DESCOBRINDO ELIXIR

Conceitos de programação funcional

Ao falarmos de programação funcional, automaticamente nos recordamos de alguns conceitos chaves que fazem com que sua aplicação se torne poderosa. Esses conceitos são a imutabilidade dos dados, funções puras, função sem efeito colateral, entre muitos outros.

Imutabilidade dos dados significa nunca alterar o estado de um objeto. Assim, sempre que uma “variável” precisa ter seu valor alterado, uma cópia dela é criada e seu valor original é mantido.

Uma boa analogia para essa característica da programação funcional é pensar em um código funcional como uma roda de pessoas brincando de telefone sem fio. A primeira pessoa tem uma palavra, um dado, que ela passa para a próxima, a segunda pega esta frase, faz alguma alteração nela e passa para a seguinte a nova frase alterada, e assim se sucede até o final da roda. Se pensarmos nas pessoas como funções que recebem um dado de entrada e utilizam esse dado com alguma alteração como parâmetro para outra função que o usa como entrada, temos um algoritmo em programação funcional separado por várias funções. Cada função recebe um dado de entrada, faz algo muito específico com ele e retorna um novo dado. Dessa maneira, as funções recebem um dado de entrada e alteram ele dentro da própria função, de modo que o dado fora da função não é alterado e, durante toda a execução do programa, o dado inicial não será alterado. Esse é o conceito chave de valor imutável: funções geram novos dados a partir de um parâmetro de entrada, sempre mantendo o valor inicial da entrada inalterado.

Um algoritmo funcional geralmente é composto por funções que recebem como entrada o retorno de outra função. Cada função tem um objetivo muito claro, o que torna essas funções mais previsíveis. Por exemplo, em uma função que multiplica uma entrada por 2, o resultado será sempre o mesmo quando a entrada for a mesma, de forma que o programador consegue saber exatamente o comportamento dela. Tendo funções puras com objetivos simples e claros, temos um código divido em várias funções que são previsíveis. Como cada função não muda o dado de entrada, dizemos que elas não têm **efeito colateral**.

Dizemos que uma função possui efeito colateral quando alguma variável de estado definida fora da função tem seu valor alterado. As operações que geram efeitos colaterais são: modificar uma variável não local, modificar o valor de um argumento passado por referência, executar entrada ou saída de dados e chamar outras funções que possuam efeitos colaterais.

No fim das contas, qual a finalidade de todos esses conceitos e propriedades?
Tudo isso facilita a programação paralela e a programação distribuída. Como hoje em dia os processadores das nossas máquinas possuem vários núcleos (e podemos ainda criar programas que são executados por vários processadores), não faz mais sentido rodar um programa grande usando apenas um núcleo de processamento. Para conseguirmos usar

todo o poder da máquina podemos distribuir pedaços do código para serem rodadas paralelamente em processadores diferentes, o que aumenta o desempenho do programa. Se temos funções puras que recebem um dado e retornam outro, consequentemente não teremos variáveis sendo alteradas ao longo da execução. Assim, a programação funcional traz uma grande facilidade para executar diferentes funções em diferentes processadores.

Essas características mostram o poder que a programação funcional possui. Um ótimo exemplo de linguagem que utiliza as propriedades da programação funcional é o Elixir. Criado por um brasileiro, a linguagem surgiu em 2012 e em apenas dois anos de existência já ocorreu a primeira conferência internacional sobre a linguagem. Elixir foi construída através de Erlang que já existia na época e usava BEAM VM.

À primeira vista, temos a impressão de que o uso de variáveis imutáveis pode gerar um gasto muito grande de memória, visto que toda vez que desejamos atribuir outro valor para uma variável, é criada uma cópia dela. Assim os programas, mesmo sendo rápidos, teriam um gasto gigante de memória. No entanto, essa questão não chega a ser um problema pois o Elixir usa o gerenciamento de memória do Erlang que tem garbage collector, um jeito inteligente e eficiente de se livrar dessas cópias que seriam criadas. E claro que ficar atribuindo novos valores a variáveis em elixir não é uma boa prática de programação. Se recebemos um valor podemos simplesmente passar o retorno de uma função para outra função.

