Бесплатная электронная книга

УЧУСЬ

Elixir Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#elixir

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: elixir-language

It is an unofficial and free Elixir Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Elixir Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с Elixir Language

замечания

Elixir - динамический функциональный язык, предназначенный для создания масштабируемых и поддерживаемых приложений.

Эликсир использует Erlang VM, известную тем, что работает с низкими задержками, распределенными и отказоустойчивыми системами, а также успешно используется в веб-разработке и встраиваемом программном обеспечении.

Версии

Версия	Дата выхода
0.9	2013-05-23
1,0	2014-09-18
1,1	2015-09-28
1.2	2016-01-03
1,3	2016-06-21
1.4	2017-01-05

Examples

Привет, мир

Инструкции по установке на elixir здесь описаны в инструкциях, относящихся к различным платформам.

Эликсир - это язык программирования, созданный с использованием `erlang` , и использует время выполнения `BEAM` erlang (например, `JVM` для java).

Мы можем использовать эликсир в двух режимах: интерактивные оболочки `iex` или непосредственно работают с помощью `elixir` команды.

Поместите в файл с именем `hello.exs` :

```
IO.puts "Hello world!"
```

В командной строке введите следующую команду для выполнения исходного файла Elixir:

```
$ elixir hello.exs
```

Это должно выводить:

    Привет, мир!

Это известно как *сценарий режима* `Elixir` . Фактически, программы Elixir также могут быть скомпилированы (и, как правило, они) в байт-код для виртуальной машины BEAM.

Вы также можете использовать `iex` для интерактивной оболочки `iex` (рекомендуется), запустите команду, вы получите приглашение следующим образом:

```
Interactive Elixir (1.3.4) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
```

Здесь вы можете попробовать свои примеры эликсира `hello world` :

```
iex(1)> IO.puts "hello, world"
hello, world
:ok
iex(2)>
```

Вы также можете скомпилировать и запустить свои модули через `iex` . Например, если у вас есть `helloworld.ex` который содержит:

```
defmodule Hello do
   def sample do
       IO.puts "Hello World!"
   end
end
```

Через `iex` выполните:

```
iex(1)> c("helloworld.ex")
[Hello]
iex(2)> Hello.sample
Hello World!
```

Hello World от IEx

Вы также можете использовать `IEx` (Interactive Elixir) для оценки выражений и выполнения кода.

Если вы находитесь на Linux или Mac, просто введите iex в свой bash и нажмите enter:

```
$ iex
```

Если вы находитесь на машине под Windows, введите:

```
C:\ iex.bat
```

Затем вы войдете в IEx REPL (Read, Evaluate, Print, Loop), и вы можете просто ввести что-то вроде:

```
iex(1)> "Hello World"
"Hello World"
```

Если вы хотите загрузить скрипт при открытии IEx REPL, вы можете сделать это:

```
$ iex script.exs
```

Данный `script.exs` - ваш скрипт. Теперь вы можете вызывать функции из сценария в консоли.

Прочитайте Начало работы с Elixir Language онлайн: https://riptutorial.com/ru/elixir/topic/954/начало-работы-с-elixir-language

глава 2: Conditionals

замечания

Обратите внимание, что синтаксис `do...end` - это синтаксический сахар для регулярных списков ключевых слов, поэтому вы действительно можете это сделать:

```
unless false, do: IO.puts("Condition is false")
# Outputs "Condition is false"

# With an `else`:
if false, do: IO.puts("Condition is true"), else: IO.puts("Condition is false")
# Outputs "Condition is false"
```

Examples

дело

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

`case` используется только для соответствия данному шаблону конкретных данных. Здесь `{1,2}` сопоставляется с другим шаблоном case, который приведен в примере кода.

если и если

```
if true do
    "Will be seen since condition is true."
end

if false do
    "Won't be seen since condition is false."
else
    "Will be seen.
end

unless false do
    "Will be seen."
end

unless true do
    "Won't be seen."
else
    "Will be seen."
```

```
end
```

конд

```
cond do
   0 == 1 -> IO.puts "0 = 1"
   2 == 1 + 1 -> IO.puts "1 + 1 = 2"
   3 == 1 + 2 -> IO.puts "1 + 2 = 3"
end

# Outputs "1 + 1 = 2" (first condition evaluating to true)
```

`cond` создаст `CondClauseError` если условия не верны.

```
cond do
  1 == 2 -> "Hmmm"
  "foo" == "bar" -> "What?"
end
# Error
```

Этого можно избежать, добавив условие, которое всегда будет истинным.

```
cond do
  ... other conditions
  true -> "Default value"
end
```

Если это никогда не ожидается, чтобы достичь случая по умолчанию, и программа должна на самом деле сбой в этот момент.

с пунктом

`with` предложением используется для комбинирования совпадающих предложений. Похоже, мы объединяем анонимные функции или обрабатываем функцию с несколькими телами (соответствующие предложения). Рассмотрим случай: мы создаем пользователя, вставляем его в БД, затем создаем приветственное письмо и отправляем его пользователю.

Без `with` пунктом мы могли бы написать что - то вроде этого (я опущена функции реализации):

```
case create_user(user_params) do
  {:ok, user} ->
    case Mailer.compose_email(user) do
      {:ok, email} ->
        Mailer.send_email(email)
      {:error, reason} ->
        handle_error
    end
  {:error, changeset} ->
```

```
    handle_error
end
```

Здесь мы обрабатываем поток нашего бизнес-процесса с помощью `case` (это может быть `cond` или `if`). Это приводит нас к так называемой «пирамиде обречения» , потому что нам приходится иметь дело с возможными условиями и решать: двигаться дальше или нет. Было бы гораздо лучше переписать этот код с `with` инструкции:

```
with {:ok, user} <- create_user(user_params),
     {:ok, email} <- Mailer.compose_email(user) do
  {:ok, Mailer.send_email}
else
  {:error, _reason} ->
    handle_error
end
```

В фрагменте кода выше мы переписываем вложенные предложения `case` с `with` . В `with` Запустим некоторые функции (либо анонимные или именованные) и сопоставление с образцом на их выходах. Если все согласовано, `with` возвратом `do` результат блока или `else` блокируйте результат в противном случае.

Мы можем опустить `else` так `with` будет возвращать либо `do` блок результат или первый сбой результат.

Таким образом, значение оператора `with` - `do` результат `do` блока.

Прочитайте Conditionals онлайн: https://riptutorial.com/ru/elixir/topic/2118/conditionals

глава 3: Doctests

Examples

Вступление

Когда вы документируете свой код с помощью `@doc` , вы можете привести примеры кода следующим образом:

```
# myproject/lib/my_module.exs

defmodule MyModule do
  @doc """
  Given a number, returns `true` if the number is even, otherwise `false`.

  ## Example
    iex> MyModule.even?(2)
    true
    iex> MyModule.even?(3)
    false
  """
  def even?(number) do
    rem(number, 2) == 0
  end
end
```

Вы можете добавить примеры кода в качестве тестовых примеров в один из тестовых наборов:

```
# myproject/test/doc_test.exs

defmodule DocTest do
  use ExUnit.Case
  doctest MyModule
end
```

Затем вы можете запустить свои тесты с помощью `mix test` .

Создание HTML-документации на основе доктрины

Поскольку генерация документации основана на уценке, вы должны сделать 2 вещи:

1 / Напишите свой доктрист и сделайте ваши примеры доктрины понятными для повышения удобочитаемости (лучше дать заголовок, например «примеры» или «тесты»). Когда вы пишете свои тесты, не забудьте указать 4 пробела для вашего кода тестов, чтобы он форматировался как код в документации HTML.

2 / Затем введите «mix docs» в консоли в корне вашего проекта elixir для создания документации HTML в каталоге doc, расположенной в корне вашего проекта elixir.

```
$> mix docs
```

Многолинейные доктрины

Вы можете сделать многострочный доктриум, используя «...>» для строк, следующих за первым

```
iex> Foo.Bar.somethingConditional("baz")
...>   |> case do
...>       {:ok, _} -> true
...>       {:error, _} -> false
...>      end
true
```

Прочитайте Doctests онлайн: https://riptutorial.com/ru/elixir/topic/2708/doctests

глава 4: Ecto

Examples

Добавление Ecto.Repo в программу эликсира

Это можно сделать в 3 этапа:

1. Вы должны определить модуль эликсира, который использует Ecto.Repo и зарегистрировать ваше приложение как otp_app.

```
defmodule Repo do
  use Ecto.Repo, otp_app: :custom_app
end
```

2. Вы также должны определить некоторую конфигурацию для Repo, которая позволит вам подключиться к базе данных. Вот пример с postgres.

```
config :custom_app, Repo,
   adapter: Ecto.Adapters.Postgres,
   database: "ecto_custom_dev",
   username: "postgres_dev",
   password: "postgres_dev",
   hostname: "localhost",
  # OR use a URL to connect instead
  url: "postgres://postgres_dev:postgres_dev@localhost/ecto_custom_dev"
```

3. Прежде чем использовать Ecto в своем приложении, вам необходимо убедиться, что Ecto запущен до запуска вашего приложения. Это можно сделать, зарегистрировав Ecto в lib / custom_app.ex как супервизор.

```
    def start(_type, _args) do
      import Supervisor.Spec

      children = [
       supervisor(Repo, [])
      ]

      opts = [strategy: :one_for_one, name: MyApp.Supervisor]
      Supervisor.start_link(children, opts)
    end
```

"и" в Repo.get_by / 3

Если у вас есть Ecto.Queryable с именем Post, у которого есть заголовок и описание.

Вы можете получить сообщение с заголовком: «привет» и описание: «мир», выполнив:

```
MyRepo.get_by(Post, [title: "hello", description: "world"])
```

Все это возможно, потому что Repo.get_by ожидает во втором аргументе списка ключевых слов.

Запрос с динамическими полями

Чтобы запросить поле, имя которого содержится в переменной, используйте функцию поля .

```
some_field = :id
some_value = 10

from p in Post, where: field(p, ^some_field) == ^some_value
```

Добавление настраиваемых типов данных для миграции и схемы

(Из этого ответа)

В приведенном ниже примере добавляется перечисляемый тип в базу данных postgres.

Сначала отредактируйте **файл миграции** (созданный с помощью `mix ecto.gen.migration`):

```
def up do
  # creating the enumerated type
  execute("CREATE TYPE post_status AS ENUM ('published', 'editing')")

  # creating a table with the column
  create table(:posts) do
    add :post_status, :post_status, null: false
  end
end

def down do
  drop table(:posts)
  execute("DROP TYPE post_status")
end
```

Во-вторых, в **файле модели** добавьте поле с типом Elixir:

```
schema "posts" do
  field :post_status, :string
end
```

или реализовать поведение `Ecto.Type` .

Хорошим примером для последнего является пакет `ecto_enum` и его можно использовать в качестве шаблона. Его использование хорошо документировано на **странице github** .

Эта фиксация показывает пример использования в проекте Phoenix из добавления

enum_ecto в проект и использования перечисляемого типа в представлениях и моделях.

Прочитайте Ecto онлайн: https://riptutorial.com/ru/elixir/topic/6524/ecto

глава 5: Erlang

Examples

Использование Erlang

Модули Erlang доступны как атомы. Например, математический модуль Erlang доступен как `:math` :

```
iex> :math.pi
3.141592653589793
```

Проверьте модуль Erlang

Используйте `module_info` для модулей Erlang, которые вы хотите проверить:

```
iex> :math.module_info
 [module: :math,
  exports: [pi: 0, module_info: 0, module_info: 1, pow: 2, atan2: 2, sqrt: 1,
   log10: 1, log2: 1, log: 1, exp: 1, erfc: 1, erf: 1, atanh: 1, atan: 1,
   asinh: 1, asin: 1, acosh: 1, acos: 1, tanh: 1, tan: 1, sinh: 1, sin: 1,
   cosh: 1, cos: 1],
  attributes: [vsn: [113168357788724588783826225069997113388]],
  compile: [options: [{:outdir,
     '/private/tmp/erlang20160316-36404-xtp7cq/otp-OTP-18.3/lib/stdlib/src/../ebin'},
    {:i,
     '/private/tmp/erlang20160316-36404-xtp7cq/otp-OTP-18.3/lib/stdlib/src/../include'},
    {:i,
     '/private/tmp/erlang20160316-36404-xtp7cq/otp-OTP-
18.3/lib/stdlib/src/../../kernel/include'},
    :warnings_as_errors, :debug_info], version: '6.0.2',
   time: {2016, 3, 16, 16, 40, 35},
   source: '/private/tmp/erlang20160316-36404-xtp7cq/otp-OTP-18.3/lib/stdlib/src/math.erl'],
  native: false,
  md5: <<85, 35, 110, 210, 174, 113, 103, 228, 63, 252, 81, 27, 224, 15, 64,
    44>>]
```

Прочитайте Erlang онлайн: https://riptutorial.com/ru/elixir/topic/2716/erlang

глава 6: ExDoc

Examples

Вступление

Чтобы генерировать документацию в формате `HTML` из `@doc` и `@moduledoc` в исходном коде, добавьте `ex_doc` и процессор уценки, прямо сейчас ExDoc поддерживает Earmark , Pandoc , Hoedown и Cmark в качестве зависимостей в файле `mix.exs` :

```
# config/mix.exs

def deps do
  [{:ex_doc, "~> 0.11", only: :dev},
   {:earmark, "~> 0.1", only: :dev}]
end
```

Если вы хотите использовать другой процессор Markdown, вы можете найти дополнительную информацию в разделе « Изменение инструмента Markdown ».

