Elixir

Gustavo Covas (7995052), Victor O. F. Faria (8515919)

1 Introdução

Elixir[3] é uma linguagem Open-Source de propósito geral permeada de aspectos funcionais. Seu design visa a construção de sistemas massivamente concorrentes e com tolerância a falhas. Para isso, Elixir executa na máquina virtual de Erlang[4], ambiente provido de ferramentas para lidar com aqueles aspectos, conforme veremos neste trabalho. Mais do que isso, o design de Elixir apresenta preocupações com a produtividade, expressividade e estética da linguagem, herdando ideias da linguagem Ruby[20] e de seu framework Ruby on Rails[19].

2 Histórico e influências

A linguagem teve origem na companhia Plataformatec, uma empresa brasileira conhecida por suas contribuições à comunidade Rails.[1] Segundo o seu criador José Valim, parte da inspiração para a criação de Elixir foi a dificuldade em melhorar a performance de aplicações Ruby on Rails em sistemas multi-core, tendo em vista que a linguagem Ruby não provê um ferramental satisfatório para lidar com problemas de concorrência [12].

Nesse sentido, Erlang aparece como uma tecnologia adequada às demandas do desenvolvimento de aplicações web[6]. Entretanto, o código desenvolvido em Erlang apresenta problemas como o excesso de repetição e de código boilerplate[7], além da falta de alguns operadores funcionais[12]. Assim, Valim criou Elixir como uma tentativa de trazer um ferramental poderoso e extensível para o ecossistema Erlang.

Ainda devido às origens de Elixir, é notável na linguagem a influência de Ruby, principalmente na sintaxe. Contudo, o fato de Elixir ser executada na VM de Erlang faz com que aquela seja muito mais próxima desta do que a mencionada Ruby[7]. Detalharemos a influência dessas duas tecnologias sobre Erlang a seguir.

2.1 Ruby

Criada em 1995 por Yukihiro Matsumoto, Ruby é uma linguagem dinâmica que combina elementos de programação funcional, imperativa e orientada a objetos. A linguagem possui ainda um objetivo ambicioso de parecer "natural" aos programadores, procurando evitar a presença de sinais de pontuação em sua sintaxe e incentivar o uso de keywords na língua inglesa[1].

Considere o exemplo da sintaxe do Ruby a seguir, retirado da página da linguagem[20]:

```ruby
# The Greeter class
class Greeter
  def initialize(name)
    @name = name.capitalize
  end

  def salute
    puts "Hello #{@name}!"
  end
end

# Create a new object
g = Greeter.new("world")

# Output "Hello World!"
g.salute
```

[1] A criação mais conhecida da Plataformatec é a gem devise[8], uma solução para autenticação de usuários. Seu repositório possui mais de 17000 estrelas no GitHub. Em comparação, o repositório do framework Rails possui 36000 estrelas [9]. O repositório do kernel do Linux conta com mais de 46000[10].

Observe o cuidado da linguagem ao preferir as keywords `class`, `def` e `end` para delimitar os blocos referentes às classes e métodos, ao contrário das usuais chaves (`{}`) presentes em outras linguagens. Note também a ausência de parênteses nas chamadas de métodos de `puts` e `salute`. São exemplos de cuidados que inspiraram a sintaxe de Elixir, que possui recursos similares.

2.2 Erlang

A linguagem Erlang foi criada nos anos 80 como um projeto de pequisa da Ericsson, uma empresa sueca da área de telecomunicações. Projetada para conter primitivas para concorrência e recuperação de erros desde o começo[5], a linguagem é classificada como funcional, e usa ostensivamente o modelo de Atores[15], implementando cada ator como um processo de sua máquina virtual. Como Elixir executa sobre a máquina virtual de Erlang, discutiremos esse modelo (demonstrando como utilizá-lo no ambiente Elixir) mais adiante. Para uma demonstração da sintaxe de Erlang, considere o trecho de código a seguir, disponível como exemplo na Wikipedia[22]:

```
%% This is the file 'fibonacci.erl', the module and the filename must match
-module(fibonacci).
-export([fib/1]). % This exports the function 'fib' of arity 1

fib(0) -> 0; % If 0, then return 0, otherwise (note the semicolon ; meaning 'else')
fib(1) -> 1; % If 1, then return 1, otherwise
fib(N) when N > 1 -> fib(N - 1) + fib(N - 2).
```

Notamos que a sintaxe da linguagem contém algumas sutilezas como o uso de ponto-e-vírgula com função semelhante à keyword `else` em outras linguagens. Entretanto, esses detalhes de sintaxe parecem não incomodar tanto programadores familiarizados com a linguagem Prolog[7] ou então outras linguagens funcionais.