Exemplos de código

- **Função pura**

O exemplo abaixo traz uma função pura, pois é retornada a soma dos dois argumentos sem que esses argumentos sejam alterados.

```
1  defmodule Math do
2    def sum(x, y) do
3      x + y
4    end
5  end
```

- **Função não pura**

Já este exemplo representa uma função não pura, já que o estado da saída padrão é alterada

```elixir
defmodule Nome do
  def imprime_nome(nome) do
    IO.puts(nome)
  end
end
```

- **Imutabilidade**

Para mostrar a imutabilidade em Elixir temos esse exemplo:

```
iex(1)> x=10
10
iex(2)> defmodule Math do
...(2)> def multiply(a,b) do
...(2)> a*b
...(2)> end
...(2)> end
{:module, Math,
 <<70, 79, 82, 49, 0, 0, 5, 104, 66, 69, 65, 77, 65, 116, 85, 56, 0, 0, 0, 162,
   0, 0, 0, 16, 11, 69, 108, 105, 120, 105, 114, 46, 77, 97, 116, 104, 8, 95,
   95, 105, 110, 102, 111, 95, 95, 10, 97, ...>>, {:multiply, 2}}
iex(3)> Math.multiply(x,2)
20
iex(4)> x
10
iex(5)>
```

Neste código estamos usando o terminal. Temos a função “multiply(a,b)” que multiplica os dois parâmetros. Antes da chamada da função definimos x=10, quando chamamos a função de multiplicar passando x, recebemos o resultado da multiplicação, no entanto, x se mantém o mesmo. As funções não alteram o valor da variável.

Se tentarmos alterar o valor da variável dentro da função como neste próximo exemplo.

```elixir
defmodule Matematica do
  def soma(a, b) do
    a = a + b
  end

  def troca(a, b) do
    aux = a
    a = b
    b = aux
  end
end
```

Em soma atribuímos a variável a o valor da soma de a com b. Em troca tentamos atribuir a variável a outra variável b.

```
warning: variable "a" is unused (there is a variable with the same name in the context,
use the pin operator (^) to match on it or prefix this variable with underscore if it is
 not meant to be used)
  exemplos.ex:24: Matematica.troca/2

warning: variable "b" is unused (there is a variable with the same name in the context,
use the pin operator (^) to match on it or prefix this variable with underscore if it is
 not meant to be used)
  exemplos.ex:25: Matematica.troca/2
```

Recebemos esses erros.

Podemos tentar usar essas funções mesmo tendo esses erros.

```
iex(2)> x=10
10
iex(3)> Matematica.soma(x,2)
12
iex(4)> x
10
iex(5)> Matematica.troca(x,20)
20
iex(6)> x
10
iex(7)>
```

Primeiro criamos a variável x=10, se tentamos passar x para somar com 2 para a função soma conseguimos o resultado de 12 mas o valor de x não é alterado ele mantém sendo 10. Quando mandamos x para trocar com 20 na função troca recebemos o resultado de 20 mas novamente x não é alterado, continua sendo 10.

Um pouco de sintaxe

Agora, mostraremos exemplos de uso do Elixir para situações simples.

- **Switch case**

Elixir tem um jeito elegante de fazer case:

```
iex(3)> case {1,2,3} do
...(3)> {1,2,_} -> "vai entrar para qualquer valor na terceira posicao"
...(3)> {5,5,5} -> "nunca vai entrar"
...(3)> _ -> "vai entrar para qualquer opcao"
...(3)> end
"vai entrar para qualquer valor na terceira posicao"
iex(4)>
```

Assim como em outras linguagens o “_” serve para indicar um valor qualquer, não importa qual seja. No case ele vai entrar na primeira condição verdadeira, neste exemplo no primeiro caso.

- **If-else**

Para fazer algo como um else if temos o cond:

```
iex(4)> cond do
...(4)> 3*3==8 -> "sempre vai ser falso"
...(4)> 1+2==3 -> "sempre vai ser verdadeiro"
...(4)> true -> "sempre vai ser verdadeiro"
...(4)> end
"sempre vai ser verdadeiro"
```

Nesse caso, será executado o trecho de código ligado à primeira condição que retornar verdadeiro, neste exemplo no segundo if.