Вы можете использовать Markdown в `@doc` Elixir `@doc` и `@moduledoc` .

Затем запустите `mix docs` .

Следует иметь в виду, что ExDoc позволяет параметры конфигурации, такие как:

```
  def project do
    [app: :my_app,
     version: "0.1.0-dev",
     name: "My App",
     source_url: "https://github.com/USER/APP",
     homepage_url: "http://YOUR_PROJECT_HOMEPAGE",
     deps: deps(),
     docs: [logo: "path/to/logo.png",
            output: "docs",
            main: "README",
            extra_section: "GUIDES",
            extras: ["README.md", "CONTRIBUTING.md"]]]
  end
```

Вы можете увидеть дополнительную информацию об этих параметрах конфигурации с `mix help docs`

Прочитайте ExDoc онлайн: https://riptutorial.com/ru/elixir/topic/3582/exdoc

глава 7: ExUnit

Examples

Утверждение исключений

Используйте `assert_raise` для проверки того, создано ли исключение. `assert_raise` принимает исключение и функцию, которая должна быть выполнена.

```
test "invalid block size" do
  assert_raise(MerkleTree.ArgumentError, (fn() -> MerkleTree.new ["a", "b", "c"] end))
end
```

Оберните любой код, который вы хотите протестировать в анонимной функции, и передайте его в `assert_raise` .

Прочитайте ExUnit онлайн: https://riptutorial.com/ru/elixir/topic/3583/exunit

глава 8: Sigils

Examples

Создайте список строк

```
iex> ~w(a b c)
["a", "b", "c"]
```

Составьте список атомов

```
iex> ~w(a b c)a
[:a, :b, :c]
```

Пользовательские сигилы

Пользовательские сигилы можно сделать, создав метод `sigil_X` где X - это буква, которую вы хотите использовать (это может быть только одна буква).

```
defmodule Sigils do
  def sigil_j(string, options) do
    # Split on the letter p, or do something more useful
    String.split string, "p"
  end
  # Use this sigil in this module, or import it to use it elsewhere
end
```

Аргумент `options` - это двоичный код аргументов, приведенных в конце сигилы, например:

```
~j/foople/abc # string is "foople", options are 'abc'
# ["foo", "le"]
```

Прочитайте Sigils онлайн: https://riptutorial.com/ru/elixir/topic/2204/sigils

глава 9: Вершины

Examples

Список всех видимых узлов в системе

```
iex(bob@127.0.0.1)> Node.list
[:"frank@127.0.0.1"]
```

Подключение узлов на одном компьютере

Начните два именованных узла в двух терминальных окнах:

```
>iex --name bob@127.0.0.1
iex(bob@127.0.0.1)>
>iex --name frank@127.0.0.1
iex(frank@127.0.0.1)>
```

Подключите два узла, указав один узел для подключения:

```
iex(bob@127.0.0.1)> Node.connect :"frank@127.0.0.1"
true
```

Два узла теперь подключены и знают друг о друге:

```
iex(bob@127.0.0.1)> Node.list
[:"frank@127.0.0.1"]
iex(frank@127.0.0.1)> Node.list
[:"bob@127.0.0.1"]
```

Вы можете выполнить код на других узлах:

```
iex(bob@127.0.0.1)> greet = fn() -> IO.puts("Hello from #{inspect(Node.self)}") end
iex(bob@127.0.0.1)> Node.spawn(:"frank@127.0.0.1", greet)
#PID<9007.74.0>
Hello from :"frank@127.0.0.1"
:ok
```

Подключение узлов на разных машинах

Запустите именованный процесс на одном IP-адресе:

```
$ iex --name foo@10.238.82.82 --cookie chocolate
iex(foo@10.238.82.82)> Node.ping :"bar@10.238.82.85"
:pong
iex(foo@10.238.82.82)> Node.list
[:"bar@10.238.82.85"]
```

Запустите другой именованный процесс на другом IP-адресе:

```
$ iex --name bar@10.238.82.85 --cookie chocolate
iex(bar@10.238.82.85)> Node.list
[:"foo@10.238.82.82"]
```

Прочитайте Вершины онлайн: https://riptutorial.com/ru/elixir/topic/2065/вершины

глава 10: Встроенные типы

Examples

чисел

Elixir поставляется с **целыми числами** и **числами** с **плавающей запятой** . **Целочисленный литерал** может быть записан в десятичных, двоичных, восьмеричных и шестнадцатеричных форматах.

```
iex> x = 291
291

iex> x = 0b100100011
291

iex> x = 0o443
291

iex> x = 0x123
291
```

Поскольку Elixir использует арифметику бигнума, **диапазон целых чисел ограничен только доступной памятью в системе** .

Номера с плавающей точкой являются двойной точностью и соответствуют спецификации IEEE-754.

```
iex> x = 6.8
6.8

iex> x = 1.23e-11
1.23e-11
```

Обратите внимание, что Elixir также поддерживает экспоненциальную форму для float.

```
iex> 1 + 1
2

iex> 1.0 + 1.0
2.0
```

Сначала мы добавили два целых числа, а результат - целое число. Позже мы добавили два числа с плавающей запятой, а результат - число с плавающей запятой.

Разделение в Elixir всегда возвращает число с плавающей запятой:

```
iex> 10 / 2
```

```
5.0
```

Точно так же, если вы добавляете, вычитаете или умножаете целое число на число с плавающей запятой, результат будет плавающей точкой:

```
iex> 40.0 + 2
42.0

iex> 10 - 5.0
5.0

iex> 3 * 3.0
9.0
```

Для целочисленного деления можно использовать функцию `div/2` :

```
iex> div(10, 2)
5
```

атомы

Атомы - это константы, которые представляют собой имя какой-то вещи. Значение атома - это имя. Имя атома начинается с двоеточия.

```
:atom   # that's how we define an atom
```

Имя атома уникально. Два атома с одинаковыми именами всегда равны.

```
iex(1)> a = :atom
:atom

iex(2)> b = :atom
:atom

iex(3)> a == b
true

iex(4)> a === b
true
```

Логические значения `true` и `false` , фактически являются атомами.

```
iex(1)> true == :true
true

iex(2)> true === :true
true
```

Атомы хранятся в специальной таблице атомов. Очень важно знать, что эта таблица не собирает мусор. Итак, если вы хотите (или случайно это факт) постоянно создавать атомы - это плохая идея.

Бинарные и бинты

Бинарники в эликсире создаются с использованием конструкции Kernel.SpecialForms << >> .

Это мощный инструмент, который делает Elixir очень полезным для работы с бинарными протоколами и кодировками.

Бинарники и битовые строки задаются с использованием списка целых чисел или значений переменных с разделителями-запятыми, задокументированных «<<» и «>>». Они состоят из «единиц», либо группировки битов, либо группировки байтов. Группировка по умолчанию - это один байт (8 бит), указанный с использованием целого числа:

```
<<222,173,190, 239>> # 0xDEADBEEF
```

Эликсирные струны также преобразуются непосредственно в двоичные файлы:

```
iex> <<0, "foo">>
<<0, 102, 111, 111>>
```

Вы можете добавить «спецификаторы» к каждому «сегменту» двоичного кода, что позволяет вам кодировать:

- Тип данных
- Размер
- Порядок байтов

Эти спецификаторы кодируются, следуя каждому значению или переменной с помощью оператора «::»:

```
<<102::integer-native>>
<<102::native-integer>> # Same as above
<<102::unsigned-big-integer>>
<<102::unsigned-big-integer-size(8)>>
<<102::unsigned-big-integer-8>> # Same as above
<<102::8-integer-big-unsigned>>
<<-102::signed-little-float-64>> # -102 as a little-endian Float64
<<-102::native-little-float-64>> # -102 as a Float64 for the current machine
```

Доступные типы данных, которые вы можете использовать:

- целое число
- поплавок
- бит (псевдоним для битовой строки)
- битовая
- двоичный
- байты (псевдоним для двоичного кода)

- utf8
- utf16
- UTF32

Имейте в виду, что при указании «размера» двоичного сегмента он изменяется в соответствии с «типом», выбранным в спецификаторе сегмента:

- integer (по умолчанию) 1 бит
- float 1 бит
- двоичные 8 бит

Прочитайте Встроенные типы онлайн: https://riptutorial.com/ru/elixir/topic/1774/встроенные-типы

глава 11: Государственная обработка в эликсире

Examples

Управление частью штата с Агентом

Самый простой способ обернуть и получить доступ к части состояния - это `Agent` . Модуль позволяет создавать процесс, который сохраняет произвольную структуру данных и позволяет отправлять сообщения для чтения и обновления этой структуры. Благодаря этому доступ к структуре автоматически сериализуется, так как процесс обрабатывает только одно сообщение за раз.

```
iex(1)> {:ok, pid} = Agent.start_link(fn -> :initial_value end)
{:ok, #PID<0.62.0>}
iex(2)> Agent.get(pid, &(&1))
:initial_value
iex(3)> Agent.update(pid, fn(value) -> {value, :more_data} end)
:ok
iex(4)> Agent.get(pid, &(&1))
{:initial_value, :more_data}
```

Прочитайте Государственная обработка в эликсире онлайн:
https://riptutorial.com/ru/elixir/topic/6596/государственная-обработка-в-эликсире

глава 12: задача

Синтаксис

- Task.async (весело)
- Task.await (задача)

параметры

параметр	подробности
веселье	Функция, которая должна выполняться в отдельном процессе.
задача	Задача, возвращаемая `Task.async` .

Examples

Выполнение работы в фоновом режиме

```
task = Task.async(fn -> expensive_computation end)
do_something_else
result = Task.await(task)
```

Параллельная обработка

```
crawled_site = ["http://www.google.com", "http://www.stackoverflow.com"]
|> Enum.map(fn site -> Task.async(fn -> crawl(site) end) end)
|> Enum.map(&Task.await/1)
```

Прочитайте задача онлайн: https://riptutorial.com/ru/elixir/topic/7588/задача

глава 13: Карты и списки ключевых слов

Синтаксис

- map =% {} // создает пустую карту
- map =% {: a => 1,: b => 2} // создает непустую карту
- list = [] // создает пустой список
- list = [{: a, 1}, {: b, 2}] // создает список непустых ключевых слов

замечания

Elixir предоставляет две ассоциативные структуры данных: *карты* и *списки ключевых слов* .

Карты - это ключ-значение Elixir (также называемый словарем или хешем в других языках).

Списки ключевых слов представляют собой кортежи ключа / значения, которые связывают значение с определенным ключом. Они обычно используются как опции для вызова функции.

Examples

Создание карты

Карты - это ключ-значение Elixir (также называемый словарем или хешем в других языках). Вы создаете карту, используя синтаксис `%w{}` :

```
%{} // creates an empty map
%{:a => 1, :b => 2} // creates a non-empty map
```

Ключами и значениями могут быть любые типы:

```
%{"a" => 1, "b" => 2}
%{1 => "a", 2 => "b"}
```

Кроме того, вы можете иметь карты со смешанными типами для ключей и значений »:

```
// keys are integer or strings
%{1 => "a", "b" => :foo}
// values are string or nil
%{1 => "a", 2 => nil}
```

Когда все ключи на карте являются атомами, вы можете использовать синтаксис ключевых слов для удобства:

```
%{a: 1, b: 2}
```

Создание списка ключевых слов

Списки ключевых слов представляют собой кортежи ключа / значения, обычно используемые как опции для вызова функции.

```
[{:a, 1}, {:b, 2}] // creates a non-empty keyword list
```

В списках ключевых слов может повторяться один и тот же ключ более одного раза.

```
[{:a, 1}, {:a, 2}, {:b, 2}]
[{:a, 1}, {:b, 2}, {:a, 2}]
```

Ключи и значения могут быть любого типа:

```
[{"a", 1}, {:a, 2}, {2, "b"}]
```

Разница между картами и списками ключевых слов

Карты и списки ключевых слов имеют разные приложения. Например, карта не может иметь две клавиши с одинаковым значением и не упорядочена. И наоборот, список ключевых слов может быть немного сложнее использовать при сопоставлении шаблонов в некоторых случаях.

