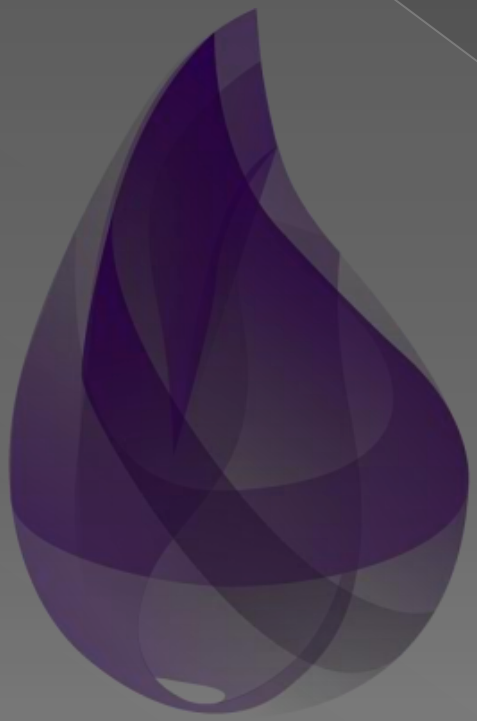

Elixir

Luiz Augusto Penna e Fábio Pimentel

Origens e Influências

- Criador: José Valim
- Objetivos: Permitir maior produtividade e extensibilidade no Erlang VM, mantendo a compatibilidade com ferramentas e ecossistemas de Erlang.
- Data de criação: 2012
- Influenciada por: Erlang

Classificação

- Funcional: Functions e Modules, não existem objetos ou classes.
- Dinâmica: Tipos são checados na execução
- Compilada: Código fonte são transformados em bytecodes que rodam sobre a Erlang VM.
- Usos: A linguagem se faz presente em diversos ramos, dentre os mais populares está framework web phoenix.
- Programação concorrente e paralela: Usando o modelo de Actors.

Concorrência e Paralelismo

- Na programação concorrente e paralela um programa pode ser executado através de diversas linhas de execução.
- Ganho de desempenho em sistemas com muitos processadores (paralelismo físico)
- Ganhos de desempenho através de acesso concorrente a dispositivos de hardware

Conceitos de Pattern Matching

- Pattern matching é uma poderosa parte de Elixir, nos permite procurar padrões simples em valores, estruturas de dados, e até funções.

- Em Elixir, o operador = é na verdade o nosso operador match. Quando usado, a expressão inteira se torna uma equação e faz com que Elixir o combine os valores do lado esquerdo com os valores do lado direito da expressão. Se a comparação for bem sucedida, o valor da equação é retornado. Se não, um erro é lançado.

Exemplo de Pattern Matching

- A linguagem vai sempre tentar "casar" o valor da esquerda com o valor da direita.

```
iex(1)> num = 42
42
iex(2)> 42 = num
42
iex(3)> 52 = num
** (MatchError) no match of right hand side value: 42
```

Exemplo de Pattern Matching

- Dependendo do parâmetro que a função recebe, ela pode tomar diferentes caminhos.

```
iex(1)> defmodule Fib do
...(1)> def fib(0) do 0 end
...(1)> def fib(1) do 1 end
...(1)> def fib(n) do fib(n-1) + fib(n-2) end
...(1)> end
{:module, Fib,
 <<70, 79, 82, 49, 0, 0, 3, 240, 66, 69, 65, 77, 65, 116, 85, 56, 0, 0, 0, 84,
   0, 0, 0, 10, 10, 69, 108, 105, 120, 105, 114, 46, 70, 105, 98, 8, 95, 95,
   105, 110, 102, 111, 95, 95, 9, 102, 117, ...>>, {:fib, 1}}
iex(2)> IO.puts Fib.fib(0)
0
:ok
iex(3)> IO.puts Fib.fib(1)
1
:ok
iex(4)> IO.puts Fib.fib(7)
13
:ok
iex(5)> IO.puts Fib.fib(2)
1
:ok
```

Conceitos de Metaprogramação

- A Metaprogramação é um recurso muito comum de linguagens dinâmicas permite a utilização de códigos para escrever código. Em outras palavras podemos dizer que é a capacidade de gerar/alterar código em tempo de execução.
- Para entender esse conceito é necessário a compreensão de como as expressões são representadas. Em Elixir, a árvore de sintaxe abstrata(AST), que é a representação interna do nosso código, é composta de tuplas. Essas tuplas contêm três partes:

1. Nome da função
2. Metadados
3. Argumentos da função

Exemplo de Metaprogramação

- Modelo tupla para AST: `{function_call, meta_data_for_context, argument_list}`

```
iex(1)> quote do: 1+2
{:+, [context: Elixir, import:
Kernel], [1, 2]}
```

Conclusão

- Elixir é incrível e fácil de entender, tem tipagem simples, porém poderosa e ferramentas muito úteis em torno da linguagem. A programação funcional faz com que o código seja mais fácil de refazer, o modelo pode receber mudanças e manutenção com facilidade.

Bibliografia

- https://elixirschool.com/pt/
- https://elixir-lang.org/
- https://pt.wikipedia.org/wiki/Elixir_%28linguagem_de_programa%C3%A7%C3%A3o%29
- https://speakerdeck.com/volcov/elixir-quem-e-este-pokemon