma mensagem para o

processo que cuida do Iex, por exemplo.

Imagine este `PID` como um CEP que cada processo tem. Caso você precise enviar uma carta ao processo, você precisa saber qual é o `PID` dele.

A função `self()`, executada dentro do Iex, retorna seu `PID`. O número deste `PID` varia de máquina para máquina. No momento, meu `PID` seria algo como:

```
iex> self()
#PID<0.79.0>
```

Para que possamos criar alguns processos isolados, vamos utilizar o `spawn/1`, que retorna uma mensagem para o processo do Iex que é o próprio terminal onde estamos executando o código. Ao executar o Iex, ele automaticamente cria um processo cujo `PID` pode ser encontrado pela função `self`.

Quando usamos `spawn`, estamos basicamente criando outro processo isolado, mas que ainda assim pode conversar com o processo do Iex através de mensagens. Sempre que precisarmos criar processos para isolar estado, ou executar funcionalidades em paralelo, estas funções serão bem úteis.

A função `send` pode enviar uma mensagem para outro processo. Já a macro `receive` espera casar um padrão de resposta para processar uma possível mensagem recebida.

No exemplo, a variável `pid_do_iex` armazena o `PID` do processo do Iex, chamando a função `self()` no contexto do Iex. A variável `pid_do_processo` armazena o `PID` de outro processo isolado que estamos criando com `spawn/1`.

Dentro deste processo que estamos criando, passamos uma função anônima que chama `send`, enviando uma mensagem de volta para o `pid_do_iex`. Nesta mensagem, estamos enviando uma tupla com `self()` — que, dentro do contexto do `spawn`, equivale ao `PID` deste novo processo interno, e não mais ao `PID` do Iex — e uma mensagem de texto para mostrar que isso está sendo chamado em outro processo.

A macro `receive` chamada dentro do Iex espera por uma mensagem dentro do processo do Iex que case com algum padrão recebido. No caso, a mensagem casa, e é executada e exibida na tela.

Se executássemos o `receive` dentro de outro processo que não fosse o Iex, ele funcionaria da mesma maneira. A única diferença seria que outro processo estaria esperando por uma mensagem, e não necessariamente o processo do IEx.

O importante a entender aqui é que cada processo tem seu próprio `PID` e que eles são únicos. O `self()` retorna o `PID` do processo e, quando chamado de dentro do Iex, é o `PID` do processo do Iex. Já quando chamado em outro processo, é então o `PID` deste outro. A função `send` envia mensagens para processos, e a macro `receive` recebe mensagens de processos.

```
$ iex
iex> pid_do_iex = self()
#PID<0.79.0>

iex> pid_do_processo = spawn fn -> send pid_do_iex, {self(), "Est
e é outro processo, veja o PID como muda"} end
#PID<0.117.0>

iex> receive do
...> {pid_do_processo, mensagem} -> IO.puts " #{mensagem}"
```

```
...> end
 Este é outro processo, veja o PID como muda
:ok
```

Se você enviar uma mensagem para um processo que não tem a função `receive`, este simplesmente não será capaz de ouvir nenhuma mensagem, nem responder a elas. Por isso é importante que utilize a função `receive` em conjunto com a função `send`.

Para que você possa entender melhor o uso de processos e ver como eles se comunicam, vamos criar um projeto de ping pong, de forma que, quando enviarmos a mensagem `ping` para o processo `pong`, ele tem de responder `pong`. E quando enviarmos a mensagem `pong` para o processo `ping`, ele tem de responder `ping`.

Crie então a pasta `ping_pong` e, dentro dela, um projeto chamado `app`.

```
$ mkdir ping_pong
$ mix new ping_pong/app
$ cd ping_pong/app
```

Dentro da pasta `lib`, vamos criar a pasta `app` e, dentro da pasta `app`, vamos criar os módulos `ping.ex` e `pong.ex`.

O módulo `App.Ping` tem duas funções simples. A função `start()` executa a função `wait()`, que é apenas uma macro `receive` que espera uma tupla com `PID` e um atom `:pong`. Se o `receive` casar com sucesso, ele vai enviar de volta para o `PID` (que mandou `:pong`) uma mensagem contendo `self()`, que como sabemos retorna seu `PID` interno e um `:ping`. Logo em seguida, ele escreve na tela `Recebi um Pong` para sabermos o que aconteceu.