Вот несколько примеров использования списков сопоставлений и списков ключевых слов.

Используйте списки ключевых слов, когда:

- вам нужны элементы, которые нужно заказать
- вам нужно несколько элементов с одним и тем же ключом

Использовать карты, когда:

- вы хотите сопоставить шаблон с некоторыми ключами / значениями
- вам не нужно больше одного элемента с одним и тем же ключом
- когда вам явно не нужен список ключевых слов

Прочитайте Карты и списки ключевых слов онлайн: https://riptutorial.com/ru/elixir/topic/2706/карты-и-списки-ключевых-слов

глава 14: Константы

замечания

Итак, это сводный анализ, который я сделал на основе методов, перечисленных в разделе Как вы определяете константы в модулях Elixir? , Я отправляю его по нескольким причинам:

- Большинство документов Elixir достаточно тщательны, но я нашел это ключевое архитектурное решение без руководства - поэтому я бы запросил его как тему.
- Я хотел получить небольшую видимость и комментарии от других о теме.
- Я также хотел протестировать новый рабочий процесс SO Documentation. ;)

Я также загрузил весь код в концепцию эликсира-константы GitHub.

Examples

Константы с областью действия модуля

```
defmodule MyModule do
  @my_favorite_number 13
  @use_snake_case "This is a string (use double-quotes)"
end
```

Они доступны только из этого модуля.

Константы как функции

Объявляет:

```
defmodule MyApp.ViaFunctions.Constants do
  def app_version, do: "0.0.1"
  def app_author, do: "Felix Orr"
  def app_info, do: [app_version, app_author]
  def bar, do: "barrific constant in function"
end
```

Потребление с требованием:

```
defmodule MyApp.ViaFunctions.ConsumeWithRequire do
  require MyApp.ViaFunctions.Constants

  def foo() do
    IO.puts MyApp.ViaFunctions.Constants.app_version
    IO.puts MyApp.ViaFunctions.Constants.app_author
    IO.puts inspect MyApp.ViaFunctions.Constants.app_info
  end
```

```
  # This generates a compiler error, cannot invoke `bar/0` inside a guard.
  # def foo(_bar) when is_bitstring(bar) do
  #   IO.puts "We just used bar in a guard: #{bar}"
  # end
end
```

Потребление с импортом:

```
defmodule MyApp.ViaFunctions.ConsumeWithImport do
  import MyApp.ViaFunctions.Constants

  def foo() do
    IO.puts app_version
    IO.puts app_author
    IO.puts inspect app_info
  end
end
```

Этот метод позволяет повторно использовать константы в проектах, но они не будут использоваться в защитных функциях, для которых требуются константы времени компиляции.

Константы через макросы

Объявляет:

```
defmodule MyApp.ViaMacros.Constants do
  @moduledoc """
  Apply with `use MyApp.ViaMacros.Constants, :app` or `import MyApp.ViaMacros.Constants,
:app`.

  Each constant is private to avoid ambiguity when importing multiple modules
  that each have their own copies of these constants.
  """

  def app do
    quote do
      # This method allows sharing module constants which can be used in guards.
      @bar "barrific module constant"
      defp app_version, do: "0.0.1"
      defp app_author, do: "Felix Orr"
      defp app_info, do: [app_version, app_author]
    end
  end

  defmacro __using__(which) when is_atom(which) do
    apply(__MODULE__, which, [])
  end
end
```

Потребляйте с `use` :

```
defmodule MyApp.ViaMacros.ConsumeWithUse do
  use MyApp.ViaMacros.Constants, :app
```

```
  def foo() do
    IO.puts app_version
    IO.puts app_author
    IO.puts inspect app_info
  end

  def foo(_bar) when is_bitstring(@bar) do
    IO.puts "We just used bar in a guard: #{@bar}"
  end
end
```

Этот метод позволяет использовать `@some_constant` охранники `@some_constant` . Я даже не уверен, что функции будут строго необходимы.

Прочитайте Константы онлайн: https://riptutorial.com/ru/elixir/topic/6614/константы

глава 15: ЛУЧ

Examples

Вступление

```
iex> :observer.start
:ok
```

`:observer.start` открывает интерфейс наблюдателя GUI, показывая вам разбивку процессора, использование памяти и другую информацию, критическую для понимания шаблонов использования ваших приложений.

Прочитайте ЛУЧ онлайн: https://riptutorial.com/ru/elixir/topic/3587/луч

глава 16: Лучшая отладка с помощью IO.inspect и ярлыков

Вступление

`IO.inspect` очень полезен, когда вы пытаетесь отлаживать `IO.inspect` вызовов методов. Это может стать беспорядочным, если вы используете его слишком часто.

Начиная с Elixir 1.4.0, опция `label IO.inspect` может помочь

замечания

Работает только с Elixir 1.4+, но пока не могу отметить.

Examples

Без ярлыков

```
url
  |> IO.inspect
  |> HTTPoison.get!
  |> IO.inspect
  |> Map.get(:body)
  |> IO.inspect
  |> Poison.decode!
  |> IO.inspect
```

Это приведет к большому количеству результатов без контекста:

```
"https://jsonplaceholder.typicode.com/posts/1"
%HTTPoison.Response{body: "{\n  \"userId\": 1,\n  \"id\": 1,\n  \"title\": \"sunt aut facere
repellat provident occaecati excepturi optio reprehenderit\",\n  \"body\": \"quia et
suscipit\\nsuscipit recusandae consequuntur expedita et cum\\nreprehenderit molestiae ut ut
quas totam\\nnostrum rerum est autem sunt rem eveniet architecto\"\n}",
 headers: [{"Date", "Thu, 05 Jan 2017 14:29:59 GMT"},
  {"Content-Type", "application/json; charset=utf-8"},
  {"Content-Length", "292"}, {"Connection", "keep-alive"},
  {"Set-Cookie",
   "__cfduid=d56d1be0a544fcbdbb262fee9477600c51483626599; expires=Fri, 05-Jan-18 14:29:59 GMT;
path=/; domain=.typicode.com; HttpOnly"},
  {"X-Powered-By", "Express"}, {"Vary", "Origin, Accept-Encoding"},
  {"Access-Control-Allow-Credentials", "true"},
  {"Cache-Control", "public, max-age=14400"}, {"Pragma", "no-cache"},
  {"Expires", "Thu, 05 Jan 2017 18:29:59 GMT"},
  {"X-Content-Type-Options", "nosniff"},
  {"Etag", "W/\"124-yv65LoT2uMHrpn06wNpAcQ\""}, {"Via", "1.1 vegur"},
  {"CF-Cache-Status", "HIT"}, {"Server", "cloudflare-nginx"},
  {"CF-RAY", "31c7a025e94e2d41-TXL"}], status_code: 200}
```

```
"{\n  \"userId\": 1,\n  \"id\": 1,\n  \"title\": \"sunt aut facere repellat provident
occaecati excepturi optio reprehenderit\",\n  \"body\": \"quia et suscipit\\nsuscipit
recusandae consequuntur expedita et cum\\nreprehenderit molestiae ut ut quas totam\\nnostrum
rerum est autem sunt rem eveniet architecto\"\n}"
%{"body" => "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit
molestiae ut ut quas totam\nnostrum rerum est autem sunt rem eveniet architecto",
  "id" => 1,
  "title" => "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
  "userId" => 1}
```

С ярлыками

использование опции `label` для добавления контекста может многое помочь:

```
url
  |> IO.inspect(label: "url")
  |> HTTPoison.get!
  |> IO.inspect(label: "raw http resonse")
  |> Map.get(:body)
  |> IO.inspect(label: "raw body")
  |> Poison.decode!
  |> IO.inspect(label: "parsed body")

url: "https://jsonplaceholder.typicode.com/posts/1"
raw http resonse: %HTTPoison.Response{body: "{\n  \"userId\": 1,\n  \"id\": 1,\n  \"title\":
\"sunt aut facere repellat provident occaecati excepturi optio reprehenderit\",\n  \"body\":
\"quia et suscipit\\nsuscipit recusandae consequuntur expedita et cum\\nreprehenderit
molestiae ut ut quas totam\\nnostrum rerum est autem sunt rem eveniet architecto\"\n}",
 headers: [{"Date", "Thu, 05 Jan 2017 14:33:06 GMT"},
  {"Content-Type", "application/json; charset=utf-8"},
  {"Content-Length", "292"}, {"Connection", "keep-alive"},
  {"Set-Cookie",
   "__cfduid=d22d817e48828169296605d27270af7e81483626786; expires=Fri, 05-Jan-18 14:33:06 GMT;
path=/; domain=.typicode.com; HttpOnly"},
  {"X-Powered-By", "Express"}, {"Vary", "Origin, Accept-Encoding"},
  {"Access-Control-Allow-Credentials", "true"},
  {"Cache-Control", "public, max-age=14400"}, {"Pragma", "no-cache"},
  {"Expires", "Thu, 05 Jan 2017 18:33:06 GMT"},
  {"X-Content-Type-Options", "nosniff"},
  {"Etag", "W/\"124-yv65LoT2uMHrpn06wNpAcQ\""}, {"Via", "1.1 vegur"},
  {"CF-Cache-Status", "HIT"}, {"Server", "cloudflare-nginx"},
  {"CF-RAY", "31c7a4b8ae042d77-TXL"}], status_code: 200}
raw body: "{\n  \"userId\": 1,\n  \"id\": 1,\n  \"title\": \"sunt aut facere repellat
provident occaecati excepturi optio reprehenderit\",\n  \"body\": \"quia et
suscipit\\nsuscipit recusandae consequuntur expedita et cum\\nreprehenderit molestiae ut ut
quas totam\\nnostrum rerum est autem sunt rem eveniet architecto\"\n}"
parsed body: %{"body" => "quia et suscipit\nsuscipit recusandae consequuntur expedita et
cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem sunt rem eveniet
architecto",
  "id" => 1,
  "title" => "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
  "userId" => 1}
```

Прочитайте Лучшая отладка с помощью IO.inspect и ярлыков онлайн:
https://riptutorial.com/ru/elixir/topic/8725/лучшая-отладка-с-помощью-io-inspect-и-ярлыков

глава 17: Метапрограммирование

Examples

Сгенерировать тесты во время компиляции

```
defmodule ATest do
  use ExUnit.Case

  [{1, 2, 3}, {10, 20, 40}, {100, 200, 300}]
  |> Enum.each(fn {a, b, c} ->
    test "#{a} + #{b} = #{c}" do
      assert unquote(a) + unquote(b) = unquote(c)
    end
  end)
end
```

Выход:

```
.

  1) test 10 + 20 = 40 (Test.Test)
     test.exs:6
     match (=) failed
     code: 10 + 20 = 40
     rhs:  40
     stacktrace:
       test.exs:7

.

Finished in 0.1 seconds (0.1s on load, 0.00s on tests)
3 tests, 1 failure
```

Прочитайте Метапрограммирование онлайн: https://riptutorial.com/ru/elixir/topic/4069/метапрограммирование

глава 18: микшировать

Examples

Создание задачи пользовательского смешивания

```
# lib/mix/tasks/mytask.ex
defmodule Mix.Tasks.MyTask do
  use Mix.Task

  @shortdoc "A simple mix task"
  def run(_) do
    IO.puts "YO!"
  end
end
```

Скомпилировать и запустить:

```
$ mix compile
$ mix my_task
"YO!"
```

Пользовательская задача смешения с аргументами командной строки

В базовой реализации модуль задачи должен определить функцию `run/1` которая принимает список аргументов. Например, `def run(args) do ... end`

```
defmodule Mix.Tasks.Example_Task do
  use Mix.Task

  @shortdoc "Example_Task prints hello + its arguments"
  def run(args) do
    IO.puts "Hello #{args}"
  end
end
```

Скомпилировать и запустить:

```
$ mix example_task world
"hello world"
```

Псевдонимы

Elixir позволяет добавлять псевдонимы для ваших команд микширования. Прохладная вещь, если вы хотите сохранить себе некоторую типизацию.

Откройте `mix.exs` в проекте Elixir.

Сначала добавьте функцию `aliases/0` в список ключевых слов, возвращаемый функцией `project` . *Добавление `()` в конце функции псевдонимов не позволит компилятору отбросить предупреждение.*

```
def project do
  [app: :my_app,
   ...
   aliases: aliases()]
end
```

Затем определите свою функцию `aliases/0` (например, в нижней части файла `mix.exs`).

```
...

defp aliases do
  [go: "phoenix.server",
   trident: "do deps.get, compile, go"]
end
```

Теперь вы можете использовать `$ mix go` для запуска вашего Phoenix-сервера (если вы используете приложение Phoenix). И используйте `$ mix trident` чтобы сообщить mix, чтобы получить все зависимости, скомпилировать и запустить сервер.