- **Strings e listas de caracteres**

Algo muito interessante em elixir são as *charlists*, elas são listas de inteiros onde cada elemento representa um número da tabela ASCII. Charlists são usadas com aspas simples ou listas de inteiros, enquanto strings são com aspas duplas. Podemos transformar string em charlist e vice versa.

```
iex(17)> "this is a string"
"this is a string"
iex(18)> 'this is a charlist'
'this is a charlist'
iex(19)> [116,104,105,115,32,105,115,32,97,32,99,104,97,114,108,105,115,116]
'this is a charlist'
iex(20)> to_charlist("hello world")
'hello world'
iex(21)> to_string('hello world')
"hello world"
iex(22)>
```

Um exemplo com uma charlist para todos os elementos que podem ser impressos da tabela ascii:

```
iex(27)> Enum.to_list(32..126)
' !"#$%&\'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuvwxyz{|}~'
iex(28)>
```

- **Estruturas de dados**

Outra coisa interessante é que as listas em elixir são listas encadeadas. Elixir também possui tuplas, onde os elementos ficam armazenados juntos na memória. Então se quiser acessar rapidamente um dado sem o modificar o melhor seria a tupla. Mas se o objetivo é modificar o dado o melhor são as listas encadeadas.

Quando um elemento é adicionado ao final de uma lista encadeada, toda a lista será percorrida, passando de ponteiro em ponteiro, o que é demorado. Por outro lado, quando um elemento é adicionado na última posição de uma tupla, será criada uma cópia dessa tupla com o novo elemento. Como as tuplas ficam em posição sequencial de memória não é possível simplesmente adicionar um elemento. Então para modificar uma tupla é muito mais custoso do que modificar uma lista.

Um exemplo de adicionar um elemento no começo e no final de uma lista.

```
iex(28)> list = [1,2,3,4,5]
[1, 2, 3, 4, 5]
iex(29)> [0] ++ list
[0, 1, 2, 3, 4, 5]
iex(30)> list ++ [6]
[1, 2, 3, 4, 5, 6]
```

Um exemplo de modificação de tupla

```
iex(32)> tuple = {:a,:b,:c,:d,:f}
{:a, :b, :c, :d, :f}
iex(33)> put_elem(tuple,4,:g)
{:a, :b, :c, :d, :g}
iex(34)>
```

- **Multiple clause functions**

Podemos criar funções com múltiplas cláusulas como por exemplo:

```
iex(42)> defmodule IdadeLegal do
...(42)> def idade(n) when n<18,do: "menor de idade"
...(42)> def idade(n) when n>=18,do: "maior de idade"
...(42)> end
{:module, IdadeLegal,
 <<70, 79, 82, 49, 0, 0, 5, 144, 66, 69, 65, 77, 65, 116, 85, 56, 0, 0, 0, 163,
   0, 0, 0, 15, 17, 69, 108, 105, 120, 105, 114, 46, 73, 100, 97, 100, 101, 76,
   101, 103, 97, 108, 8, 95, 95, 105, 110, ...>>, {:idade, 1}}
```

E testando a função:

```
iex(43)> IdadeLegal.idade(17)
"menor de idade"
iex(44)> IdadeLegal.idade(18)
"maior de idade"
iex(45)>
```

Extra - Aplicação distribuída com Elixir

Uma aplicação de Elixir é conseguir fazer programação distribuída facilmente.
Fizemos um exemplo onde tenho em um terminal Ubuntu o iex (Elixir iterativo) com um nome e em outro terminal Ubuntu um iex com outro nome.

```
luanaoliveira@LUANA:~$ iex --sname terminal1
Erlang/OTP 25 [erts-13.0.1] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1] [jit:ns]

Interactive Elixir (1.13.0) - press Ctrl+C to exit (type h() ENTER for help)
iex(terminal1@LUANA)1>
```

```
luanaoliveira@LUANA:~$ iex --sname terminal2
Erlang/OTP 25 [erts-13.0.1] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1] [jit:ns]

Interactive Elixir (1.13.0) - press Ctrl+C to exit (type h() ENTER for help)
iex(terminal2@LUANA)1>
```

Assim temos dois terminais, o primeiro com o nome de “terminal1@LUANA” e o segundo de “terminal2@LUANA”.