Um detalhe importante é que a função `wait()` executa a si mesma no final. Essa é a estratégia recursiva que usamos para reexecutar o bloco `receive`, para que o processo possa esperar por mais uma mensagem.

Isso é necessário porque, depois de receber uma mensagem, o processo simplesmente morreria, já que não teria mais nada para processar. Mas como ele se autoexecuta, ele ficará vivo esperando novas mensagens chegarem.

```
defmodule App.Ping do

  def start() do
    wait()
  end

  def wait() do
    receive do
      {pid, :pong} ->
        send(pid, {self(), :ping})
        IO.puts "Recebi um Pong"
    end
    wait()
  end
end
```

Já podemos testar este módulo enviando uma mensagem para ele com a função `spawn/3`. Esta função, como foi mencionado, espera três argumentos, que são o nome do módulo, um atom que representa qual função do módulo que queremos executar, e uma lista que seria os argumentos que esta função recebe. Em nosso caso, a função `start` não recebe nenhum argumento, então simplesmente passamos uma lista vazia.

A função então retorna o `PID` do processo que acabamos de criar, e podemos agora mandar uma mensagem para ele. Como ele espera receber um `:pong`, é exatamente esta a mensagem que

enviamos para ele e podemos ver o resultado na tela.

```
$ iex -S mix

iex> pid = spawn(App.Ping, :start, [])
#PID<0.119.0>

iex> send(pid, {self(), :pong})
Recebi um Pong
{#PID<0.117.0>, :pong}
```

O módulo `App.Pong` é quase igual ao módulo `App.Ping`, a única diferença aqui é que ele espera receber um `:ping` e envia de volta um `:pong`.

```
defmodule App.Pong do

  def start() do
    wait()
  end

  def wait() do
    receive do
      {pid, :ping} ->
        send(pid, {self(), :pong})
        IO.puts "Recebi um Ping"
    end
    wait()
  end
end
```

Vamos agora construir um módulo `App.Table` para jogar ping pong com os dois módulos que criamos anteriormente.

O módulo `App.Table` tem uma função `start()` que, quando executada, cria um processo para o `ping` e outro para o `pong`. Logo em seguida, ela envia uma mensagem para `ping`, dizendo que quem enviou a mensagem foi `pong`, passando `:pong`.

Quando o módulo `App.Ping` recebe a mensagem, ele responde de volta com `:ping`. E como o módulo `App.pong` também responde de volta com `:pong`, o que acontece é que eles ficam jogando ping pong eternamente e precisaremos fechar o terminal para acabar com o jogo deles!

```
defmodule App.Table do
  def start() do
    ping = spawn(App.Ping, :start, [])
    pong = spawn(App.Pong, :start, [])

    send(ping, {pong, :pong})
  end
end

$ $iex -S mix
$ App.Table.start
# Recebi um Pong
# Recebi um Ping
# ...
```

9.1 EXERCÍCIOS

- Crie outro projeto e tente criar processos utilizando `spawn`;
- Pesquise como saber se um processo está vivo ou não;
- Pesquise o que acontece se um processo receber uma mensagem diferente da qual ele espera e como resolver isso usando o que você já sabe;
- Pesquise como ligar um processo ao outro, de forma que, se um morrer, o outro também morre.

9.2 VOCÊ APRENDEU

- O que são processos;

- Como criar e trocar mensagens entre processos utilizando `spawn` ;
- Como manter um processo vivo indefinidamente através de recursão.

Capítulo 10

PROGRAMAÇÃO CONCORRENTE E PARALELA

A indústria de hardware não produz mais processadores com um único núcleo de processamento. Hoje ela está mais interessada em saber quantos cores (núcleos) consegue encaixar dentro de um único chip. No passado, eram comuns máquinas conhecidas como Dual Core (dois núcleos), porém hoje já existem máquinas com 4, 8 e até mesmo 12 Cores. Múltiplos Cores eram disponíveis apenas para supersservidores, mas hoje eles podem ser encontrados até mesmo em um simples telefone celular.

Esta alta capacidade de processamento em paralelo é um dos principais problemas que a maioria das linguagens tenta resolver. Isso porque, na maioria dos casos, simplesmente não é possível lidar com isso de maneira adequada, ou geralmente é muito difícil gerir programação paralela de forma a utilizar todos os cores disponíveis.