Получение справки о доступных задачах микширования

Для отображения доступных задач микширования используйте:

```
mix help
```

Чтобы получить помощь по конкретной задаче, используйте `mix help task` например:

```
mix help cmd
```

Прочитайте микшировать онлайн: https://riptutorial.com/ru/elixir/topic/3585/микшировать

глава 19: Модули

замечания

Имена модулей

В Elixir имена модулей, такие как `IO` или `String` являются атомами под капотом и преобразуются в форму `:"Elixir.ModuleName"` во время компиляции.

```
iex(1)> is_atom(IO)
true
iex(2)> IO == :"Elixir.IO"
true
```

Examples

Список функций или макросов модуля

Функция `__info__/1` принимает один из следующих атомов:

- `:functions` - возвращает список ключевых слов с публичными функциями вместе со своими задачами
- `:macros` Возвращает список ключевых слов с общедоступными макросами вместе с их свойствами.

Чтобы перечислить функции модуля `Kernel` :

```
iex> Kernel.__info__ :functions
[!=: 2, !==: 2, *: 2, +: 1, +: 2, ++: 2, -: 1, -: 2, --: 2, /: 2, <: 2, <=: 2,
 ==: 2, ===: 2, =~: 2, >: 2, >=: 2, abs: 1, apply: 2, apply: 3, binary_part: 3,
 bit_size: 1, byte_size: 1, div: 2, elem: 2, exit: 1, function_exported?: 3,
 get_and_update_in: 3, get_in: 2, hd: 1, inspect: 1, inspect: 2, is_atom: 1,
 is_binary: 1, is_bitstring: 1, is_boolean: 1, is_float: 1, is_function: 1,
 is_function: 2, is_integer: 1, is_list: 1, is_map: 1, is_number: 1, is_pid: 1,
 is_port: 1, is_reference: 1, is_tuple: 1, length: 1, macro_exported?: 3,
 make_ref: 0, ...]
```

Замените `Kernel` любым выбранным вами модулем.

Использование модулей

Модули имеют четыре связанных ключевых слова, чтобы использовать их в других модулях: `alias` , `import` , `use` и `require` .

`alias` будет регистрировать модуль под другим (обычно коротким) именем:

```
defmodule MyModule do
  # Will make this module available as `CoolFunctions`
  alias MyOtherModule.CoolFunctions
  # Or you can specify the name to use
  alias MyOtherModule.CoolFunctions, as: CoolFuncs
end
```

`import` сделает все функции в модуле доступными без имени перед ними:

```
defmodule MyModule do
  import Enum
  def do_things(some_list) do
    # No need for the `Enum.` prefix
    join(some_list, " ")
  end
end
```

`use` позволяет модулю вводить код в текущий модуль - обычно это делается как часть структуры, которая создает свои собственные функции, чтобы ваш модуль подтвердил какое-либо поведение.

`require` загрузки макросов из модуля, чтобы их можно было использовать.

Передача функций другому модулю

Используйте `defdelegate` для определения функций, которые делегируются функциям с тем же именем, определенным в другом модуле:

```
defmodule Math do
  defdelegate pi, to: :math
end


iex> Math.pi
3.141592653589793
```

Прочитайте Модули онлайн: https://riptutorial.com/ru/elixir/topic/2721/модули

глава 20: Монтаж

Examples

Установка Fedora

```
dnf install erlang elixir
```

Установка OSX

В OS X и MacOS Elixir можно установить через общих менеджеров пакетов:

Homebrew

```
$ brew update
$ brew install elixir
```

MacPorts

```
 $ sudo port install elixir
```

Установка Debian / Ubuntu

```
# Fetch and install package to setup access to the official APT repository
wget https://packages.erlang-solutions.com/erlang-solutions_1.0_all.deb
sudo dpkg -i erlang-solutions_1.0_all.deb

# Update package index
sudo apt-get update

# Install Erlang and Elixir
sudo apt-get install esl-erlang
sudo apt-get install elixir
```

Установка Gentoo / Funtoo

Эликсир доступен в основном репозитории пакетов.
Обновите список пакетов перед установкой любого пакета:

```
emerge --sync
```

Это одноэтапная установка:

```
emerge --ask dev-lang/elixir
```

Прочитайте Монтаж онлайн: https://riptutorial.com/ru/elixir/topic/4208/монтаж

глава 21: операторы

Examples

Оператор труб

Оператор трубы `|>` принимает результат выражения слева и передает его в качестве первого параметра функции справа.

```
expression |> function
```

Используйте оператор трубы для объединения выражений вместе и визуального документирования потока ряда функций.

Рассмотрим следующее:

```
Oven.bake(Ingredients.Mix([:flour, :cocoa, :sugar, :milk, :eggs, :butter]), :temperature)
```

В примере, `Oven.bake` приходит до `Ingredients.mix` , но выполняется последним. Кроме того, не может быть очевидно, что `:temperature` является параметром `Oven.bake`

Переписывая этот пример с помощью оператора труб:

```
[:flour, :cocoa, :sugar, :milk, :eggs, :butter]
|> Ingredients.mix
|> Oven.bake(:temperature)
```

дает тот же результат, но порядок выполнения более ясен. Кроме того, ясно, что `:temperature` является параметром для вызова `Oven.bake` .

Обратите внимание, что при использовании оператора трубок первый параметр для каждой функции перемещается до оператора трубы, и поэтому вызываемая функция имеет один меньший параметр. Например:

```
Enum.each([1, 2, 3], &(&1+1)) # produces [2, 3, 4]
```

такой же как:

```
[1, 2, 3]
|> Enum.each(&(&1+1))
```

Оператор и круглые скобы

Круглые скобки необходимы, чтобы избежать двусмысленности:

```
foo 1 |> bar 2 |> baz 3
```

Должен быть написан как:

```
foo(1) |> bar(2) |> baz(3)
```

Булевы операторы

В Эликсире существует два типа булевых операторов:

- логические операторы (они ожидают, что в качестве первого аргумента они будут либо `true` либо `false`)

```
x or y      # true if x is true, otherwise y

x and y     # false if x is false, otherwise y

not x       # false if x is true, otherwise true
```

Все логические операторы будут поднимать `ArgumentError` если первый аргумент не будет строго логическим значением, что означает только `true` или `false` (`nil` не является логическим).

```
iex(1)> false and 1 # return false
iex(2)> false or 1 # return 1
iex(3)> nil and 1 # raise (ArgumentError) argument error: nil
```

- релаксированные булевы операторы (работа с любым типом, все, что ни `false` ни `nil` считается `true`)

```
x || y     # x if x is true, otherwise y

x && y     # y if x is true, otherwise false

!x         # false if x is true, otherwise true
```

Оператор `||` всегда будет возвращать первый аргумент, если он правдивый (Elixir рассматривает все, кроме `nil` и `false` чтобы быть истинным в сравнении), в противном случае вернется второй.

```
iex(1)> 1 || 3 # return 1, because 1 is truthy
iex(2)> false || 3 # return 3
iex(3)> 3 || false # return 3
iex(4)> false || nil # return nil
iex(5)> nil || false # return false
```

Оператор `&&` всегда будет возвращать второй аргумент, если он прав. В противном случае возвращаются соответственно аргументам, `false` или `nil` .

```
iex(1)> 1 && 3 # return 3, first argument is truthy
iex(2)> false && 3 # return false
iex(3)> 3 && false # return false
iex(4)> 3 && nil # return nil
iex(5)> false && nil # return false
iex(6)> nil && false # return nil
```

И `&&` и `||` являются операторами короткого замыкания. Они выполняют только правую сторону, если левой части недостаточно, чтобы определить результат.

Оператор `!` будет возвращать логическое значение отрицания текущего термина:

```
iex(1)> !2 # return false
iex(2)> !false # return true
iex(3)> !"Test" # return false
iex(4)> !nil # return true
```

Простой способ получить логическое значение выбранного термина - просто удвоить этот оператор:

```
iex(1)> !!true # return true
iex(2)> !!"Test" # return true
iex(3)> !!nil # return false
iex(4)> !!false # return false
```

Операторы сравнения

Равенство:

- value равен `x == y` (`1 == 1.0 # true`)
- значение неравенства `x == y` (`1 != 1.0 # false`)
- строгое равенство `x === y` (`1 === 1.0 # false`)
- строгое неравенство `x === y` (`1 !== 1.0 # true`)

Сравнение:

- `x > y`
- `x >= y`
- `x < y`
- `x <= y`

Если типы совместимы, сравнение использует естественный порядок. В противном случае существует правило сравнения общих типов:

```
number < atom < reference < function < port < pid < tuple < map < list < binary
```

Присоединяйтесь

Вы можете объединять (объединить) двоичные файлы (включая строки) и списки:

```
iex(1)> [1, 2, 3] ++ [4, 5]
[1, 2, 3, 4, 5]

iex(2)> [1, 2, 3, 4, 5] -- [1, 3]
[2, 4, 5]

iex(3)> "qwe" <> "rty"
"qwerty"
```

Оператор «В»

`in` операторе позволяет проверить, содержит ли список или диапазон элемент:

```
iex(4)> 1 in [1, 2, 3, 4]
true

iex(5)> 0 in (1..5)
false
```

Прочитайте операторы онлайн: https://riptutorial.com/ru/elixir/topic/1161/операторы

глава 22: оптимизация

Examples

Всегда измерьте сначала!

Это общие советы, которые в целом улучшают производительность. Если ваш код медленный, всегда важно профилировать его, чтобы выяснить, какие части медленны. Угадать **никогда не бывает** достаточно. Улучшение скорости выполнения чего-то, что занимает только 1% времени выполнения, вероятно, не стоит усилий. Ищите большие раковины.

Чтобы получить несколько точные цифры, убедитесь, что код, который вы оптимизируете, выполняется не менее одной секунды при профилировании. Если вы тратите 10% времени выполнения этой функции, убедитесь, что полное выполнение программы занимает не менее 10 секунд и убедитесь, что вы можете выполнять одни и те же точные данные через код несколько раз, чтобы получить повторяющиеся числа.

ExProf с этим легко начать.

Прочитайте оптимизация онлайн: https://riptutorial.com/ru/elixir/topic/6062/оптимизация

глава 23: основное использование охранных оговорок

Examples

основные применения защитных оговорок

В Elixir можно создать несколько реализаций функции с тем же именем и указать правила, которые будут применяться к параметрам функции *перед вызовом функции* , чтобы определить, какую реализацию выполнить.

Эти правила отмечены ключевым словом `when` , и они идут между `def function_name(params)` и `do` в определении функции. Тривиальный пример:

```
defmodule Math do

  def is_even(num) when num === 1 do
    false
  end
  def is_even(num) when num === 2 do
    true
  end

  def is_odd(num) when num === 1 do
    true
  end
  def is_odd(num) when num === 2 do
    false
  end

end
```

Скажем, я запустил `Math.is_even(2)` этом примере. Существуют две реализации `is_even` , с различными предложениями охраны. Система будет смотреть на них по порядку и запускать первую реализацию, где параметры удовлетворяют условию охраны. Первый указывает, что `num === 1` который не является истинным, поэтому он переходит к следующему. Второй указывает, что `num === 2` , что верно, поэтому это реализация, которая используется, и возвращаемое значение будет `true` .

Что делать, если я запускаю `Math.is_odd(1)` ? Система смотрит на первую реализацию и видит, что, поскольку `num` равно `1` выполняется условие охраны первой реализации. Затем он будет использовать эту реализацию и вернет `true` , и не будет интересоваться любыми другими реализациями.

Охранники ограничены в типах операций, которые они могут выполнять. В документации Elixir перечислены все разрешенные операции ; в двух словах они позволяют сравнивать,

математику, двоичные операции, проверку типов (например, `is_atom`) и несколько небольших удобных функций (например, `length`). Можно определить пользовательские предложения охраны, но для этого требуется создание макросов, и лучше всего использовать более продвинутое руководство.

Обратите внимание, что охранники не выдают ошибок; они рассматриваются как обычные отказы в предложении охраны, и система переходит к следующей реализации. Если вы обнаружите, что вы получаете `(FunctionClauseError) no function clause matching` при вызове защищенной функции с параметрами, которые вы ожидаете работать, может быть, что предложение охраны, которое вы ожидаете работать, вызывает ошибку, которая проглатывается.

Чтобы увидеть это для себя, создайте, а затем вызовите функцию с защитой, которая не имеет смысла, например, которая пытается делить на ноль:

```
defmodule BadMath do
  def divide(a) when a / 0 === :foo do
    :bar
  end
end
```

Вызов `BadMath.divide("anything")` предоставит некоторую бесполезную ошибку `(FunctionClauseError) no function clause matching in BadMath.divide/1` тогда как если бы вы пытались запустить `"anything" / 0` напрямую, вы получили бы более полезную error: `(ArithmeticError) bad argument in arithmetic expression`.