Vamos criar duas funções no terminal 1 para depois acessar no outro terminal. A primeira função Hello.world/0 não retorna nada, e a segunda função Math.sum/2 retorna um número.

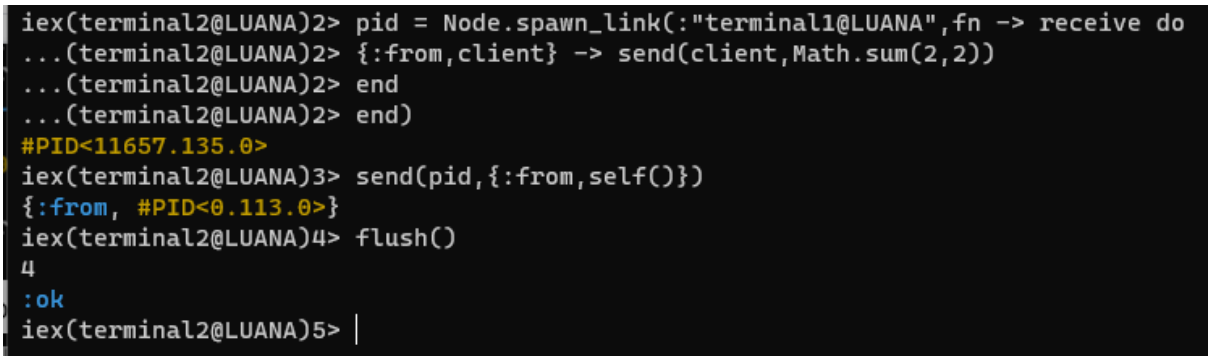
```
iex(terminal1@LUANA)1> defmodule Hello do
...(terminal1@LUANA)1> def world do
...(terminal1@LUANA)1> IO.puts "hello world"
...(terminal1@LUANA)1> end
...(terminal1@LUANA)1> end
{:module, Hello,
 <<70, 79, 82, 49, 0, 0, 5, 72, 66, 69, 65, 77, 65, 116, 85, 56, 0, 0, 0, 173,
   0, 0, 0, 17, 12, 69, 108, 105, 120, 105, 114, 46, 72, 101, 108, 108, 111, 8,
   95, 95, 105, 110, 102, 111, 95, 95, 10, ...>>, {:world, 0}}
iex(terminal1@LUANA)2> defmodule Math do
...(terminal1@LUANA)2> def sum(a,b) do
...(terminal1@LUANA)2> a+b
...(terminal1@LUANA)2> end
...(terminal1@LUANA)2> end
{:module, Math,
 <<70, 79, 82, 49, 0, 0, 5, 80, 66, 69, 65, 77, 65, 116, 85, 56, 0, 0, 0, 157,
   0, 0, 0, 16, 11, 69, 108, 105, 120, 105, 114, 46, 77, 97, 116, 104, 8, 95,
   95, 105, 110, 102, 111, 95, 95, 10, 97, ...>>, {:sum, 2}}
iex(terminal1@LUANA)3>
```

Agora no terminal 2 vamos acessar a função Hello.world/0.

```
iex(terminal2@LUANA)1> Node.spawn_link(:"terminal1@LUANA",fn -> Hello.world() end)
hello world
#PID<11657.134.0>
iex(terminal2@LUANA)2>
```

Com a função spawn_link conseguimos chamar uma função de outro módulo, outra máquina e neste caso outro terminal. Acessando de outro computador faço a mesma coisa, crio um iex com algum nome e uso o nome dele na chamada de por exemplo “Node.spawn_link(:”nome_iex@nome_maquina”,funcao)”.

Mas se a função tem um retorno, temos que adicionar algumas outras funções juntas como a send/2 e a receive.
Para chamar a função Math.sum/2 no terminal 2 temos que:



Dessa forma conseguimos fazer a chamada de uma função de outro módulo e receber o resultado na própria máquina.