




Programação Funcional com Elixir



Lazy Evaluation (Enum.take e Stream)

Lazy Evaluation (Enum.take e Stream)

- Vamos começar falando sobre o Enum.take
`https://hexdocs.pm/elixir/Enum.html#take/2`
- O take permite que peguemos uma certa quantidade de elementos de uma coleção.

```
range = 1..9
```

```
Enum.take(range, 3)
```

Lazy Evaluation (Enum.take e Stream)

- Ok, agora que já conhecemos o take, vamos imaginar a seguinte situação...
- Imagine que vc precisa gerar um range de 5 milhões de itens...

```
range = 1..5_000_000
```

Lazy Evaluation (Enum.take e Stream)

- Até aí tudo parece normal, visto que o range em si é apenas uma estrutura que informa o valor inicial e final, mas, e se a gente precisar passar por cada um deles com o map e ao final pegar apenas 10 elementos?

```
Enum.map(range, &(&1)) |> Enum.take(10)
```

Lazy Evaluation (Enum.take e Stream)

- Perceba que levou um tempo para que ele pudesse carregar tudo em memória e só depois efetuar o take.
- Agora imagine isso em uma aplicação real, carregando um arquivo com 50GB de linhas ou algo neste sentido.

Lazy Evaluation (Enum.take e Stream)

- É fácil perceber que isso vai se tornar um gargalo em algum momento da aplicação, então, para isso podemos nos utilizar da Lazy Evaluation (carregamento lento / interpretação lenta), que nada mais é que do que instruções que serão executadas apenas quando ela for necessária.

Lazy Evaluation (Enum.take e Stream)

- Lazy Evaluation nos permite trabalhar com coleções na qual não sabemos o tamanho ou ainda coleções gigantes que não podem ser trabalhadas de uma única vez.
- Até agora vimos que o módulo Enum trabalha com coleções, mas para esses casos citados o Elixir nos entrega o módulo Stream.

Lazy Evaluation (Enum.take e Stream)

- A ideia por trás do módulo Stream é que existem situações como que precisaremos trabalhar com o infinito... isso mesmo, imagine um servidor web que recebe requisições/conexões o tempo todo, ou mesmo um console que precisa gerir os jogadores e suas ações, ou seja, nessas situações precisamos trabalhar com uma estrutura que manipule fluxo de dados que muitas vezes não possuem um fim e é isso que o módulo Stream faz.

Lazy Evaluation (Enum.take e Stream)

- Partindo do princípio do nosso último exemplo, podemos usar agora o módulo Stream para contornar o problema do tempo desperdiçado...

```
range = 1..5_000_000  
Stream.map(range, &(&1)) |> Enum.take(10)
```

- Podemos perceber que foi muito mais rápido o resultado.

Lazy Evaluation (Enum.take e Stream)

- O entendimento da operação anterior é simples. Quando usamos o Enum.map, tradicionalmente o Elixir carregou em memória cada um dos elementos do range e depois aplicou o Enum.take.
- Já usando Stream o Elixir carregou apenas os elementos necessários para que o Enum.take funcionasse.

Recursão

- Assim, aprendemos o último princípio do paradigma funcional.
 - First-Class functions ✓
 - Pure functions ✓
 - Immutable variables ✓
 - Recursion ✓
 - ✓ **Nonstrict evaluation**
 - Statements ✓
 - Pattern matching ✓