Прочитайте основное использование охранных оговорок онлайн:
https://riptutorial.com/ru/elixir/topic/6121/основное-использование-охранных-оговорок

глава 24: Основной .gitignore для программы elixir

Прочитайте Основной .gitignore для программы elixir онлайн:
https://riptutorial.com/ru/elixir/topic/6493/основной--gitignore-для-программы-elixir

глава 25: Основной .gitignore для программы elixir

замечания

Обратите внимание, что папка `/rel` может не понадобиться в вашем файле .gitignore. Это генерируется, если вы используете средство управления выпуском, например, `exrm`

Examples

Основной .gitignore для Elixir

```
/_build
/cover
/deps
erl_crash.dump
*.ez

# Common additions for various operating systems:
# MacOS
.DS_Store

# Common additions for various editors:
# JetBrains IDEA, IntelliJ, PyCharm, RubyMine etc.
.idea
```

пример

```
### Elixir ###
/_build
/cover
/deps
erl_crash.dump
*.ez

### Erlang ###
.eunit
deps
*.beam
*.plt
ebin
rel/example_project
.concrete/DEV_MODE
.rebar
```

Автономное применение эликсира

```
/_build
```

```
/cover
/deps
erl_crash.dump
*.ez
/rel
```

Приложение Phoenix

```
/_build
/db
/deps
/*.ez
erl_crash.dump
/node_modules
/priv/static/
/config/prod.secret.exs
/rel
```

Автогенерация .gitignore

По умолчанию `mix new <projectname>` будет генерировать файл `.gitignore` в корне проекта, подходящем для Elixir.

```
# The directory Mix will write compiled artifacts to.
/_build

# If you run "mix test --cover", coverage assets end up here.
/cover

# The directory Mix downloads your dependencies sources to.
/deps

# Where 3rd-party dependencies like ExDoc output generated docs.
/doc

# If the VM crashes, it generates a dump, let's ignore it too.
erl_crash.dump

# Also ignore archive artifacts (built via "mix archive.build").
*.ez
```

Прочитайте Основной .gitignore для программы elixir онлайн:
https://riptutorial.com/ru/elixir/topic/6526/основной--gitignore-для-программы-elixir

глава 26: поведения

Examples

Вступление

Поведение - это список спецификаций функций, которые может реализовать другой модуль. Они похожи на интерфейсы на других языках.

Вот пример поведения:

```
defmodule Parser do
  @callback parse(String.t) :: any
  @callback extensions() :: [String.t]
end
```

И модуль, который его реализует:

```
defmodule JSONParser do
  @behaviour Parser

  def parse(str), do: # ... parse JSON
  def extensions, do: ["json"]
end
```

`@behaviour` модуля `@behaviour` выше указывает, что этот модуль должен определять каждую функцию, определенную в модуле Parser. Отсутствующие функции приведут к ошибкам компиляции неопределенных ошибок функции.

Модули могут иметь несколько атрибутов `@behaviour` .

Прочитайте поведения онлайн: https://riptutorial.com/ru/elixir/topic/3558/поведения

глава 27: Полиморфизм в Эликсире

Вступление

Полиморфизм - это предоставление единого интерфейса объектам разных типов. В принципе, он позволяет различным типам данных реагировать на одну и ту же функцию. Таким образом, одни и те же функции формируют разные типы данных для достижения такого же поведения. Язык Elixir имеет `protocols` для реализации полиморфизма с чистым способом.

замечания

Если вы хотите охватить все типы данных, вы можете определить реализацию для `Any` типа данных. Наконец, если у вас есть время, проверьте исходный код Enum и String.Char , которые являются хорошими примерами полиморфизма в ядре Elixir.

Examples

Полиморфизм с протоколами

Давайте реализуем базовый протокол, который преобразует температуры Кельвина и Фаренгейта в цель.

```
defmodule Kelvin do
  defstruct name: "Kelvin", symbol: "K", degree: 0
end

defmodule Fahrenheit do
  defstruct name: "Fahrenheit", symbol: "°F", degree: 0
end

defmodule Celsius do
  defstruct name: "Celsius", symbol: "°C", degree: 0
end

defprotocol Temperature do
  @doc """
  Convert Kelvin and Fahrenheit to Celsius degree
  """
  def to_celsius(degree)
end

defimpl Temperature, for: Kelvin do
  @doc """
  Deduct 273.15
  """
  def to_celsius(kelvin) do
    celsius_degree = kelvin.degree - 273.15
    %Celsius{degree: celsius_degree}
```

```
  end
end

defimpl Temperature, for: Fahrenheit do
  @doc """
  Deduct 32, then multiply by 5, then divide by 9
  """
  def to_celsius(fahrenheit) do
    celsius_degree = (fahrenheit.degree - 32) * 5 / 9
    %Celsius{degree: celsius_degree}
  end
end
```

Теперь мы внедрили наши конвертеры для типов Кельвина и Фаренгейта. Давайте сделаем некоторые преобразования:

```
iex> fahrenheit = %Fahrenheit{degree: 45}
%Fahrenheit{degree: 45, name: "Fahrenheit", symbol: "°F"}
iex> celsius = Temperature.to_celsius(fahrenheit)
%Celsius{degree: 7.22, name: "Celsius", symbol: "°C"}
iex> kelvin = %Kelvin{degree: 300}
%Kelvin{degree: 300, name: "Kelvin", symbol: "K"}
iex> celsius = Temperature.to_celsius(kelvin)
%Celsius{degree: 26.85, name: "Celsius", symbol: "°C"}
```

Попробуем преобразовать любой другой тип данных, который не имеет реализации для функции `to_celsius` :

```
iex> Temperature.to_celsius(%{degree: 12})
** (Protocol.UndefinedError) protocol Temperature not implemented for %{degree: 12}
    iex:11: Temperature.impl_for!/1
    iex:15: Temperature.to_celsius/1
```

Прочитайте Полиморфизм в Эликсире онлайн: https://riptutorial.com/ru/elixir/topic/9519/полиморфизм-в-эликсире

глава 28: Получение справки в консоли IEx

Вступление

IEx предоставляет доступ к документации Elixir. Когда Elixir установлен в вашей системе, вы можете запустить IEx, например, с `iex` команды `iex` в терминале. Затем введите команду `h` в командной строке IEx, за которой следует имя функции, добавленное ее именем модуля, например `h List.foldr`

Examples

Список модулей и функций Elixir

Чтобы получить список модулей Elixir, просто введите

```
h Elixir.[TAB]
```

Нажатие [TAB] автозаполняет имена модулей и функций. В этом случае он перечисляет все модули. Чтобы найти все функции в модуле, например, использование `List`

```
h List.[TAB]
```

Прочитайте Получение справки в консоли IEx онлайн:
https://riptutorial.com/ru/elixir/topic/10780/получение-справки-в-консоли-iex

глава 29: Поток

замечания

Потоки являются составными, ленивыми перечислениями.

Из-за их лени потоки полезны при работе с большими (или даже бесконечными) коллекциями. При объединении многих операций с `Enum` создаются промежуточные списки, а `Stream` создает рецепт вычислений, которые выполняются в более поздний момент.

Examples

Цепочка нескольких операций

`Stream` особенно полезен, когда вы хотите запускать несколько операций в коллекции. Это связано с тем, что `Stream` ленив и выполняет только одну итерацию (тогда как, например, `Enum` несколько итераций).

```
numbers = 1..100
|> Stream.map(fn(x) -> x * 2 end)
|> Stream.filter(fn(x) -> rem(x, 2) == 0 end)
|> Stream.take_every(3)
|> Enum.to_list

[2, 8, 14, 20, 26, 32, 38, 44, 50, 56, 62, 68, 74, 80, 86, 92, 98, 104, 110,
 116, 122, 128, 134, 140, 146, 152, 158, 164, 170, 176, 182, 188, 194, 200]
```

Здесь мы связали 3 операции (`map` , `filter` и `take_every`), но окончательная итерация была выполнена только после `Enum.to_list` .

Что делает `Stream` внутренне, так это то, что он ждет, пока не потребуется фактическая оценка. До этого он создает список всех функций, но после того, как оценка необходима, она проходит через коллекцию один раз, выполняя все функции для каждого элемента. Это делает его более эффективным, чем `Enum` , который в этом случае будет делать 3 итерации, например.

Прочитайте Поток онлайн: https://riptutorial.com/ru/elixir/topic/2553/поток

глава 30: Присоединиться к строкам

Examples

Использование интерполяции строк

```
iex(1)> [x, y] = ["String1", "String2"]
iex(2)> "#{x} #{y}"
# "String1 String2"
```

Использование списка IO

```
["String1", " ", "String2"] |> IO.iodata_to_binary
# "String1 String2"
```

Это даст некоторые улучшения производительности, поскольку строки не дублируются в памяти.
Альтернативный метод:

```
iex(1)> IO.puts(["String1", " ", "String2"])
# String1 String2
```

Использование Enum.join

```
Enum.join(["String1", "String2"], " ")
# "String1 String2"
```

Прочитайте Присоединиться к строкам онлайн: https://riptutorial.com/ru/elixir/topic/9202/присоединиться-к-строкам

глава 31: протоколы

замечания

Замечание о структурах

Вместо совместного использования протокола с картами, структуры требуют собственной реализации протокола.

Examples

Вступление

Протоколы позволяют полиморфизм в Эликсире. Определение протоколов с `defprotocol` :

```
defprotocol Log do
  def log(value, opts)
end
```

Внедрить протокол с `defimpl` :

```
require Logger
# User and Post are custom structs

defimpl Log, for: User do
  def log(user, _opts) do
    Logger.info "User: #{user.name}, #{user.age}"
  end
end

defimpl Log, for: Post do
  def log(user, _opts) do
    Logger.info "Post: #{post.title}, #{post.category}"
  end
end
```

С приведенными выше реализациями мы можем сделать:

```
iex> Log.log(%User{name: "Yos", age: 23})
22:53:11.604 [info]  User: Yos, 23
iex> Log.log(%Post{title: "Protocols", category: "Protocols"})
22:53:43.604 [info]  Post: Protocols, Protocols
```

Протоколы позволяют отправлять любой тип данных, если он реализует протокол. Это включает в себя несколько встроенных типов , таких как `Atom` , `BitString` , `Tuples` и другие.

Прочитайте протоколы онлайн: https://riptutorial.com/ru/elixir/topic/3487/протоколы

глава 32: Процессы

Examples

Истечение простого процесса

В следующем примере функция `greet` внутри модуля `Greeter` запускается в отдельном процессе:

```
defmodule Greeter do
    def greet do
        IO.puts "Hello programmer!"
    end
end

iex> spawn(Greeter, :greet, [])
Hello
#PID<0.122.0>
```

Здесь `#PID<0.122.0>` - это *идентификатор процесса* для порожденного процесса.

Отправка и получение сообщений

```
defmodule Processes do
    def receiver do
        receive do
            {:ok, val} ->
                IO.puts "Received Value: #{val}"
            _ ->
                IO.puts "Received something else"
        end
    end
end
```

```
iex(1)> pid = spawn(Processes, :receiver, [])
#PID<0.84.0>
iex(2)> send pid, {:ok, 10}
Received Value: 10
{:ok, 10}
```

Рекурсия и получение

Рекурсия может использоваться для приема нескольких сообщений

```
defmodule Processes do
    def receiver do
        receive do
            {:ok, val} ->
                IO.puts "Received Value: #{val}"
            _ ->
```

```
                IO.puts "Received something else"
        end
        receiver
    end
end
```

```
iex(1)> pid = spawn Processes, :receiver, []
#PID<0.95.0>
iex(2)> send pid, {:ok, 10}
Received Value: 10
{:ok, 10}
iex(3)> send pid, {:ok, 42}
{:ok, 42}
Received Value: 42
iex(4)> send pid, :random
:random
Received something else
```

Эликсир будет использовать оптимизацию рекурсии хвостового вызова, пока вызов функции является последним, что происходит в функции, как в примере.

Прочитайте Процессы онлайн: https://riptutorial.com/ru/elixir/topic/3173/процессы

глава 33: Секреты и уловки

Вступление

Elixir Дополнительные советы и рекомендации, которые экономят время во время кодирования.

