EBook Gratis

APRENDIZAJE
Elixir Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#elixir

Tabla de contenido

Acerca de

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

Capítulo 1: Empezando con Elixir Language

Observaciones

Elixir es un lenguaje dinámico y funcional diseñado para crear aplicaciones escalables y mantenibles.

Elixir aprovecha la máquina virtual de Erlang, conocida por ejecutar sistemas de baja latencia, distribuidos y tolerantes a fallas, mientras que también se utiliza con éxito en el desarrollo web y el dominio de software integrado.

Versiones

Versión	Fecha de lanzamiento
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

Hola Mundo

Para obtener instrucciones de instalación en cheque elixir aquí , que describe las instrucciones relacionadas con diferentes plataformas.

Elixir es un lenguaje de programación que se crea usando `erlang` y usa el tiempo de ejecución `BEAM` de erlang (como `JVM` para Java).

Podemos usar elixir en dos modos: shell interactivo `iex` o ejecutar directamente usando el comando `elixir` .

Coloque lo siguiente en un archivo llamado `hello.exs` :

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

Desde la línea de comandos, escriba el siguiente comando para ejecutar el archivo fuente de

Elixir:

```
$ elixir hello.exs
```

Esto debería dar salida:

    ¡Hola Mundo!

Esto se conoce como el *modo* de `Elixir` de `Elixir` . De hecho, los programas de Elixir también se pueden compilar (y, en general, están) en un código de bytes para la máquina virtual BEAM.

También puede usar `iex` para el shell de elixir interactivo (recomendado), ejecute el comando y obtendrá un mensaje como este:

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

Aquí puedes probar tus ejemplos de elixir `hello world` :

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

También puede compilar y ejecutar sus módulos a través de `iex` . Por ejemplo, si tiene un `helloworld.ex` que contiene:

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

A través de `iex` , haz:

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

Hola mundo desde IEx

También puede usar el `IEx` (Interactive Elixir) para evaluar expresiones y ejecutar código.

Si está en Linux o Mac, simplemente escriba iex en su base y presione enter:

```
$ iex
```

Si está en una máquina con Windows, escriba:

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

Luego ingresará en el IEx REPL (Leer, Evaluar, Imprimir, Bucle), y puede escribir algo como:

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

Si desea cargar un script mientras abre un IEx REPL, puede hacer esto:

```
$ iex script.exs
```

Dado `script.exs` es su script. Ahora puede llamar a funciones desde el script en la consola.

Lea Empezando con Elixir Language en línea:
https://riptutorial.com/es/elixir/topic/954/empezando-con-elixir-language

Capítulo 2: Comportamientos

Examples

Introducción

Los comportamientos son una lista de especificaciones de funciones que otro módulo puede implementar. Son similares a las interfaces en otros idiomas.

Aquí hay un ejemplo de comportamiento:

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

Y un módulo que lo implementa:

```
defmodule JSONParser do
  @behaviour Parser

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

El `@behaviour` módulo `@behaviour` anterior indica que se espera que este módulo defina cada función definida en el módulo de análisis. Las funciones que faltan darán como resultado errores de compilación de la función de comportamiento indefinido.

Los módulos pueden tener múltiples atributos de `@behaviour` .

Lea Comportamientos en línea: https://riptutorial.com/es/elixir/topic/3558/comportamientos

Capítulo 3: Condicionales

Observaciones

Tenga en cuenta que la sintaxis `do...end` es azúcar sintáctico para las listas de palabras clave normales, por lo que puede hacer esto:

```
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

caso

```
 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` solo se utiliza para coincidir con el patrón dado de los datos particulares. Aquí, `{1,2}` se corresponde con un patrón de caso diferente que se da en el ejemplo de código.

si ya menos que

```
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
```

cond

```
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` generará un `CondClauseError` si no se cumplen las condiciones.

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

Esto se puede evitar agregando una condición que siempre será verdadera.

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

A menos que nunca se espere que llegue al caso predeterminado, y el programa debería fallar en ese punto.

con cláusula

`with` cláusula se utiliza para combinar cláusulas coincidentes. Parece que combinamos funciones anónimas o manejamos funciones con varios cuerpos (cláusulas coincidentes). Considere el caso: creamos un usuario, lo insertamos en la base de datos, luego creamos un correo electrónico de bienvenida y luego lo enviamos al usuario.