Elixir nos ajuda a resolver este problema, porque opera sobre uma máquina virtual naturalmente projetada para lidar com concorrência. Com isso, podemos criar programas realmente

concorrentes e paralelos, extraindo o máximo de potencial do hardware onde o programa está sendo executado.

Execução paralela é um pouco diferente de execução concorrente. Paralelismo significa que duas ou mais tarefas são livres para executar ao mesmo tempo. Também não há necessidade de interromper nenhuma delas para que outras possam prosseguir, salvo em casos em que isso é feito de forma intencional.

Execução concorrente é quando duas ou mais tarefas também parecem ser executadas ao mesmo tempo, mas o processador poderia estar rapidamente trocando o foco entre elas, interrompendo uma para que outras possam continuar. Ele faz essa troca de forma tão rápida que tudo parece estar sendo executado ao mesmo tempo, mas na verdade não está.

O processador de um único núcleo é um bom exemplo de processador que age trocando as tarefas tão rapidamente que parece que são executadas ao mesmo tempo. Entretanto, na verdade, ele cria cache de uma, executa outra, interrompe uma, executa outra, e assim por diante.

Quando adicionamos mais núcleos a este processador, ele estará apto a executar processos em paralelo. Mas ainda assim ele poderá continuar executando todas as tarefas concorrentemente se os núcleos extras não forem usados.

10.1 CRIANDO UM APP DE CLIMA

Neste capítulo, vamos utilizar tudo que aprendemos para criar uma aplicação projetada para executar em paralelo de forma eficiente. Este projeto será desenvolvido em duas etapas. Na

primeira, sem processos em paralelo, para que possamos demonstrar o problema que isso traz; e na segunda, será feito um refactoring para executar a mesma aplicação em paralelo.

O projeto será uma aplicação de clima que deve consultar uma API externa e trazer informações sobre o clima de uma ou mais cidades em paralelo. O primeiro passo então é criar uma conta no site que fornecerá o serviço de consulta aos dados de clima. Este fornecerá um token que será usado para efetuar as consultas na API. Todo o processo de cadastro é bem simples e totalmente gratuito.

Basta criar uma conta no site `https://openweathermap.org/appid` . Após efetuar o login, procure pela sua APIKey que será usada na aplicação. Geralmente, demora cerca de 10 minutos para que a APIKey esteja ativa e pronta para executar requisições.

Vamos criar uma pasta chamada `clima` e, dentro dela, como é de praxe, uma aplicação chamada `App` .

```
$ mkdir clima && cd clima
$ mix new app && cd app
$ mkdir lib/app
```

Usaremos BDD para guiar o desenvolvimento desta aplicação, então vamos começar escrevendo primeiro os testes, para depois implementar as funcionalidades em si.

Primeiro testaremos se a aplicação é capaz de retornar um endpoint correto ao passar uma localização como argumento. Este endpoint nada mais é do que o formato correto da URL que a aplicação precisará para consultar o serviço de clima.

```
// weather_test.exs

defmodule App.Weather.Test do
  use ExUnit.Case, async: true

  @api "http://api.openweathermap.org/data/2.5/weather?q="

  test "should return a encoded endpoint when take a location" do
    appid = App.Weather.get_appid()
    endpoint = App.Weather.get_endpoint("Rio de Janeiro")

    assert "#{@api}Rio%20de%20Janeiro&appid=#{appid}" == endpoint
  end
end
```

Então executaremos o teste no qual podemos verificar que não existe um módulo chamado `App.Weather`, e muito menos as funções `get_endpoint` e `get_appid`.

```
$ mix test

1) test should return a encoded endpoint when take a location (Ap
.WeatherTest)
 test/weather_test.exs:6
 ** (UndefinedFunctionError) function App.Weather.get_appid/0
 is undefined (module App.Weather is not available)
 stacktrace:
  App.Weather.get_appid()
  test/weather_test.exs:7: (test)

Finished in 0.04 seconds
1 test, 1 failure
```

Para resolver este problema, precisamos criar o módulo e as funções necessárias para que ele funcione, e isso é o que vamos fazer em seguida.

Crie um módulo chamado `App.Weather` dentro da pasta `app`. Usaremos este módulo para comunicar com a API externa.