Examples

Создание пользовательских символов и документирование

Каждый x sigil вызывает соответствующее определение sigil_x

Определение пользовательских символов

```
defmodule MySigils do
  #returns the downcasing string if option l is given then returns the list of downcase
letters
  def sigil_l(string,[]), do: String.Casing.downcase(string)
  def sigil_l(string,[?l]), do: String.Casing.downcase(string) |> String.graphemes

  #returns the upcasing string if option l is given then returns the list of downcase letters
  def sigil_u(string,[]), do: String.Casing.upcase(string)
  def sigil_u(string,[?l]), do: String.Casing.upcase(string) |> String.graphemes
end
```

Несколько [OR]

Это просто другой способ записи нескольких условий OR. Это не рекомендуется, потому что при регулярном подходе, когда условие оценивается как true, оно перестает выполнять оставшиеся условия, которые экономят время оценки, в отличие от этого подхода, который сначала оценивает все условия в списке. Это просто плохо, но полезно для открытий.

```
# Regular Approach
find = fn(x) when x>10 or x<5 or x==7 -> x end

# Our Hack
hell = fn(x) when true in [x>10,x<5,x==7] -> x end
```

iex Custom Configuration - iex Decoration

Скопируйте содержимое в файл и сохраните файл как .iex.exs в ~ домашнем каталоге и увидите волшебство. Вы также можете скачать файл ЗДЕСЬ

```
# IEx.configure colors: [enabled: true]
# IEx.configure colors: [ eval_result: [ :cyan, :bright ] ]
```

```
IO.puts IO.ANSI.red_background() <> IO.ANSI.white() <> "  ❄❄❄ Good Luck with Elixir ❄❄❄ " <> IO.ANSI.reset
Application.put_env(:elixir, :ansi_enabled, true)
IEx.configure(
 colors: [
   eval_result: [:green, :bright] ,
   eval_error: [[:red,:bright,"Bug Bug ..!!"]],
   eval_info: [:yellow, :bright ],
 ],
 default_prompt: [
   "\e[G",    # ANSI CHA, move cursor to column 1
    :white,
    "I",
    :red,
    "❤" ,       # plain string
    :green,
    "%prefix",:white,"|",
     :blue,
     "%counter",
     :white,
     "|",
    :red,
    "▶" ,        # plain string
    :white,
    "▶▶"  ,       # plain string
      # ❤ ❤-»" ,  # plain string
    :reset
  ] |> IO.ANSI.format |> IO.chardata_to_string

)
```

Прочитайте Секреты и уловки онлайн: https://riptutorial.com/ru/elixir/topic/10623/секреты-и-уловки

глава 34: Советы и рекомендации консоли IEx

Examples

Перекомпилировать проект с `перекомпилировать`

```
iex(1)> recompile
Compiling 1 file (.ex)
:ok
```

См. Документацию с буквой `h`

```
iex(1)> h List.last

                                def last(list)

Returns the last element in list or nil if list is empty.

Examples

| iex> List.last([])
| nil
|
| iex> List.last([1])
| 1
|
| iex> List.last([1, 2, 3])
| 3
```

Получить значение из последней команды с помощью `v`

```
iex(1)> 1 + 1
2
iex(2)> v
2
iex(3)> 1 + v
3
```

См. Также: Получить значение строки с `v`

Получить значение предыдущей команды с помощью `v`

```
iex(1)> a = 10
10
iex(2)> b = 20
20
iex(3)> a + b
```

```
30
```

Вы можете получить определенную строку, передающую индекс строки:

```
iex(4)> v(3)
30
```

Вы также можете указать индекс относительно текущей строки:

```
iex(5)> v(-1) # Retrieves value of row (5-1) -> 4
30
iex(6)> v(-5) # Retrieves value of row (5-4) -> 1
10
```

Значение может быть повторно использовано в других расчетах:

```
iex(7)> v(2) * 4
80
```

Если вы укажете несуществующую строку, `IEx` вызовет ошибку:

```
iex(7)> v(100)
** (RuntimeError) v(100) is out of bounds
    (iex) lib/iex/history.ex:121: IEx.History.nth/2
    (iex) lib/iex/helpers.ex:357: IEx.Helpers.v/1
```

Выход из консоли IEx

1. Используйте Ctrl + C, Ctrl + C для выхода

```
iex(1)>
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
```

2. Используйте `Ctrl+ \` для немедленного выхода

См. Информацию с `i`

```
iex(1)> i :ok
Term
  :ok
Data type
  Atom
Reference modules
  Atom
iex(2)> x = "mystring"
"mystring"
iex(3)> i x
Term
  "mystring"
Data type
```

```
  BitString
Byte size
  8
Description
  This is a string: a UTF-8 encoded binary. It's printed surrounded by
  "double quotes" because all UTF-8 encoded codepoints in it are printable.
Raw representation
  <<109, 121, 115, 116, 114, 105, 110, 103>>
Reference modules
  String, :binary
```

Создание PID

Это полезно, если вы не сохранили PID из предыдущей команды

```
iex(1)> self()
#PID<0.138.0>
iex(2)> pid("0.138.0")
#PID<0.138.0>
iex(3)> pid(0, 138, 0)
#PID<0.138.0>
```

Подготовьте свои псевдонимы при запуске IEx

Если вы `.iex.exs` ваши обычно используемые псевдонимы в файл `.iex.exs` в корне вашего приложения, IEx загрузит их для вас при запуске.

```
alias App.{User, Repo}
```

Постоянная история

По умолчанию история ввода пользователя в `IEx` не сохраняется в разных сеансах.

`erlang-history` добавляет поддержку истории как для оболочки Erlang, так и для `IEx` :

```
git clone git@github.com:ferd/erlang-history.git
cd erlang-history
sudo make install
```

Теперь вы можете получить доступ к своим предыдущим входам, используя клавиши со стрелками вверх и вниз, даже в разных сеансах `IEx` .

Когда консоль Elixir застряла ...

Иногда вы можете случайно запустить что-то в оболочке, которая заканчивается навсегда, и, таким образом, блокирует оболочку:

```
iex(2)> receive do _ -> :stuck end
```

В этом случае нажмите Ctrl-g. Вот увидишь:

```
User switch command
```

Введите следующие команды:

- `k` (для уничтожения процесса оболочки)
- `s` (для запуска нового процесса оболочки)
- `c` (для подключения к новому процессу оболочки)

Вы попадете в новую оболочку Erlang:

```
Eshell V8.0.2  (abort with ^G)
1>
```

Чтобы запустить оболочку Elixir, введите:

```
'Elixir.IEx.CLI':local_start().
```

(не забудьте последнюю точку!)

Затем вы увидите новый процесс оболочки Elixir:

```
Interactive Elixir (1.3.2) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> "I'm back"
"I'm back"
iex(2)>
```

Чтобы выйти из режима «ожидание для большего ввода» (из-за незамкнутой кавычки, скобки и т. Д.), **`#iex:break`** , а затем возврат каретки (▯):

```
iex(1)> "Hello, "world"
...(1)>
...(1)> #iex:break
** (TokenMissingError) iex:1: incomplete expression

iex(1)>
```

вышеописанное особенно полезно, когда копирование в относительно огромный фрагмент превращает консоль в режим ожидания для большего ввода.

вырваться из неполного выражения

Когда вы введете что-то в IEx, который ожидает завершения, например многострочную строку, IEx изменит приглашение, чтобы указать, что он ждет вас, закончив, изменив приглашение на наличие многоточия (`...`), а не `iex` .

Если вы обнаружите, что IEx ждет вас, чтобы завершить выражение, но вы не знаете, что ему нужно для завершения выражения, или вы просто хотите прервать эту строку ввода,

введите `#iex:break` в качестве входа в консоль. Это заставит IEx выкинуть `TokenMissingError` и отменить ожидание ввода большего количества, возвращая вас к стандартным `TokenMissingError` консоли верхнего уровня.

```
iex:1> "foo"
"foo"
iex:2> "bar
...:2> #iex:break
** (TokenMissingError) iex:2: incomplete expression
```

Дополнительная информация доступна в документации IEx .

Загрузите модуль или скрипт в сеанс IEx

Если у вас есть файл эликсира; сценарий или модуль и хотите загрузить его в текущий сеанс IEx, вы можете использовать метод `c/1` :

```
iex(1)> c "lib/utils.ex"
iex(2)> Utils.some_method
```

Это скомпилирует и загрузит модуль в IEx, и вы сможете вызвать все его общедоступные методы.

Для скриптов он немедленно выполнит содержимое скрипта:

```
iex(3)> c "/path/to/my/script.exs"
Called from within the script!
```

Прочитайте Советы и рекомендации консоли IEx онлайн:
https://riptutorial.com/ru/elixir/topic/1283/советы-и-рекомендации-консоли-iex

глава 35: Советы по отладке

Examples

Отладка с помощью IEX.pry / 0

Отладка с помощью `IEx.pry/0` довольно проста.

1. `require IEx` в вашем модуле
2. Найдите строку кода, которую вы хотите проверить
3. Добавьте `IEx.pry` после строки

Теперь запустите свой проект (например, `iex -S mix`).

Когда линия с `IEx.pry/0` будет достигнута, программа остановится, и вы сможете проверить ее. Это похоже на точку останова в традиционном отладчике.

Когда вы закончите, просто введите `respawn` в консоль.

```
require IEx;

defmodule Example do
  def double_sum(x, y) do
    IEx.pry
    hard_work(x, y)
  end

  defp hard_work(x, y) do
    2 * (x + y)
  end
end
```

Отладка с помощью IO.inspect / 1

В качестве инструмента для отладки программы эликсиров можно использовать IO.inspect / 1.

```
defmodule MyModule do
  def myfunction(argument_1, argument_2) do
    IO.inspect(argument_1)
    IO.inspect(argument_2)
  end
end
```

Он распечатает аргументы_1 и argument_2 на консоли. Поскольку `IO.inspect/1` возвращает свой аргумент, его очень легко включить в вызовы функций или конвейеры, не нарушая поток:

```
do_something(a, b)
|> do_something_else(c)

# can be adorned with IO.inspect, with no change in functionality:

do_something(IO.inspect(a), IO.inspect(b))
|> IO.inspect
do_something(IO.inspect(c))
```

Отладка в трубе

```
defmodule Demo do
  def foo do
    1..10
    |> Enum.map(&(&1 * &1))           |> p
    |> Enum.filter(&rem(&1, 2) == 0) |> p
    |> Enum.take(3)                   |> p
  end

  defp p(e) do
    require Logger
    Logger.debug inspect e, limit: :infinity
    e
  end
end
```

```
iex(1)> Demo.foo

23:23:55.171 [debug] [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

23:23:55.171 [debug] [4, 16, 36, 64, 100]

23:23:55.171 [debug] [4, 16, 36]

[4, 16, 36]
```

Приколите трубку

```
defmodule Demo do
  def foo do
    1..10
    |> Enum.map(&(&1 * &1))
    |> Enum.filter(&rem(&1, 2) == 0) |> pry
    |> Enum.take(3)
  end

  defp pry(e) do
    require IEx
    IEx.pry
    e
  end
end
```

```
iex(1)> Demo.foo
```

```
Request to pry #PID<0.117.0> at lib/demo.ex:11

      def pry(e) do
        require IEx
        IEx.pry
        e
      end

Allow? [Yn] Y

Interactive Elixir (1.3.2) - press Ctrl+C to exit (type h() ENTER for help)
pry(1)> e
[4, 16, 36, 64, 100]
pry(2)> respawn

Interactive Elixir (1.3.2) - press Ctrl+C to exit (type h() ENTER for help)
[4, 16, 36]
iex(1)>
```

Прочитайте Советы по отладке онлайн: https://riptutorial.com/ru/elixir/topic/2719/советы-по-отладке

глава 36: Согласование образцов

Examples

Функции сопоставления шаблонов

```
#You can use pattern matching to run different
#functions based on which parameters you pass

#This example uses pattern matching to start,
#run, and end a recursive function

defmodule Counter do
    def count_to do
        count_to(100, 0) #No argument, init with 100
    end

    def count_to(counter) do
        count_to(counter, 0) #Initialize the recursive function
    end

    def count_to(counter, value) when value == counter do
        #This guard clause allows me to check my arguments against
        #expressions. This ends the recursion when the value matches
        #the number I am counting to.
        :ok
    end

    def count_to(counter, value) do
        #Actually do the counting
        IO.puts value
        count_to(counter, value + 1)
    end
end
```

Согласование шаблонов на карте

```
%{username: username} = %{username: "John Doe", id: 1}
# username == "John Doe"
```

```
%{username: username, id: 2} = %{username: "John Doe", id: 1}
** (MatchError) no match of right hand side value: %{id: 1, username: "John Doe"}
```

Сопоставление шаблонов в списке

Вы также можете сопоставлять шаблоны в Elixir Data Structures, таких как списки.

