



Uma simples introdução a Elixir

Concorrência do jeito certo



@paulosergiolima

Estrutura

- (Yet another) Introdução a linguagens funcionais
- Erlang e Elixir
- Como fazer X no elixir
- As particularidades de elixir
- Exemplos práticos

Linguagens funcionais, o que são onde vivem, o
que comem?

Linguagens funcionais λ

- É um paradigma de programação, baseado na imutabilidade de variáveis e funções.
- O flow de programas é criado se utilizando principalmente de funções sendo compostas por outras funções(duh).
- Elas são cidadãos de primeira classe, assim sendo passadas por outras funções.
- Elas nasceram de lambda calculus, uma sistema de computação, mas não bem uma linguagem de computação
- A maioria das linguagens hoje utilizam alguns aspectos de linguagens funcionais, até um certo ponto.

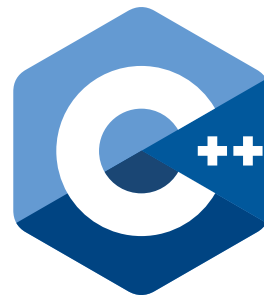
Mais funcionais



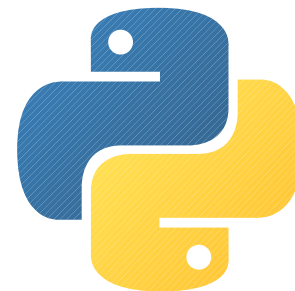
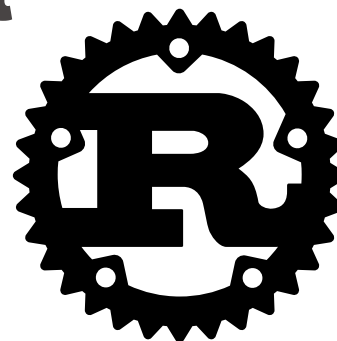
Menos funcionais



elixir



OCaml



But Why?


- POO(Programação orientada a objetos) é bom para modelar sistemas onde á claridade sobre os possíveis objetos e classes(UIs, jogos, etc), porém fora desses campos, há uma dificuldade de modelar compartamentos dentro de objetos, criando sistemas complicados,
- A imutabilidade de variáveis nos garante uma segurança quando compartilhamos elas com variás funções, já que sabemos que seu valor não será modificado.

Elixir e Erlang entram na cena

Erlang

- Criado por Joe Armstrong e seu time na Ericson, uma empresa responsável pela linha telefonica americana, que necessitava de uma linguagem que lidasse com simultaneidade de processos.
- Por isso ela utiliza uma VM, BEAM(como o java), para rodar processos leves que podem ser alternados a qualquer momento
- Muitas linguagens de programação utilizam a BEAM para serem executadas.

Elixir

- Criada por José Valim() , que veio da comunidade de Ruby, e utilizou sua experiência nela para criar uma nova linguagem para BEAM.
- A junção de elementos de Ruby, Clojure e Erlang, criou uma linguagem functional extremamente poderosa e facil de usar.
- Ela passou a popularidade de Erlang, e reviveu o interesse em linguagens funcionais dentro da BEAM, o que levou a novas linguagens serem desenvolvidas.

But why?

- Com o avanço das CPUs, a maioria dos computadores tem mais de uma thread, porém a maioria das linguagens foram criadas quando eles tinham somente uma.
- Por isso, muitas linguagens foram criadas sem pensar em paralelismo, e depois pensaram nisso, dificultando o processo de trabalhar com sistemas distribuídos
- Erlang(e por consequência Elixir) foi criada pensando em sistemas distribuídos, por isso é uma ótima linguagem para sistemas modernos

Quem usa?



PEPSICO



**MINISTÈRE
DE LA TRANSITION
ÉCOLOGIQUE
ET DE LA COHÉSION
DES TERRITOIRES**

*Liberté
Égalité
Fraternité*



But how?

- Utilizando de bibliotecas como Axon e Scholar, conseguimos criar inteligências artificiais.
- Com a Broadway conseguimos criar pipelines de dados, os filtrando modificando e publicando.
- Nerves nos ajuda em aplicações IOT, visando RaspberrysPi
- Porém, Elixir(e a BEAM) brilham mais servidores, com bibliotecas como a Phoenix e Ecto.