O módulo `App.Weather`, por enquanto, possui duas funções:

`get_endpoint`, que codifica e retorna a URL da API no formato correto; e `get_appid`, que simplesmente retorna a chave necessária para consultar o serviço — aquela que você criou no site `https://openweathermap.org`.

```
// weather.ex

defmodule App.Weather do

  def get_appid() do
    "8961657a9594868f8a4e77babe8db1f7"
  end

  def get_endpoint(location) do
    location = URI.encode(location)
    "http://api.openweathermap.org/data/2.5/weather?q=#{location}
&appid=#{get_appid()}"
  end

end
```

Ao rodar o teste, podemos perceber que agora passa com sucesso.

```
$ mix test
Compiling 1 file (.ex)
Generated app app
.
Finished in 0.03 seconds
1 test, 0 failures
```

Vamos criar outro caso de teste. Desta vez, dado que eu passe uma temperatura no formato Kelvin, que é o formato que o website retorna, precisamos converter para o formato Celsius, que é o usado no Brasil.

```
// weather_test.exs

test "should return Celsius when take Kelvin" do
  kelvin_example  = 296.48
```

```
  celsius_example = 23.3
  temperature = App.Weather.kelvin_to_celsius(kelvin_example)

  assert temperature == celsius_example
end

$ mix test

1) test should return Celsius when take Kelvin (App.Weather.Test)
     test/weather_test.exs:13
     ** (UndefinedFunctionError) function App.Weather.kelvin_to_c
elsius/1 is
     undefined or private
     stacktrace:
       (app) App.Weather.kelvin_to_celsius(296.48)
       test/weather_test.exs:15: (test)

Finished in 0.04 seconds
2 tests, 1 failure
```

Para resolver o problema, precisamos criar a função que faz a conversão. A função simplesmente subtrai 273.15 do Kelvin passado e arredonda para encontrar o equivalente em Celsius.

```
// weather.ex

def kelvin_to_celsius(kelvin) do
  (kelvin - 273.15) |> Float.round(1)
end

$ mix test
Compiling 1 file (.ex)
..
Finished in 0.04 seconds
2 tests, 0 failures
```

Vamos agora criar o teste da função principal que de fato se conecta ao serviço para buscar as informações sobre a temperatura do local. O primeiro teste espera receber de volta o local que passamos, junto com sua temperatura. Para isso, precisamos checar se o local está contido na string de retorno com

`String.contains?/2`.

O segundo teste nos garante que, caso o local informado não exista, receberemos de volta o local seguido de `not found`.

```
// weather_test.exs

test "should return temperature when take a valid location" do
  temperature = App.Weather.temperature_of("Rio de Janeiro")

  assert String.contains?(temperature, "Rio de Janeiro") == true
end

test "should return not found when take an invalid location" do
  result = App.Weather.temperature_of("00000")

  assert result == "00000 not found"
end
```

```
$ mix test

1) test should return not found when take an invalid location (Ap
p.Weather.Test)
     test/weather_test.exs:27
     ** (UndefinedFunctionError) function App.Weather.temperature
_of/1 is undefined or private
     stacktrace:
       (app) App.Weather.temperature_of("00000")
       test/weather_test.exs:28: (test)

2) test should return temperature when take a valid location (App
.Weather.Test)
     test/weather_test.exs:21
     ** (UndefinedFunctionError) function App.Weather.temperature
_of/1 is undefined or private
     stacktrace:
       (app) App.Weather.temperature_of("Rio de Janeiro")
       test/weather_test.exs:22: (test)

Finished in 0.05 seconds
4 tests, 2 failures
```

Precisamos implementar as funções para que o teste passe. A

função `temperature_of` utiliza `get_endpoint` que já implementamos para pegar a URL do serviço. Depois, ela passa a URL para o `HTTPoison`, que é uma biblioteca externa para realizar requisições HTTP e, em seguida, passa a resposta do `HTTPoison` para a função `parser_response`.

A função `parser_response` espera receber uma tupla com `:ok` e um mapa do `HTTPoison` com status de retorno `200`, o que significa que o serviço retornou com sucesso. Em seguida, ela pega o corpo da resposta HTTP e converte para `json`, para computar a temperatura.

A outra assinatura de `parser_response` espera receber qualquer outra coisa e, por isso, utilizamos o coringa `_`. Caso isso aconteça, simplesmente entendemos que se trata de um erro e mandamos um atom `:error` de volta como retorno.