Списки

Совпадение по списку довольно простое.

```
[head | tail] = [1,2,3,4,5]
# head == 1
# tail == [2,3,4,5]
```

Это работает, сопоставляя первые (или более) элементы в списке с левой стороны `|` (труба), а остальная часть списка - с правой стороны переменной `|` ,

Мы также можем сопоставлять определенные значения списка:

```
[1,2 | tail] = [1,2,3,4,5]
# tail = [3,4,5]

[4 | tail] = [1,2,3,4,5]
** (MatchError) no match of right hand side value: [1, 2, 3, 4, 5]
```

Связывание нескольких последовательных значений слева от `|` также разрешено:

```
[a, b | tail] = [1,2,3,4,5]
# a == 1
# b == 2
# tail = [3,4,5]
```

Еще сложнее - мы можем сопоставлять определенное значение и сопоставлять его с переменной:

```
iex(11)> [a = 1 | tail] = [1,2,3,4,5]
# a == 1
```

Получите сумму списка с использованием соответствия шаблону

```
defmodule Math do
  # We start of by passing the sum/1 function a list of numbers.
  def sum(numbers) do
    do_sum(numbers, 0)
  end

  # Recurse over the list when it contains at least one element.
  # We break the list up into two parts:
  #   head: the first element of the list
  #   tail: a list of all elements except the head
  # Every time this function is executed it makes the list of numbers
  # one element smaller until it is empty.
  defp do_sum([head|tail], acc) do
    do_sum(tail, head + acc)
  end

 # When we have reached the end of the list, return the accumulated sum
  defp do_sum([], acc), do: acc
end
```

Анонимные функции

```
f = fn
  {:a, :b} -> IO.puts "Tuple {:a, :b}"
  [] -> IO.puts "Empty list"
end

f.({:a, :b}) # Tuple {:a, :b}
f.([])       # Empty list
```

Кортеж

```
{ a, b, c } = { "Hello", "World", "!" }

IO.puts a # Hello
IO.puts b # World
IO.puts c # !

# Tuples of different size won't match:

{ a, b, c } = { "Hello", "World" } # (MatchError) no match of right hand side value: {
"Hello", "World" }
```

Чтение файла

Согласование шаблонов полезно для операции, такой как чтение файла, которое возвращает кортеж.

Если файл `sample.txt` содержит `This is a sample text` , тогда:

```
{ :ok, file } = File.read("sample.txt")
# => {:ok, "This is a sample text"}

file
# => "This is a sample text"
```

В противном случае, если файл не существует:

```
{ :ok, file } = File.read("sample.txt")
# => ** (MatchError) no match of right hand side value: {:error, :enoent}

{ :error, msg } = File.read("sample.txt")
# => {:error, :enoent}
```

Учет соответствия анонимных функций

```
fizzbuzz = fn
  (0, 0, _) -> "FizzBuzz"
  (0, _, _) -> "Fizz"
  (_, 0, _) -> "Buzz"
  (_, _, x) -> x
end

my_function = fn(n) ->
```

```
  fizzbuzz.(rem(n, 3), rem(n, 5), n)
end
```

Прочитайте Согласование образцов онлайн: https://riptutorial.com/ru/elixir/topic/1602/согласование-образцов

глава 37: Списки

Синтаксис

- []
- [1, 2, 3, 4]
- [1, 2] ++ [3, 4] # -> [1,2,3,4]
- hd ([1, 2, 3, 4]) # -> 1
- tl ([1, 2, 3, 4]) # -> [2,3,4]
- [глава | хвост]
- [1 | [2, 3, 4]] # -> [1,2,3,4]
- [1 | [2 | [3 | [4 | []]]]] -> [1,2,3,4]
- 'hello' = [? h,? e,? l,? l,? o]
- keyword_list = [a: 123, b: 456, c: 789]
- keyword_list [: a] # -> 123

Examples

Списки ключевых слов

Списки ключевых слов представляют собой списки, в которых каждый элемент в списке является кортежем атома, за которым следует значение.

```
keyword_list = [{:a, 123}, {:b, 456}, {:c, 789}]
```

Сокращенное обозначение для написания списков ключевых слов выглядит следующим образом:

```
keyword_list = [a: 123, b: 456, c: 789]
```

Списки ключевых слов полезны для создания упорядоченных структур данных пары «ключ-значение», где для данного ключа могут существовать несколько элементов.

Первый элемент в списке ключевых слов для заданного ключа может быть получен следующим образом:

```
iex> keyword_list[:b]
456
```

Вариант использования для списков ключевых слов может быть последовательностью запущенных именованных задач:

```
defmodule TaskRunner do
```

```
  def run_tasks(tasks) do
    # Call a function for each item in the keyword list.
    # Use pattern matching on each {:key, value} tuple in the keyword list
    Enum.each(tasks, fn
      {:delete, x} ->
        IO.puts("Deleting record " <> to_string(x) <> "...")
      {:add, value} ->
        IO.puts("Adding record \"" <> value <> "\"...")
      {:update, {x, value}} ->
        IO.puts("Setting record " <> to_string(x) <> " to \"" <> value <> "\"...")
    end)
  end
end
```

Этот код можно вызвать с таким списком ключевых слов:

```
iex> tasks = [
...>   add: "foo",
...>   add: "bar",
...>   add: "test",
...>   delete: 2,
...>   update: {1, "asdf"}
...> ]

iex> TaskRunner.run_tasks(tasks)
Adding record "foo"...
Adding record "bar"...
Adding record "test"...
Deleting record 2...
Setting record 1 to "asdf"...
```

Списки Char

Строки в Elixir - это «двоичные файлы». Однако в коде Erlang строки традиционно являются «списками символов», поэтому при вызове функций Erlang вам, возможно, придется использовать списки символов вместо обычных строк Elixir.

В то время как обычные строки записываются с помощью двойных кавычек `"` , символьные списки записываются с помощью одиночных кавычек `'` :

```
string = "Hello!"
char_list = 'Hello!'
```

Списки Char - это просто списки целых чисел, представляющие кодовые точки каждого символа.

```
'hello' = [104, 101, 108, 108, 111]
```

Строка может быть преобразована в список символов с помощью `to_charlist/1` :

```
iex> to_charlist("hello")
'hello'
```

И обратное можно сделать с помощью `to_string/1` :

```
iex> to_string('hello')
"hello"
```

Вызов функции Erlang и преобразование вывода в обычную строку Elixir:

```
iex> :os.getenv |> hd |> to_string
"PATH=/usr/local/bin:/usr/bin:/bin"
```

Минусы

Списки в Elixir являются связанными списками. Это означает, что каждый элемент в списке состоит из значения, за которым следует указатель на следующий элемент в списке. Это реализовано в Elixir с использованием cons-ячеек.

Минусы - это простые структуры данных с «левым» и «правильным» значением, или «голова» и «хвост».

А `|` символ может быть добавлен до последнего элемента в списке, чтобы обозначить (неправильный) список с заданной головой и хвостом. Ниже приведена единственная ячейка cons с `1` в качестве головы и `2` как хвост:

```
[1 | 2]
```

Стандартный синтаксис Elixir для списка на самом деле эквивалентен написанию цепочки вложенных cons-ячеек:

```
[1, 2, 3, 4] = [1 | [2 | [3 | [4 | []]]]]
```

Пустой список `[]` используется как хвост ячейки cons для представления конца списка.

Все списки в Elixir эквивалентны форме `[head | tail]` , где `head` - это первый элемент списка, а `tail` - остальная часть списка, минус голова.

```
iex> [head | tail] = [1, 2, 3, 4]
[1, 2, 3, 4]
iex> head
1
iex> tail
[2, 3, 4]
```

Использование `[head | tail]` полезно для сопоставления образцов в рекурсивных функциях:

```
def sum([]), do: 0
```

```
def sum([head | tail]) do
  head + sum(tail)
end
```

Списки сопоставлений

`map` - функция в функциональном программировании, которая задает список и функцию, возвращает новый список с функцией, применяемой к каждому элементу в этом списке. В Elixir функция `map/2` находится в модуле Enum .

```
iex> Enum.map([1, 2, 3, 4], fn(x) -> x + 1 end)
[2, 3, 4, 5]
```

Использование альтернативного синтаксиса захвата для анонимных функций:

```
iex> Enum.map([1, 2, 3, 4], &(&1 + 1))
[2, 3, 4, 5]
```

Ссылаясь на функцию с синтаксисом захвата:

```
iex> Enum.map([1, 2, 3, 4], &to_string/1)
["1", "2", "3", "4"]
```

Перемещение операций с использованием оператора трубы:

```
iex> [1, 2, 3, 4]
...> |> Enum.map(&to_string/1)
...> |> Enum.map(&("Chapter " <> &1))
["Chapter 1", "Chapter 2", "Chapter 3", "Chapter 4"]
```

Список рекомендаций

Эликсир не имеет петель. Вместо них для списков есть большие модули `Enum` и `List` , но есть также List Consrehensions.

Список Пояснения могут быть полезны для:

- создавать новые списки

```
iex(1)> for value <- [1, 2, 3], do: value + 1
[2, 3, 4]
```

- фильтрация списков, используя `guard` выражения , но использовать их без `when` это ключевое слово.

```
iex(2)> odd? = fn x -> rem(x, 2) == 1 end
iex(3)> for value <- [1, 2, 3], odd?.(value), do: value
[1, 3]
```

- создать пользовательскую карту, используя `into` ключевого слова:

```
iex(4)> for value <- [1, 2, 3], into: %{}, do: {value, value + 1}
%{1 => 2, 2=>3, 3 => 4}
```

Комбинированный пример

```
iex(5)> for value <- [1, 2, 3], odd?.(value), into: %{}, do: {value, value * value}
%{1 => 1, 3 => 9}
```

Резюме

Список рекомендаций:

- использует синтаксис `for..do` с дополнительными `for..do` после запятых и `into` ключевое слово при возврате другой структуры, чем списки, т. е. карта.
- в других случаях возвращают новые списки
- не поддерживает аккумуляторы
- не может прекратить обработку при выполнении определенного условия
- `guard` заявления должны быть первыми, чтобы после того, как `for` и перед `do` или `into` символы. Порядок символов не имеет значения

В соответствии с этими ограничениями List Consrehensions ограничены только для простого использования. В более сложных случаях использование функций из модулей `Enum` и `List` было бы лучшей идеей.

Переменная списка

```
iex> [1, 2, 3] -- [1, 3]
[2]
```

`--` удаляет первое вхождение элемента в левом списке для каждого элемента справа.

Список участников

Используйте `in` операторе, чтобы проверить, является ли элемент членом списка.

```
iex> 2 in [1, 2, 3]
true
iex> "bob" in [1, 2, 3]
false
```

Преобразование списков в карту

Используйте `Enum.chunk/2` для группировки элементов в под-списки и `Map.new/2` для преобразования его в карту:

```
[1, 2, 3, 4, 5, 6]
|> Enum.chunk(2)
|> Map.new(fn [k, v] -> {k, v} end)
```

Даст:

```
%{1 => 2, 3 => 4, 5 => 6}
```

Прочитайте Списки онлайн: https://riptutorial.com/ru/elixir/topic/1279/списки

глава 38: Структуры данных

Синтаксис

- [глава | tail] = [1, 2, 3, true] # можно использовать совпадение шаблонов для разбивки cons-ячеек. Это назначает головке 1 и хвосту [2, 3, true]
- % {d: val} =% {d: 1, e: true} # это присваивает val 1; никакая переменная d не создается, потому что d на lhs на самом деле является просто символом, который используется для создания шаблона% {: d => _} (обратите внимание, что обозначение ракеты хешей позволяет иметь несимволы в качестве ключей для карт, подобных в рубине)

замечания

Что касается того, какая структура данных для нас здесь представляет собой несколько кратких замечаний.

Если вам нужна структура данных массива, если вы собираетесь много писать списки использования. Если вместо этого вы будете много читать, вы должны использовать кортежи.

Что касается карт, это просто то, как вы делаете хранилища ключей.

Examples

Списки

```
a = [1, 2, 3, true]
```

Обратите внимание, что они хранятся в памяти в виде связанных списков. Id est это серия cons-ячеек, где head (List.hd / 1) является значением первого элемента списка, а хвост (List.tail / 1) является значением остальной части списка.

```
List.hd(a) = 1
List.tl(a) = [2, 3, true]
```

Кортеж

```
b = {:ok, 1, 2}
```

Кортежи являются эквивалентом массивов на других языках. Они хранятся смежно в

памяти.

Прочитайте Структуры данных онлайн: https://riptutorial.com/ru/elixir/topic/1607/структуры-данных

глава 39: Струны

замечания

`String` в Elixir является `UTF-8` .