Como fazer X no Elixir

Ambiente

- Elixir é interpretada(IEx) e compilada.
- Existe uma variedade de ferramentas desenvolvidas em conjunto com a linguagem, como o package manager mix, e a biblioteca de testes ExUnit
- Extensões para todos os editores de códigos conhecidos(Vim, NeoVim, Emacs, VsCode e etc).
- O Elixir contém uma REPL(read, eval, print, loop), chamada IEx

Tipos no Elixir

- Eles são dinâmicos, mas fortes, ou seja, a linguagem irá decidir o tipo da variável caso não seja especificada, mas não vai fazer conversão implícita deles.
- Existe um trabalho recente de aplicar tipos algébricos na linguagem.

Tipos de dados - Primitivos

- Números representados como Int e Float
- Char e String para caracteres
- Atoms, que contem o mesmo valor que o próprio nome(como o true e false)

Exemplos

```
iex(1)> number = 1
1
iex(2)> i number
Term
  1
Data type
Integer
```

```
iex(1)> number = 1.1
1.1
iex(2)> i 1.1
Term
  1.1
Data type
Float
```

```
iex(1)> text = "cool text"
"cool text"
iex(2)> i text
Term
  "cool text"
Data type
BitString
```

```
iex(1)> best_game = :nier
:nier
iex(2)> i best_game
Term
  :nier
Data type
Atom
```

Tipos de dados - Compostos

- Arrays, que são listas encadeadas, ligando o primeiro elemento com o resto dos elementos, e assim sucessivamente
- Tuples, que são como arrays, porém são preferencialmente usados para referir a elementos com mais de uma propriedade
- Maps, que são estruturas valores chaves, como objetos em outras linguagens.

Fluxo de controle

- **If**
 - Funciona como em qualquer outra linguagem,
- **Case**
 - Como um switch case, compara um valor com vários modelos, correspondendo com todos os que baterem, e performando pattern matching.
- **Cond**
 - Como o case, porém checa se expressões se não se tornam Nil ou false

Pattern Matching

- O operador `=`, comumente conhecido como operador de atribuição, no Elixir e em outras linguagens é conhecido como matching operador, já que ele performa pattern matching.
- Pattern matching é o ato de checar a estrutura de um objeto contendo variáveis(ou não) com outro, e caso essa estrutura seja igual, as variáveis recebem os valores do objeto sendo comparado

Pattern Matching

```
iex(1)> {0, x} = {1, 2}
** (MatchError) no match of right hand side value: {1, 2}
    (stdlib 5.2.1) erl_eval.erl:498: :erl_eval.expr/6
    iex:1: (file)
iex(1)> {1, x} = {1, 2}
{1, 2}
iex(2)> x
2
iex(3)> 
```

Funções

- A parte mais importante de qualquer linguagem funcional
- Elas são definidas pelo seu nome, e sua aridade(a quantidade de argumentos que elas recebem), e não se importa se você define duas funções com o mesmo nome, porém com um número diferente de argumentos

Funções

- A parte mais importante de qualquer linguagem funcional
- Elas são definidas pelo seu nome, e sua aridade(a quantidade de argumentos que elas recebem), e não se importa se você define duas funções com o mesmo nome, porém com um número diferente de argumentos

As particularidades de Elixir

Processos

- No Elixir, todo o código é executado dentro de processos. Os processos são isolados uns dos outros, executados simultaneamente entre si e se comunicam por meio de passagem de mensagens. Os processos não são apenas a base para a competição no Elixir, mas também fornecem os meios para a construção de programas distribuídos e tolerantes a falhas.
- Processos não são processos do sistema, é comum ter milhares de processos rodando ao mesmo tempo

Link de Processos

- Os processos podem ser ligados, forçando quando um processo cai, o outro cai junto, reiniciando juntos e voltando para o estado inicial.
- Processos não são processos do sistema, é comum ter milhares de processos rodando ao mesmo tempo

Estado no Elixir

- Na maioria das aplicações, vai chegar um momento onde vamos querer ter algum estado.
- Nós mantemos estados dentro de processos, modificando e resgatando através de mensagens entre os processos.
- Usamos a Agents, que são uma abstração para esses processos.

```
defmodule KV do
  def start_link do
    Task.start_link(fn -> loop(%{})) end
  end

  defp loop(map) do
    receive do
      {:get, key, caller} ->
        send(caller, Map.get(map, key))
        loop(map)
      {:put, key, value} ->
        loop(Map.put(map, key, value))
    end
  end
end
```

Exercícios Práticos

Servidor de chat

- Github: [o_link_seria_aqui](#)
- Servidor de chat

Programa de ingestão de dados

- Github: `link_seria_aqui`
- Elixir é bem utilizado na ingestão de dados