A função `compute_temperature` espera receber o `json` que foi criado anteriormente. Caso o receba, simplesmente convertemos a temperatura que está neste `json` de Kelvin para Celsius, e a retornamos em uma tupla com `:ok`; caso contrário, retornamos um atom `:error`.

```
// weather.ex

def temperature_of(location) do
  result = get_endpoint(location) |> HTTPoison.get |> parser_resp
onse
  case result do
    {:ok, temp} ->
      "#{location}: #{temp} °C"
    :error ->
      "#{location} not found"
  end
end
```

```
defp parser_response({:ok, %HTTPoison.Response{body: body, status
_code: 200}}) do
  body |> JSON.decode! |> compute_temperature
end

defp parser_response(_), do: :error

defp compute_temperature(json) do
  try do
    temp = json["main"]["temp"] |> kelvin_to_celsius
    {:ok, temp}
  rescue
    _ -> :error
  end
end
```

Como `HTTPoison` e `Json` são bibliotecas externas, precisamos instalá-las antes de podermos usá-las. Para isso, basta declarar as dependências no arquivo `mix.exs`, responsável por isso, como dito em capítulos anteriores.

```
// mix.exs

# Configuration for the OTP application
#
# Type "mix help compile.app" for more information
def application do
  # Specify extra applications you'll use from Erlang/Elixir
  [extra_applications: [:logger, :httpoison]]
end

defp deps do
  [
    {:httpoison, "~> 0.9.0"},
    {:json, "~> 0.3.0"}
  ]
end
```

Após isso, precisamos baixar e compilar as dependências. O `Mix` pode nos auxiliar nesta tarefa com o comando `mix deps.get`.

Caso não tenhamos o `Hex` , que é necessário para realizar esta tarefa, o `Mix` vai informar isso na tela. Basta confirmar com `Y` que tudo será compilado e instalado automaticamente.

```
$mix deps.get
Could not find Hex, which is needed to build dependency :httpoiso

Shall I install Hex? (if running non-interactively, use: "mix loc
al.hex --force") [Yn] Y
```

Ao executar o teste novamente, podemos notar que todos eles passam com sucesso.

```
$ mix test
....
Finished in 0.9 seconds
4 tests, 0 failures
```

Vamos abrir o Iex e visualizar o que implementamos até agora.

```
$ iex -S mix

iex> App.Weather.temperature_of "Rio de Janeiro"
"Rio de Janeiro: 25.4 °C"

iex> App.Weather.temperature_of "Cuiaba"
"Cuiaba: 25.0 °C"

iex> App.Weather.temperature_of "Brazil, Cuiaba"
"Brazil, Cuiaba: 25.0 °C"

iex> App.Weather.temperature_of "Brazil, Manaus"
"Brazil, Manaus: 23.0 °C"

iex> App.Weather.temperature_of "Brazil, Brasilia"
"Brazil, Brasilia: 21.0 °C"

iex> App.Weather.temperature_of "00000"
"00000 not found"
```

Parabéns! Você criou um app de clima completamente funcional utilizando Elixir e ExUnit! Contudo, este app ainda não

está preparado para executar em paralelo, e este é próximo assunto que vamos estudar.

10.2 EXECUTANDO O APP DE CLIMA EM PARALELO

O app que você criou anteriormente é muito legal, mas tem um problema. Imagine que você criou este app como um serviço no seu website, e que você precisa consultar os dados de temperatura de uma lista de cidades.

Todos os dias, pessoas de diferentes cidades se cadastram no seu site para utilizar seu serviço e a lista de cidades cresce conforme o número de cidades novas são cadastradas. Imagine que, em um dia normal de operação do seu website, você precise consultar 10, 100, 500, 1.000 ou até mais cidades de diferentes partes do mundo, ao mesmo tempo.

Vamos criar uma lista de cidades, percorrê-las, passá-las para o app processar e trazer de volta a temperatura de cada uma delas.

```
$ iex -S mix

iex> cities = ["Rio de Janeiro", "Niteroi", "Sao Paulo", "Porto A
legre"]
["Rio de Janeiro", "Niteroi", "Sao Paulo", "Porto Alegre"]

iex> cities |> Enum.map(&(App.Weather.temperature_of(&1)))
["Rio de Janeiro: 28.6 °C", "Niteroi: 28.5 °C", "Sao Paulo: 26.4
°C",
 "Porto Alegre: 18.4 °C"]
```

O problema com a solução anterior é que há um desperdício de tempo entre uma consulta e outra. Conforme a lista cresce, o tempo entre as chamadas cresce junto, fazendo com que a solução

se torne cada vez mais ineficiente até o ponto de demorar minutos, ou até horas, para entregar uma temperatura.