Examples

Преобразовать в строку

Используйте Kernel.inspect для преобразования чего-либо в строку.

```
iex> Kernel.inspect(1)
"1"
iex> Kernel.inspect(4.2)
"4.2"
iex> Kernel.inspect %{pi: 3.14, name: "Yos"}
"%{pi: 3.14, name: \"Yos\"}"
```

Получить подстроку

```
iex> my_string = "Lorem ipsum dolor sit amet, consectetur adipiscing elit."
iex> String.slice my_string, 6..10
"ipsum"
```

Разделить строку

```
iex> String.split("Elixir, Antidote, Panacea", ",")
["Elixir", "Antidote", "Panacea"]
```

Интерполяция строк

```
iex(1)> name = "John"
"John"
iex(2)> greeting = "Hello, #{name}"
"Hello, John"
iex(3)> num = 15
15
iex(4)> results = "#{num} item(s) found."
"15 item(s) found."
```

Проверьте, содержит ли строка String

```
iex(1)> String.contains? "elixir of life", "of"
true
iex(2)> String.contains? "elixir of life", ["life", "death"]
```

```
true
iex(3)> String.contains? "elixir of life", ["venus", "mercury"]
false
```

Присоединиться к строкам

Вы можете объединить строки в Elixir с помощью оператора `<>` :

```
"Hello" <> "World"                          # => "HelloWorld"
```

Для `List` строк вы можете использовать `Enum.join/2` :

```
Enum.join(["A", "few", "words"], " ")       # => "A few words"
```

Прочитайте Струны онлайн: https://riptutorial.com/ru/elixir/topic/2618/струны

глава 40: функции

Examples

Анонимные функции

В Elixir распространенной практикой является использование анонимных функций. Создание анонимной функции прост:

```
iex(1)> my_func = fn x -> x * 2 end
#Function<6.52032458/1 in :erl_eval.expr/5>
```

Общий синтаксис:

```
fn args -> output end
```

Для удобства чтения вы можете поместить круглые скобки вокруг аргументов:

```
iex(2)> my_func = fn (x, y) -> x*y end
#Function<12.52032458/2 in :erl_eval.expr/5>
```

Чтобы вызвать анонимную функцию, вызовите ее по назначенному имени и добавьте `.` между именем и аргументами.

```
iex(3)>my_func.(7, 5)
35
```

Анонимные функции можно объявлять без аргументов:

```
iex(4)> my_func2 = fn -> IO.puts "hello there" end
iex(5)> my_func2.()
hello there
:ok
```

Использование оператора захвата

Чтобы сделать анонимные функции более краткими, вы можете использовать **оператор захвата** `&` . Например, вместо:

```
iex(5)> my_func = fn (x) -> x*x*x end
```

Ты можешь написать:

```
iex(6)> my_func = &(&1*&1*&1)
```

С помощью нескольких параметров используйте число, соответствующее каждому аргументу, считая от `1` :

```
iex(7)> my_func = fn (x, y) -> x + y end

iex(8)> my_func = &(&1 + &2)    # &1 stands for x and &2 stands for y

iex(9)> my_func.(4, 5)
9
```

Несколько тел

Анонимная функция также может иметь несколько тел (в результате сопоставления с образцом):

```
my_func = fn
  param1 -> do_this
  param2 -> do_that
end
```

Когда вы вызываете функцию с несколькими телами, Elixir пытается сопоставить параметры, которые вы предоставили с правильным телом функции.

Списки ключевых слов как функциональные параметры

Используйте списки ключевых слов для параметров параметров «options», которые содержат несколько пар ключ-значение:

```
def myfunc(arg1, opts \\ []) do
  # Function body
end
```

Мы можем вызвать функцию выше:

```
iex> myfunc "hello", pizza: true, soda: false
```

что эквивалентно:

```
iex> myfunc("hello", [pizza: true, soda: false])
```

Значения аргументов доступны как `opts.pizza` и `opts.soda` соответственно.
В качестве альтернативы вы можете использовать атомы: `opts[:pizza]` и `opts[:soda]` .

Именованные функции и частные функции

Именованные функции

```
defmodule Math do
    # one way
    def add(a, b) do
        a + b
    end

    # another way
    def subtract(a, b), do: a - b
end

iex> Math.add(2, 3)
5
:ok
iex> Math.subtract(5, 2)
3
:ok
```

Частные функции

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

    # Private Function
    defp add(a, b) do
        a + b
    end
end

iex> Math.add(2, 3)
** (UndefinedFunctionError) undefined function Math.add/2
Math.add(3, 4)
iex> Math.sum(2, 3)
5
```

Соответствие шаблону

Эликсир сопоставляет вызов функции своему телу на основе значения его аргументов.

```
defmodule Math do
    def factorial(0): do: 1
    def factorial(n): do: n * factorial(n - 1)
end
```

Здесь факториал положительных чисел соответствует второму предложению, а `factorial(0)` соответствует первому. (игнорируя отрицательные числа для простоты). Эликсир пытается сопоставить функции сверху донизу. Если вторая функция написана над первой, мы получим неожиданный результат, так как это приведет к бесконечной рекурсии. Поскольку `factorial(0)` соответствует `factorial(n)`

Оговорки о защите

Оговорки Guard позволяют нам проверять аргументы перед выполнением функции. Оговорки о защите обычно предпочтительнее, `if` и `cond` из-за их удобочитаемости, и для того, чтобы упростить компилятор для определенной методики оптимизации . Первое определение функции, в котором выполняется все защитные меры.

Ниже приведен пример реализации факториальной функции с использованием защиты и сопоставления шаблонов.

```
defmodule Math do
    def factorial(0), do: 1
    def factorial(n) when n > 0: do: n * factorial(n - 1)
end
```

Первый шаблон соответствует if (и только если) аргумент равен `0` . Если аргумент не равен `0` , совпадение шаблона не выполняется, и проверяется следующая функция ниже.

Это второе определение функции имеет предложение охраны: `when n > 0` . Это означает, что эта функция соответствует только аргументу `n` больше `0` . В конце концов, математическая факторная функция не определена для отрицательных целых чисел.

Если ни одно определение функции (включая их соответствие шаблону и предложения охраны) не будет `FunctionClauseError` будет `FunctionClauseError` . Это происходит для этой функции, когда мы передаем отрицательное число в качестве аргумента, так как оно не определено для отрицательных чисел.

Обратите внимание, что этот сам `FunctionClauseError` не является ошибкой. Возвращение `-1` или `0` или другое «значение ошибки», как это принято на некоторых других языках, скроет тот факт, что вы вызывали неопределенную функцию, скрывая источник ошибки, возможно, создавая огромную болезненную ошибку для будущего разработчика.

Параметры по умолчанию

Вы можете передавать параметры по умолчанию любой именованной функции с помощью синтаксиса: `param \\ value` :

```
defmodule Example do
    def func(p1, p2 \\ 2) do
        IO.inspect [p1, p2]
    end
end

Example.func("a")     # => ["a", 2]
Example.func("b", 4) # => ["b", 4]
```

Функции захвата

Используйте `&` для захвата функций из других модулей. Вы можете использовать захваченные функции непосредственно как функциональные параметры или в анонимных функциях.

```
Enum.map(list, fn(x) -> String.capitalize(x) end)
```

Можно сделать более сжатым использование `&` :

```
Enum.map(list, &String.capitalize(&1))
```

Захват функций без передачи каких-либо аргументов требует явного указания его арности, например `&String.capitalize/1` :

```
defmodule Bob do
  def say(message, f \\ &String.capitalize/1) do
    f.(message)
  end
end
```

Прочитайте функции онлайн: https://riptutorial.com/ru/elixir/topic/2442/функции

глава 41: Функциональное программирование в Elixir

Вступление

Попробуем реализовать основные функции более высоких заказов, такие как карта и уменьшить с помощью Elixir

Examples

карта

Карта - это функция, которая будет принимать массив и функцию и возвращать массив после применения этой функции к ***каждому элементу*** в этом списке

```
defmodule MyList do
  def map([], _func) do
    []
  end

  def map([head | tail], func) do
    [func.(head) | map(tail, func)]
  end
end
```

Скопируйте пасту в `iex` и выполните:

```
MyList.map [1,2,3], fn a -> a * 5 end
```

Синтаксис Shorthand - `MyList.map [1,2,3], &(&1 * 5)`

уменьшить

Сокращение - это функция, которая будет принимать массив, функцию и аккумулятор и использовать ***накопитель в качестве семени для запуска итерации с первым элементом, чтобы дать следующий аккумулятор, и итерация продолжается для всех элементов в массиве*** (см. Ниже пример)

```
defmodule MyList do
  def reduce([], _func, acc) do
    acc
  end

  def reduce([head | tail], func, acc) do
    reduce(tail, func, func.(acc, head))
  end
```

```
end
```

Скопируйте вставку вышеприведенного фрагмента в iex:

1. Чтобы добавить все числа в массив: `MyList.reduce [1,2,3,4], fn acc, element -> acc + element end, 0`
2. Чтобы скорректировать все числа в массиве: `MyList.reduce [1,2,3,4], fn acc, element -> acc * element end, 1`

Объяснение, например, 1:

```
Iteration 1 => acc = 0, element = 1 ==> 0 + 1 ===> 1 = next accumulator
Iteration 2 => acc = 1, element = 2 ==> 1 + 2 ===> 3 = next accumulator
Iteration 3 => acc = 3, element = 3 ==> 3 + 3 ===> 6 = next accumulator
Iteration 4 => acc = 6, element = 4 ==> 6 + 4 ===> 10 = next accumulator = result(as all
elements are done)
```

Отфильтруйте список, используя

```
MyList.reduce [1,2,3,4], fn acc, element -> if rem(element,2) == 0 do acc else acc ++
[element] end end, []
```

Прочитайте Функциональное программирование в Elixir онлайн:
https://riptutorial.com/ru/elixir/topic/10186/функциональное-программирование-в-elixir

кредиты

S. No	Главы	Contributors
1	Начало работы с Elixir Language	alejosocorro, Andrey Chernykh, Ben Bals, Community, cwc, Delameko, Douglas Correa, helcim, I Am Batman, JAlberto, koolkat, leifg, MattW., rap-2-h, Simone Carletti, Stephan Rodemeier, Vinicius Quaiato, Yedhu Krishnan, Zimm i48
2	Conditionals	Andrey Chernykh, evuez, javanut13, Musfiqur Rahman, Paweł Obrok
3	Doctests	aholt, milmazz, Philippe-Arnaud de MANGOU, Yos Riady
4	Ecto	fgutierr, Philippe-Arnaud de MANGOU, toraritte
5	Erlang	4444, Yos Riady
6	ExDoc	milmazz, Yos Riady
7	ExUnit	Yos Riady
8	Sigils	javanut13, Yos Riady
9	Вершины	Yos Riady
10	Встроенные типы	Andrey Chernykh, Arithmeticbird, Oskar, TreyE, Vinicius Quaiato
11	Государственная обработка в эликсире	Paweł Obrok
12	задача	mario
13	Карты и списки ключевых слов	Sam Mercier, Simone Carletti, Yos Riady
14	Константы	ibgib
15	ЛУЧ	Yos Riady
16	Лучшая отладка с помощью IO.inspect и ярлыков	leifg
17	Метапрограммирование	4444, Paweł Obrok

18	микшировать	4444, helcim, rainteller, Slava.K, Yos Riady
19	Модули	Alex G, javanut13, Yos Riady
20	Монтаж	cwc, Douglas Correa, Eiji, JAlberto, MattW.
21	операторы	alxndr, Andrey Chernykh, Dair, Gazler, Mitkins, nirev, PatNowak
22	оптимизация	Filip Haglund, legoscia
23	основное использование охранных оговорок	alxndr
24	Основной .gitignore для программы elixir	Yos Riady
25	поведения	Yos Riady
26	Полиморфизм в Эликсире	mustafaturan
27	Получение справки в консоли IEx	helcim
28	Поток	Oskar
29	Присоединиться к строкам	Agung Santoso
30	протоколы	Yos Riady
31	Процессы	Alex G, Yedhu Krishnan
32	Секреты и уловки	Ankanna
33	Советы и рекомендации консоли IEx	alxndr, Cifer, fahrradflucht, legoscia, mudasobwa, muttonlamb, PatNowak, Paweł Obrok, sbs, Sheharyar, Simone Carletti, Stephan Rodemeier, Uniaika, Vincent, Yos Riady
34	Советы по отладке	javanut13, Paweł Obrok, Pfitz, Philippe-Arnaud de MANGOU, sbs
35	Согласование образцов	Alex Anderson, Dair, Danny Rosenblatt, evuez, Gabriel C, gmile, Harrison Lucas, javanut13, Oskar, PatNowak, theIV, Thomas, Yedhu Krishnan

36	Списки	Ben Bals, Candy Gumdrop, emoragaf, PatNowak, Sheharyar, Yos Riady
37	Структуры данных	Sam Mercier, Simone Carletti, Stephan Rodemeier, Yos Riady
38	Струны	Alex G, Sheharyar, Yos Riady
39	функции	Andrey Chernykh, cwc, Dair, Eiji, Filip Haglund, PatNowak, rainteller, Simone Carletti, Stephan Rodemeier, Yedhu Krishnan, Yos Riady
40	Функциональное программирование в Elixir	Dinesh Balasubramanian