Sin la cláusula `with` podríamos escribir algo como esto (omití implementaciones de funciones):

```
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
```

Aquí manejamos el flujo de nuestro proceso de negocio con el `case` (podría ser `cond` o `if`). Eso nos lleva a la llamada 'pirámide de la fatalidad' , porque tenemos que lidiar con las posibles condiciones y decidir: seguir avanzando o no. Sería mucho mejor reescribir este código con `with`

declaración:

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

En el fragmento de código anterior, hemos reescrito las cláusulas de `case` anidados `with` . Dentro de `with` invocamos algunas funciones (ya sea anónimas o con nombre) y la coincidencia de patrones en sus salidas. Si todo coincidentes, `with` retorno de `do` resultado del bloque, o de `else` resultado del bloque de otra manera.

Podemos omitir `else` manera `with` devolverá o bien `do` resultado del bloque o el resultado primero fallar.

Por lo tanto, el valor de `with` sentencia es su resultado de bloque `do` .

Lea Condicionales en línea: https://riptutorial.com/es/elixir/topic/2118/condicionales

Capítulo 4: Consejos de depuración

Examples

Depuración con IEX.pry / 0

La depuración con `IEx.pry/0` es bastante simple.

1. `require IEx` en su módulo
2. Encuentra la línea de código que quieres inspeccionar
3. Añadir `IEx.pry` después de la línea

Ahora inicie su proyecto (por ejemplo, `iex -S mix`).

Cuando se `IEx.pry/0` la línea con `IEx.pry/0` el programa se detendrá y usted tendrá la oportunidad de inspeccionar. Es como un punto de quiebre en un depurador tradicional.

Cuando hayas terminado solo escribe `respawn` en la consola.

```
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
```

Depuración con IO.inspect / 1

Es posible usar IO.inspect / 1 como una herramienta para depurar un programa de elixir.

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

Imprimirá argumento_1 y argumento_2 en la consola. Dado que `IO.inspect/1` devuelve su argumento, es muy fácil incluirlo en llamadas a funciones o tuberías sin interrumpir el flujo:

```
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))
```

Depuración en la tubería

```
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]
```

Hacer palanca en el tubo

```
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)>
```

Lea Consejos de depuración en línea: https://riptutorial.com/es/elixir/topic/2719/consejos-de-depuracion

Capítulo 5: Consejos y trucos

Introducción

Elixir Avanzado consejos y trucos que ahorran nuestro tiempo durante la codificación.

Examples

Creando sigilos personalizados y documentando

Cada x sigil llama a la respectiva definición de sigil_x

Definiendo sigilos personalizados

```
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
```

Múltiples [o]

Esta es solo la otra forma de escribir las condiciones OR múltiples. Este no es el enfoque recomendado porque en el enfoque regular cuando la condición se evalúa como verdadera, deja de ejecutar las condiciones restantes que ahorran el tiempo de evaluación, a diferencia de este enfoque que evalúa todas las condiciones primero en la lista. Esto es solo malo pero bueno para descubrimientos.

```
# 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
```

Configuración personalizada de iex - Decoración de iex

Copie el contenido en un archivo y guárdelo como .iex.exs en su directorio principal y vea la magia. También puedes descargar el archivo AQUÍ.

```
# 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

)
```

Lea Consejos y trucos en línea: https://riptutorial.com/es/elixir/topic/10623/consejos-y-trucos

Capítulo 6: Consejos y trucos de la consola IEx

Examples

Recompila proyecto con `recompile`

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

Ver documentación con `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
```

Obtener valor del último comando con `v`

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

Vea también: Obtenga el valor de una fila con `v`

Obtenga el valor de un comando anterior con `v`

```
iex(1)> a = 10
10
iex(2)> b = 20
20
iex(3)> a + b
30
```

Puede obtener una fila específica pasando el índice de la fila:

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

También puede especificar un índice relativo a la fila actual:

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

El valor puede ser reutilizado en otros cálculos:

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

Si especifica una fila no existente, `IEx` generará un error:

```
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
```

Salir de la consola IEx

1. Use Ctrl + C, Ctrl + C para salir

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

2. Use `Ctrl+ \` para salir inmediatamente

Ver información con `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
```

Creación de PID

Esto es útil cuando no guardó el PID de un comando anterior

```
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>
```

Tenga sus alias listos cuando inicie IEx

Si coloca sus alias de uso común en un archivo `.iex.exs` en la raíz de su aplicación, IEx los cargará por usted al inicio.

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

Historia persistente

De forma predeterminada, el historial de entrada del usuario en `IEx` no se mantiene en las diferentes sesiones.

`erlang-history` agrega soporte de historial tanto para el shell Erlang como para `IEx` :

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

Ahora puede acceder a sus entradas anteriores con las teclas de flecha hacia arriba y hacia abajo, incluso a través de diferentes sesiones `IEx` .