Cada vez que uma consulta é realizada, a próxima consulta acontece somente depois que a consulta anterior foi completada. Isso faz com que o processador fique ocioso aguardando pelas respostas entre as chamadas, para então começar a processar novas requisições.

Vamos utilizar o que aprendemos para refatorar o app. Faremos com que ele utilize processamento em paralelo e deixe de esperar por uma resposta para começar a processar outras.

Neste refactoring, nossa intenção é fazer com que o app possa consultar todas as cidades que ele puder ao mesmo tempo, fazendo com que o potencial da máquina seja aproveitado ao máximo e o app tenha um tempo de resposta mais eficiente.

É importante notar que as requisições que são feitas no serviço de clima não dependem umas das outras. Isso significa que você pode colocar cada uma delas para executar em paralelo dentro de um processo isolado.

Desta vez, explicarei o código dentro do próprio código para facilitar o entendimento do que está acontecendo internamente.

```
def start(cities) do
  # Cria um processo da função manager inicializando com
  # uma lista vazia e o total de cidades.

  # O manager fica "segurando" o estado da lista vazia e do total
 de cidades.

  # __MODULE__ se refere ao próprio módulo em que estamos no mome
nto.
  manager_pid = spawn(__MODULE__, :manager, [[], Enum.count(citie
```

```
s)])

  # Percorre a lista de cidades e cria um processo para cada uma
com a função get_temperature().

  # Envia uma mensagem para este processo passando a cidade e o P
ID do manager.
  cities |> Enum.map(fn city ->
    pid = spawn(__MODULE__, :get_temperature, [])
    send pid, {manager_pid, city}
  end)
end

def get_temperature() do
  # Recebe o PID do manager e a cidade.

  # Envia uma mensagem de volta ao manager com a temperatura da c
idade.

  # O coringa entende qualquer outra coisa como um erro.

  # Chama get_temperature() no final para o processo continuar vi
vo e esperando por mensagens.
  receive do
    {manager_pid, location} ->
      send(manager_pid, {:ok, temperature_of(location)})
    _ ->
     IO.puts "Error"
  end
  get_temperature()
end

def manager(cities \\ [], total) do
  # Se o manager receber a temperatura e :ok a mantém em uma list
a (que foi inicializada como vazia no início).

  # Se o total da lista for igual ao total de cidades avisa a si
mesmo para parar o processo com :exit.
  # Se receber :exit ele executa a si mesmo uma última vez para p
rocessar o resultado.
  # Ao receber o atom :exit para o processo, ordena o resultado e
 o mostra na tela.

  # Caso não receba :exit executa a si mesmo de maneira recursiva
 passando a nova lista e o total.
```

```
  # O coringa no final executa a si mesmo com os mesmos argumento
s em caso de erro.
  receive do
    {:ok, temp} ->
      results = [ temp | cities ]
      if(Enum.count(results) == total) do
         send self(), :exit
      end
      manager(results, total)
    :exit ->
      IO.puts(cities |> Enum.sort |> Enum.join(", "))
    _ ->
      manager(cities, total)
  end
end

iex> cities = ["Rio de Janeiro", "Niteroi", "Sao Paulo", "Porto A
legre"]
["Rio de Janeiro", "Niteroi", "Sao Paulo", "Porto Alegre"]

iex> App.Weather.start cities
[{#PID<0.176.0>, "Rio de Janeiro"}, {#PID<0.176.0>, "Niteroi"},
 {#PID<0.176.0>, "Sao Paulo"}, {#PID<0.176.0>, "Porto Alegre"}]
Niteroi: 32.7 °C, Porto Alegre: 20.8 °C, Rio de Janeiro: 32.7 °C,
Sao Paulo: 30.8 °C
```

Parabéns! Você escreveu seu primeiro app em paralelo utilizando Elixir, e agora ele já é plenamente capaz de seguir em frente para temas mais avançados usando a linguagem Elixir.

O código-fonte deste projeto se encontra em: https://github.com/tiagodavi/livro-elixir-demo-2.

10.3 EXERCÍCIOS

- Crie outro projeto que rode em paralelo para resolver outro problema;
- Descubra como poderia fazer as coisas de maneira mais fácil com GenServer, Agent ou Task;

- Pesquise mais sobre OTP e Elixir.

