Metaprogramação em Elixir

O conceito e suas aplicações práticas

Paulo Valente

Desenvolvedor Backend @ Stone Pagamentos

ElugCE - Dezembro/2020

O que é metaprogramação

Definição de dicionário

- *meta-* é o prefixo grego que significa "para além de" ou transcedental
- No nosso contexto, segue o conceito de "sobre si mesmo"
- Programas que geram ou modificam a si mesmos
  - *Código que gera código*

Primeiros Passos

Queremos implementar a função XOR(A, B) onde:

A	B	XOR(A, B)
zero	zero	0
zero	one	1
one	zero	1
one	one	0

Primeiros Passos

```
defmodule VerbalXOR.Simple do
  def xor("zero", "zero"), do: 0
  def xor("zero", "one"), do: 1
  def xor("one", "zero"), do: 1
  def xor("one", "one"), do: 0
end
```

Primeiros Passos

```elixir
defmodule VerbalXOR.Simple do
  @mapping %{
      "zero" => 0,
      "one" => 1
  }

  defguardp is_valid(x) when x in ["zero", "one"]

  def xor(a, b) when is_valid(a) and is_valid(b) do
    Bitwise.bxor(@mapping[a], @mapping[b])
  end
end
```

Primeiros Passos

```elixir
defmodule VerbalXOR.Metaprogrammed do
  @mapping %{
      "zero" => 0,
      "one" => 1
  }

  for a <- @mapping, b <- @mapping do
    a_num = to_num(a)
    b_num = to_num(b)

    result = Bitwise.bxor(a_num, b_num)

    def xor(unquote(a), unquote(b)), do: unquote(result)
  end

  defp to_num("zero"), do: 0
  defp to_num("one"), do: 1
end
```

Destrinchando o código

O que é `unquote()`?

- Em strings, inserimos valores com #{}:
  - "Hello, #{name}!"
- Em inglês, aspas se chamam *quotes*
- *unquote* seria "desfazer as aspas"
- Por analogia, se estou inserindo código na Abstrata (*Abstract Syntax Tree*, Árvore Abstrata de Sintaxe em tradução livre), é preciso fazer `unquote()`

Destrinchando o código

Quando precisamos de unquote

```elixir
defmodule Exemplo do
  x = 1
  y = 2
  def f(x = unquote(x), y), do: x + y
  def g(x, y), do: x + unquote(y)
end
```

- Injetar valor no `function head` pede `unquote`
- Injetar valor definido fora da função no corpo dela pede `unquote`
- Injetar valor em um `quote` pede `unquote`

Destrinchando o código

E cadê os `quotes` então?

```elixir
defmodule MyApp.DataCase do
  # Toda macro retorna um `quote`, o que significa que
  # ela retorna a máquina de estados do código, em vez
  # de retornar o resultado dele

  # Além disso, toda macro também recebe a máquina de estados dos argumentos
  defmacro __using__(opts) do
    quote do
      # Não funciona se opts não estiver definido no
      # módulo que chamar `use MyApp.DataCase`
      unless opts[:async] do
        Ecto.Adapters.SQL.Sandbox.mode(MyApp.Repo, {:shared, self()})
      end
    end
  end
end
```

E cadê os `quotes` então?

```elixir
defmodule MyApp.DataCase do
  defmacro __using__(opts) do
    quote do
      # Agora, injetamos o VALOR do opts recebidos pela macro
      # no código que por sua vez será injetado no
      # local onde chamamos `use MyApp.DataCase`
      unless unquote(opts)[:async] do
        Ecto.Adapters.SQL.Sandbox.mode(MyApp.Repo, {:shared, self()})
      end
    end
  end
end
```

E cadê os `quotes` então?

```
defmodule PlusAsMinus do
  defmacro run({:+, env, args}) do
    {:-, env, args}
  end

  defmacro run({:-, env, args}) do
    {:+, env, args}
  end
end

iex(1)> require PlusAsMinus
PlusAsMinus
iex(2)> PlusAsMinus.run(1 + 2)
-1
iex(3)> PlusAsMinus.run(1 - 2)
3
```

Geração de código a partir de um arquivo

```elixir
defmodule ABCDto1234 do
  @filename :my_app
            |> :code.priv_dir()
            |> Path.join("my_source.json")

  @external_resource @filename

  @mapping @filename
           |> File.read!()
           |> Jason.decode!()

  @doc "converts a, b, c, d to 1, 2, 3, 4"
  @spec convert(letter :: String.t()) :: {:ok, pos_integer()} | {:error, :invalid_letter}
  # é boa prática colocar a spec, doc e function head fora do loop de iteração
  def convert(letter)

  Enum.each(@mapping, fn {k, v} ->
    def convert(unquote(k)), do: {:ok, unquote(v)}
  end)

  def convert(_), do: {:error, :invalid_letter}
end
```

Geração de código a partir de um arquivo

Conteúdo do arquivo "my_app/priv/my_source.json"

```
{
  "a": 1,
  "b": 2,
  "c": 3,
  "d": 4
}
```

Geração de testes mais completos

```elixir
defmodule ABCDto1234Test do
  use ExUnit.Case, async: true

  # Esse módulo contem 5 test cases
  describe "convert/1" do
    for {k, v} <- %{"a" => 1, "b" => 2, "c" => 3, "d" => 4} do
      test "should convert #{k} to #{v}" do
        # atenção: como injeta um valor, eu posso usar unquote
        # no lado esquerdo de uma operação de match
        assert {:ok, unquote(v)} = ABCDto1234.convert(unquote(k))
      end
    end

    test "should convert other values to error" do
      assert {:error, :invalid_letter} == ABCDto1234.convert("bla")
    end
  end
end
```

Conclusão

- Podemos usar metaprogramação para gerar código mais conciso
- É possível gerar código a partir de dados
- Metaprogramação deixa o programa menos legível
- Metaprogramação torna a depuração mais difícil
- Testes andam muito bem com metaprogramação,
  pois permite testarmos mais casos com menos código

Obrigado!

Perguntas?