Outro aspecto interessante de Erlang são as marcas alcançadas devido à alta concorrência e tolerância a falhas no ambiente de desenvolvilmento propoporcionado pela linguagem. Como exemplos marcantes, temos um switch da Ericsson com um índice de 99.9999999% de disponbilidade (9 noves) [23] e ainda uma demonstração do Whatsapp de um servidor com mais de 2 milhões de conexões TCP abertas simultaneamente [21].

3 Sintaxe

A sintaxe de Elixir lembra bastante a sintaxe de Ruby, conforme veremos a seguir. Destaca-se a baixa presença de sinais de pontuação e a preferência por keywords na língua inglesa como `def`, `do` e `end`. Para uma introdução abrangente sobre a sintaxe de Elixir, consulte o guia da linguagem [3]. Comentaremos sobre alguns recursos de Elixir relacionados à programação funcional.

3.1 Exemplos de recursos funcionais

3.1.1 Pipes

Elixir apresenta um recurso sintático para o encadeamento de chamadas de função semelhante aos pipes nos shells do sistema operacional Unix. Considere o exemplo de chamadas encadeadas de funções a seguir:

```
Enum.sum(Enum.filter(Enum.map(1..100_000, fn x -> 3*x end, odd?)))
```

O trecho de código acima utiliza operações comuns em linguagens funcionais (map, filter, reduce) para iterar pela sequência dos números inteiros de 1 a 100000, somando o triplo de todos os ímpares nesta sequência. A seguir, o mesmo trecho reproduzido com o uso do operador pipe de Elixir, que proporciona uma clareza maior em relação à ordem das operações:

```
1..100_000 |> Enum.map(&(&1 * 3)) |> Enum.filter(odd?) |> Enum.sum
```

3.1.2 Pattern matching

O sinal `=` é na verdade um operador de Pattern Matching em Elixir. Assim, expressões do tipo `1 = x` são válidas quando a variável `x` tem valor 1. Caso a variável x tenha um valor diferente de 1, a expressão ocasiona um erro do tipo `MatchError`. O exemplo a seguir, retirado do guia[17],

ilustra o comportamento do operador `=`. Note que só é possível atribuir valores a variáveis do lado esquerdo da expressão.

```
iex> x = 1
1
iex> x
1
iex> 1 = x
1
iex> 2 = x
** (MatchError) no match of right hand side value: 1
```

Também é possível usar o operador `=` sobre estruturas mais complexas:

```
iex> {:ok, result} = {:ok, 13}
{:ok, 13}
iex> result
13

iex> {:ok, result} = {:error, :oops}
** (MatchError) no match of right hand side value: {:error, :oops}
```

4 Conceitos herdados de Erlang

4.1 Processos

Assim como em Erlang, em Elixir temos processos como unidade básica de execução, isto é, todo código executa em um processo. Ressaltamos que não deve se confundir um processo Elixir com os processos do sistema operacional. Os primeiros, em geral, são leves, de modo que não é incomum haver dezenas ou centenas de milhares de Processos Elixir executando concorrentemente em uma mesma máquina[18]. Ressaltamos ainda que todos os processos são isolados (não compartilham memória) e se comunicam através da troca de mensagens, utilizando para isso a estrutura provida pela Erlang VM.

Devido à leveza do Processos em Erlang/Elixir, há um princípio conhecido que aplica-se ao projetar sistemas com essas tecnologias: o princípio *Let it Crash*. [15]. A ideia é evitar a programação preventiva - a checagem de casos de erro, captura de exceções etc - e deixar que os processos morram. Utiliza-se então as ferramentas providas pela biblioteca padrão de Erlang para gerenciar a supervisão e reinicialização dos mesmos, para garantir que o sistema como um todo (que é um conjunto de milhares de processos) permaneça saudável. Com isso, resolve-se erros causados por acumulação de estado, apenas reiniciando os processos envolvidos. Procura-se evitar erros facilmente reprodutíveis (isto é, que não dependam de estados) por meio de testes de unidade.