10.4 VOCÊ APRENDEU

- Programação concorrente e paralela;
- Os problemas que podem ocorrer ao não fazer uso de paralelismo;
- Como criar um projeto que roda em paralelo.

Capítulo 11

TASKS

Neste capítulo veremos como podemos simplificar ainda mais o app de clima utilizando `Tasks`. Em Elixir/Erlang, existem diversas abstrações úteis para gerir processos que visam facilitar ainda mais a vida do desenvolvedor. Uma destas abstrações é o módulo `Task` que pode simplificar e muito os atos de criar, executar e capturar a resposta de um processo.

Criar um processo utilizando `Task` é bastante simples. Primeiro, você deve utilizar `Task.async` para criar uma Task. Ela recebe como argumento uma função que retorna alguma coisa. Ao fazer isso você já criou um processo para essa função sem ter que utilizar `Spawn` e nem `Receive`! Depois basta executar `Task.await` passando a Task que você acabou de criar para pegar a resposta do processo, que no caso é a resposta da função que estava encapsulada em um processo. Viu como é simples?

Exercite os comandos no Iex.

```
iex> task = Task.async(fn -> { :ok, "Elixir" } end)
%Task{owner: #PID<0.159.0>, pid: #PID<0.182.0>, ref: #Reference<0
.0.5.808>}

iex> Task.await task
{:ok, "Elixir"}
```

Com isso você já deve imaginar que é bastante possível

criarmos diversos processos com `Task.async` e utilizar o `Task.await` para capturar a resposta de todos eles sem fazer muito esforço. É exatamente isso que vamos fazer com o nosso app de clima para simplificá-lo.

```
// simple_weather.ex
defmodule App.SimpleWeather do

  def start(cities) do
    cities #-> Recebe uma lista de cidades
    |> Enum.map(&create_task/1) #-> Cria uma Task para cada uma d
elas
    |> Enum.map(&Task.await/1) #->  Processa a resposta de cada T
ask
  end

  defp create_task(city) do
    #-> Cria uma Task com a temperatura da cidade informada
   Task.async(fn -> temperature_of(city) end)
  end

  #-> Restante do código permanece o mesmo

  defp temperature_of(location) do
    result = get_endpoint(location) |> HTTPoison.get |> parser_re
sponse
    case result do
      {:ok, temp} ->
        "#{location}: #{temp} °C"
      :error ->
        "#{location} not found"
    end
  end

  defp get_endpoint(location) do
    location = URI.encode(location)
    "http://api.openweathermap.org/data/2.5/weather?q=#{location}
&appid=#{get_appid()}"
  end

  defp get_appid() do
    "8961657a9594868f8a4e77babe8db1f7"
  end
```

```
  defp parser_response({:ok, %HTTPoison.Response{body: body, stat
us_code: 200}}) do
    body |> JSON.decode! |> compute_temperature
  end

  defp parser_response(_), do: :error

  defp compute_temperature(json) do
    try do
      temp = json["main"]["temp"] |> kelvin_to_celsius
      {:ok, temp}
    rescue
      _ -> :error
    end
  end

  defp kelvin_to_celsius(kelvin) do
    (kelvin - 273.15) |> Float.round(1)
  end

end

iex> App.SimpleWeather.start ["Rio", "Amazonas", "Brasilia", "Sao
 Paulo"]
["Rio: 15.0 °C", "Amazonas: 31.5 °C", "Brasilia: 26.0 °C", "Sao P
aulo: 28.0 °C"]
```

Você acabou de simplificar ainda mais o app de clima utilizando `Task` !

O código-fonte deste projeto se encontra em: https://github.com/tiagodavi/livro-elixir-demo-2.

11.1 EXERCÍCIOS

- Tente utilizar GenServer em vez do Task para resolver o mesmo problema;
- Descubra se faria sentido utilizar o módulo Agent neste caso.

11.2 VOCÊ APRENDEU

- Como utilizar Task para gerir processos;
- Como refatorar um projeto que utiliza Spawn para utilizar Task.

Capítulo 12

CONCLUSÃO

Chegamos ao fim de uma longa jornada, mas seu aprendizado não deve parar por aqui. Agora que você aprendeu sobre os fundamentos de Elixir, você já tem base suficiente para seguir os estudos em OTP, GenServer, Supervisor, Protocolos, Metaprogramação e outros, pois eles utilizarão os conceitos aqui demonstrados.