
elixir

Elixir

- Динамически типизированный функциональный язык программирования.

- Спроектирован специально для построения легких в обслуживании и масштабируемых систем.

- Компилятор генерирует байткод для Erlang VM, виртуальной машины для языка Erlang.

- Позволяет использовать существующие Erlang библиотеки.

Erlang VM

- Является виртуальной машиной для исполнения байткода ЯП Erlang.

- Erlang VM работает как один процесс ОС. По умолчанию выделяется по одному потоку на каждое ядро процессора для достижения максимальной производительности. Количество используемых потоков и ядер процессора можно задать при запуске виртуальной машины.

- Процессы Erlang реализованы полностью на уровне Erlang VM и не имеют отношения к процессам или потокам ОС. Благодаря этому можно запустить миллион потоков Erlang используя один процесс ОС и по одному потоку на каждое ядро.

Философия

По свидетельству Майка Вильямса, одного из разработчиков Erlang VM, язык задумывался для решения трёх проблем разработки распределённых систем мягкого реального времени с высокой степенью параллелизма:

- Возможности быстрой и эффективной разработки ПО.
- Получения системы, устойчивой к программным и аппаратным сбоям.
- Возможности обновления системы «на лету», без простоя оборудования.

Let it crash

Высокая отказоустойчивость кроется в применении изолированных друг от друга легковесных процессов, связанных лишь механизмом обмена сообщениями и сигналами выхода. Принцип разработчиков на Erlang по отношению к обработке ошибочных ситуаций в процессах можно выразить в виде высказывания: «Позвольте приложению упасть, и пускай кто-то другой займётся им».

Elixir == Erlang?

- Elixir - это Erlang с улучшенным синтаксисом.
- Производительность не страдает, так как компилятор Elixir генерирует такой же байткод Erlang VM как и компилятор Erlang.

Interactive Elixir

- `iex` - Интерактивная оболочка, REPL для Elixir

```
iex> 40 + 2
42
iex> "hello" <> " world"
"hello world»
iex> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
```

Базовые Типы

```
iex> 1           # integer
iex> 0x1F        # integer
iex> 1.0         # float
iex> true        # boolean
iex> :atom       # atom / symbol
iex> "elixir"    # string
iex> [1, 2, 3]   # list
iex> {1, 2, 3}   # tuple
```

```
# Атомы это константы
:hello # atom

# Кортежи хранятся непрерывно в памяти.
{1,2,3} # tuple

# Можно получить элемент кортежа с помощью функции elem:
elem({1, 2, 3}, 0) #=> 1

# Списки реализованы как односвязные списки.
[1,2,3] # list

# Можно получить доступ к голове списка и хвосту:
[head | tail] = [1,2,3]
head #=> 1
tail #=> [2,3]
```

Арифметические операции

```
iex> 1 + 2
3
iex> 5 * 5
25
iex> 10 / 2
5.0
iex> div(10, 2)
5
iex> div 10, 2
5
iex> rem 10, 3
1
```

Базовые операции

```
iex> true and true
true
iex> false or is_atom(:example)
true
iex> true && 17
17
iex> 1 || true
1
iex> !true
false
```

Pattern Matching

```
iex> [h | _] = [1, 2, 3]
[1, 2, 3]
iex> h
1
iex> x = 1
1
iex> x
1
iex> 1 = x
1
iex> 2 = x
** (MatchError) no match of right hand side value: 1
```

Case

```
iex> case {1, 2, 3} do
...>   {4, 5, 6} ->
...>     "This clause won't match"
...>   {1, x, 3} ->
...>     "This will match and bind x to 2 in this clause"
...>   _ ->
...>     "This clause would match any value"
...> end
"This will match and bind x to 2 in this clause"
```

Cond

```
iex> cond do
...>   2 + 2 == 5 ->
...>     "This will not be true"
...>   2 * 2 == 3 ->
...>     "Nor this"
...>   1 + 1 == 2 ->
...>     "But this will"
...> end
"But this will"
```

Guards

```
iex> case {1, 2, 3} do
...>   {1, x, 3} when x > 0 ->
...>     "Will match"
...>   _ ->
...>     "Would match, if guard condition were not
satisfied"
...> end
"Will match"

iex> case :ok do
...>   :error -> "Won't match"
...> end
** (CaseClauseError) no case clause matching: :ok
```

If & unless

```
iex> if nil do
...>   "This won't be seen"
...> else
...>   "This will"
...> end
"This will"

iex> unless true do
...>   "This will never be seen"
...> end
```

do/end блоки

```
iex> if true do
...>   a = 1 + 2
...>   a + 10
...> end
13

iex> is_number(if true do
...>  1 + 2
...> end)
true
```

Keyword List & Map

```
iex> list = [{:a, 1}, {:b, 2}]
[a: 1, b: 2]
iex> list == [a: 1, b: 2]
true
iex> list[:a]
1

iex> list ++ [c: 3]
[a: 1, b: 2, c: 3]
iex> [a: 0] ++ list
[a: 0, a: 1, b: 2]
```

```
iex> map = %{:a => 1, 2 => :b}
%{2 => :b, :a => 1}
iex> map[:a]
1
iex> map[2]
:b
iex> map[:c]
nil

iex> %{} = %{:a => 1, 2 => :b}
%{:a => 1, 2 => :b}
iex> %{:a => a} = %{:a => 1, 2 => :b}
%{:a => 1, 2 => :b}
iex> a
1
iex> %{:c => c} = %{:a => 1, 2 => :b}
** (MatchError) no match of right hand side value: %{2 =>
:b, :a => 1}
```

Функции в Elixir

Функция однозначно характеризуется двумя параметрами: **name/arity**, где **name** это имя функции, а **arity** количество передаваемых ей аргументов

```
iex> String.upcase("hellö")
"HELLÖ"
```

Здесь мы использовали функцию upcase/1 из стандартного модуля String.

Модули

```
defmodule Math do
  def sum(a, b) do
    do_sum(a, b)
  end

  defp do_sum(a, b) do
    a + b
  end
end

IO.puts Math.sum(1, 2)    #=> 3
IO.puts Math.do_sum(1, 2) #=> ** (UndefinedFunctionError)
```

Анонимные функции

Определяются с помощью ключевых слов **fn** и **end**.

```
iex> add = fn a, b -> a + b end
#Function<12.71889879/2 in :erl_eval.expr/5>
iex> is_function(add)
true
iex> is_function(add, 2)
true
iex> is_function(add, 1)
false
iex> add.(1, 2)
3
```

Рекурсия

```
defmodule Recursion do
  def print_multiple_times(msg, n) when n <= 1 do
    IO.puts msg
  end

  def print_multiple_times(msg, n) do
    IO.puts msg
    print_multiple_times(msg, n - 1)
  end
end

Recursion.print_multiple_times("Hello!", 3)
# Hello!
# Hello!
# Hello!
```

Куда пойти дальше?

- http://elixir-lang.org/ - официальный сайт Elixir.
- http://www.erlang.org/ - официальный сайт Erlang.
- http://learnyousomeerlang.com/ - отличная бесплатная книга по Erlang для начинающих.

Спасибо!