4.1.1 Criação

A criação de um processo é feita utilizando-se a função `spawn`. No exemplo a seguir, demonstramos a criação de um processo que imprime a string "Hello, world!" e termina sua execução em seguida. O estado do processo é checado utilizando-se a função `Process.alive?`

```
iex(1)> pid = spawn(fn -> IO.puts "Hello, world!" end)
Hello, world!
#PID<0.87.0>
iex(2)> Process.alive? pid
false
```

4.1.2 Troca de mensagens

A troca de mensagens entre processos é feita principalmente utilizando-se as funções `send` e `receive`[2]. Utiliza-se os PIDs dos processos para endereçamento de mensagens, que são retornados por `spawn` ou obtidos chamando `self()` no shell de Elixir interativo (iex), como demonstrado no exemplo a seguir. `send` recebe o PID do destinatário e a mensagem a ser enviada, e é uma função assíncrona. `receive`, por sua vez, recebe um bloco de código a ser executado na chegada de uma

[2]Note que em Elixir existem funcionalidades muito mais sofisticadas para o endereçamento e troca de mensagens, que estão fora do escopo deste trabalho. Demonstraremos apenas o básico para a troca de mensagens simples.

mensagem, e é uma função bloqueante, ou seja, a execução é interrompida até a chegada de uma mensagem.

```
iex(1)> parent = self()
#PID<0.84.0>
iex(2)> spawn fn -> send(parent, {:hello, self()}) end
#PID<0.87.0>
iex(3)> receive do
...(3)> {:hello, pid} -> "Got hello from #{inspect pid}"
...(3)> end
"Got hello from #PID<0.87.0>"
```

Este exemplo é inspirado na seção do Guia de Elixir sobre Processos e troca de mensagens. Para mais detalhes e funcionalidades possíveis na troca de mensagens, consulte o mencionado guia [18].

4.2 Usando processos para armazenar estado

Uma aplicação interessante é utilizar um terceiro processo como armazenamento de estado compartilhado entre processos. O exemplo a seguir, retirado do guia [18], demonstra como criar um processo para armazenar uma estrutura chave-valor (daí o nome do módulo criado, `KV` de Key-Value). Observamos ainda que o código a seguir é meramente uma demonstração e não uma solução robusta para armazenamento de estado em 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
```

`loop` é uma função privada do módulo `KV` (por isso é definida com a keyword `defp` e não `def`). `start_link` utiliza o módulo `Task` provido por Elixir, que gerencia a sincronização entre processos. A interface do processo `KV` consiste em dois tipos de mensagens: `{:get, key, caller}` para o processo `caller` obter o valor do valor associado a `key` e `{:put, key, value}` para armazenar-se o valor `value` associado à chave `key`. O uso dos dois tipos de mensagem é exemplificado a seguir:

```
iex(3)> {:ok, pid} = KV.start_link
{:ok, #PID<0.94.0>}
iex(7)> send pid, {:put, :hello, "World!"}
{:put, :hello, "Gold!"}
iex(8)> send pid, {:get, :hello, self()}
{:get, :hello, #PID<0.84.0>}
iex(9)> flush()
"World!"
:ok
```

Onde `flush()` é a função chamada para retirar o conteúdo da mailbox do processo atual, no caso, do próprio processo iex.

4.3 Elixir e OTP

OTP (Open Telecom Platform) é um conjunto de bibliotecas disponibilizado nas instalações de Erlang. É imprescindível utilizar os recursos do OTP para construir aplicações Elixir robustas, dentre os quais estão inclusos:

- Agents: Abstração de estados
- Tasks: Ferramentas para lidar com sincronização de processos

- GenServer: Interface para implementação do servidor em arquiteturas cliente/servidor
- Supervisor: Gerenciamento de uma árvore de processos

Para um tutorial detalhado sobre como utilizar os módulos que o OTP disponibiliza em uma aplicação Elixir, consulte o guia oficial de Elixir[14].

5 Casos de uso

5.1 Framework Phoenix

O Phoenix é um framework web MVC projetado para lidar com altíssimo throughput e ser executado em ambientes distribuídos.

Além dos recursos típicos de um framework MVC como o Rails, o Phoenix conta com:

- Ferramentas de testes automatizados para código concorrente
- Distributed live reloading. Isto é diversos. componentes que formam uma aplicação web moderna podem ter seus endpoints em um sistema distribuído arbitrário e todas as atualizações que impliquem em uma mudança no frontend ou que impacte outros componentes é automaticamente propagada e conta com o suporte de tolerância a falhas nativo da Erlang VM.
- Um gerenciador de pacotes para o framework.
- Ferramentas nativas para suporte a recursos de ECMA Script 6.

O Phoenix tem sido amplamente usado pela comunidade Elixir, que embora pequena tem crescido significativamente impulsionada pela comunidade Ruby. O que por sua vez é bastante plausível dado que Ruby on Rails possui sérias complicações de desempenho e limitações para operar em ambientes concorrentes em larga escala. Assim sendo, Elixir e Phoenix vem a encontro de suprir essa demanda oferecendo recursos familiares à comunidade de Ruby.

Um caso de uso interessante do Phoenix é o Heroic Labs [11], cujo uso é uma aplicação de infraestrutura escalável. Podemos ver então que é possível desenvolver tais aplicações que Go talvez fosse a primeira opção (isto por esta ter sido criada com o ambiente de servidores e infraestrutura em vista); no entanto, ainda há o aspecto de concisão de Elixir a ser considerado monstrando assim que há espaço para considerar Phoenix uma opção nessa modalidade de aplicação.

5.2 Pinterest

Outro caso de uso do Elixir na indústria é relatado pela equipe de engenharia da empresa Pinterest[13]. Usando a linguagem, o sistema que gerencia os limites de chamadas de API para o sistema de anúncios da plataforma possui um tempo de resposta menor que 800 microssegundos para 90% das requisições.

A implementação do sistema de notificações, convertida de um sistema de atores em Java para Elixir, diminuiu de 10000 para 1000 linhas de código. A nova implementação é, segundo a equipe, mais rápida e concisa, além de rodar na metade do número de servidores.

5.3 Rose Point

Elixir também é usado em software embutido. É o caso da empresa americana Rose Point Navigation Systems, que utiliza a linguagem para o desenvolvimento de software usado na indústria náutica para conversão entre protocolos de rede[2]. Usando o projeto Nerves [16], a empresa fabrica dispositivos que rodam a Erlang VM, entregando um ambiente estável em Elixir.

Referências

[1] About ruby. https://www.ruby-lang.org/en/about/. Acesso em: 2017-06-27.

[2] Elixir in production interview: Garth hitchens. http://blog.plataformatec.com.br/2015/06/elixir-in-production-interview-garth-hitches/. Acesso em: 2017-06-28.

[3] Elixir language. https://elixir-lang.org/. Acesso em: 2017-06-25.

[4] Erlang programming language. https://www.erlang.org/. Acesso em: 2017-06-26.

[5] Erlang programming language - history of erlang. https://www.erlang.org/course/history. Acesso em: 2017-06-27.

[6] The erlangelist: Erlang based server systems. http://theerlangelist.blogspot.com.br/2013/01/erlang-based-server-systems.html. Acesso em: 2017-06-26.

[7] The erlangelist: Why elixir. http://theerlangelist.com/article/why_elixir. Acesso em: 2017-06-26.

[8] Github - plataformatec/devise. https://github.com/plataformatec/devise. Acesso em: 2017-06-26.

[9] Github - rails/rails. https://github.com/rails/rails. Acesso em: 2017-06-26.

[10] Github - torvalds/linux. https://github.com/torvalds/linux. Acesso em: 2017-06-26.

[11] Heroic labs. https://heroiclabs.com/. Acesso em: 2017-06-28.

[12] An interview with elixir creator josé valim. https://www.sitepoint.com/an-interview-with-elixir-creator-jose-valim/. Acesso em: 2017-06-26.

[13] Introducing new open-source tools for the elixir community. https://medium.com/@Pinterest_Engineering/introducing-new-open-source-tools-for-the-elixir-community-2f7bb0bb7d8c. Acesso em: 2017-06-28.

[14] Introduction to the mix. https://elixir-lang.org/getting-started/mix-otp/introduction-to-mix.html. Acesso em: 2017-06-28.

[15] Learn you some erlang for great good! - what is erlang? https://www.erlang.org/course/history. Acesso em: 2017-06-27.

[16] Nerves project. http://nerves-project.org/. Acesso em: 2017-06-28.

[17] Pattern matching - elixir language. https://elixir-lang.org/getting-started/pattern-matching.html. Acesso em: 2017-06-28.

[18] Processes - elixir. https://elixir-lang.org/getting-started/processes.html. Acesso em: 2017-06-28.

[19] Ruby on rails. http://rubyonrails.org/. Acesso em: 2017-06-26.

[20] Ruby programming language. https://www.ruby-lang.org/en/. Acesso em: 2017-06-26.

[21] Whatsapp blog - 1 million is so 2011. https://blog.whatsapp.com/196/1-million-is-so-2011. Acesso em: 2017-06-28.

[22] Wikipedia - erlang (programming language). https://en.wikipedia.org/wiki/Erlang_(programming_language). Acesso em: 2017-06-27.

[23] Armstrong, J. Concurrency oriented programming in erlang. http://www.rabbitmq.com/resources/armstrong.pdf. Acesso em: 2017-06-28.