Cuando la consola de Elixir está atascada ...

A veces, es posible que ejecutes accidentalmente algo en el shell que termina esperando para siempre y, por lo tanto, bloquea el shell:

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

En ese caso, presione Ctrl-g. Verás:

```
User switch command
```

Ingrese estos comandos en orden:

- `k` (para matar el proceso de shell)
- `s` (para iniciar un nuevo proceso de shell)
- `c` (para conectarse al nuevo proceso de shell)

Terminarás en una nueva concha de Erlang:

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

Para iniciar un shell Elixir, escriba:

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

(No olvides el punto final!)

Luego verás un nuevo proceso de shell 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)>
```

Para escapar del modo "en espera de más entrada" (debido a las comillas no cerradas, corchetes, etc.) escriba **`#iex:break`** , seguido de un retorno de carro (␤):

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

iex(1)>
```

lo anterior es especialmente útil cuando copiar y pegar un fragmento relativamente grande convierte la consola en el modo de "espera de más entrada".

salir de la expresión incompleta

Cuando haya ingresado algo en IEx que espera una finalización, como una cadena multilínea, IEx cambiará la solicitud para indicar que está esperando a que termine cambiando la solicitud para que tenga puntos suspensivos (`...`) en lugar de `iex` .

Si encuentra que IEx está esperando que termine una expresión pero no está seguro de lo que necesita para terminar la expresión, o simplemente desea cancelar esta línea de entrada, ingrese `#iex:break` como entrada de la consola. Esto hará que IEx lance un `TokenMissingError` y cancele la espera de más entrada, devolviéndole a una entrada de consola estándar de "nivel superior".

```
iex:1> "foo"
"foo"
```

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

Más información está disponible en la documentación de IEx .

Cargue un módulo o script en la sesión IEx

Si tienes un archivo de elixir; un script o un módulo y desea cargarlo en la sesión IEx actual, puede usar el método `c/1` :

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

Esto compilará y cargará el módulo en IEx, y podrás llamar a todos sus métodos públicos.

Para los scripts, ejecutará inmediatamente el contenido del script:

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

Lea Consejos y trucos de la consola IEx en línea:
https://riptutorial.com/es/elixir/topic/1283/consejos-y-trucos-de-la-consola-iex

Capítulo 7: Constantes

Observaciones

Así que este es un análisis de resumen que he hecho en base a los métodos listados en ¿Cómo se definen las constantes en los módulos de Elixir? . Lo estoy publicando por un par de razones:

- La mayoría de la documentación de Elixir es bastante completa, pero encontré que esta decisión arquitectónica clave carece de orientación, por lo que la habría solicitado como tema.
- Quería obtener un poco de visibilidad y comentarios de otros sobre el tema.
- También quería probar el nuevo flujo de trabajo de Documentación SO. ;)

También he cargado el código completo en el concepto de elixir-constantes de GitHub.

Examples

Constantes de ámbito de módulo

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

Estos solo son accesibles desde este módulo.

Constantes como funciones

Declarar:

```
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
```

Consumir con requerir:

```
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
```

Consumir con importación:

```
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
```

Este método permite reutilizar las constantes en los proyectos, pero no se podrán utilizar dentro de las funciones de guarda que requieren constantes de tiempo de compilación.

Constantes a través de macros

Declarar:

```
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
```

Consumir con `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
```

Este método le permite usar `@some_constant` dentro de guardias. Ni siquiera estoy seguro de que las funciones sean estrictamente necesarias.

Lea Constantes en línea: https://riptutorial.com/es/elixir/topic/6614/constantes

Capítulo 8: Corriente

Observaciones

Las transmisiones son compostables, enumerables perezosos.

Debido a su pereza, las secuencias son útiles cuando se trabaja con colecciones grandes (o incluso infinitas). Al encadenar muchas operaciones con `Enum` , se crean listas intermedias, mientras que `Stream` crea una receta de cálculos que se ejecutan en un momento posterior.

Examples

Encadenamiento de múltiples operaciones.

`Stream` es especialmente útil cuando desea ejecutar varias operaciones en una colección. Esto se debe a que `Stream` es perezoso y solo hace una iteración (mientras que `Enum` haría varias iteraciones, por ejemplo).

```
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]
```

Aquí, encadenamos 3 operaciones (`map` , `filter` y `take_every`), pero la iteración final solo se realizó después de `Enum.to_list` .

Lo que `Stream` hace internamente, es que espera hasta que se requiera una evaluación real. Antes de eso, crea una lista de todas las funciones, pero una vez que se necesita una evaluación, recorre la colección una vez, ejecutando todas las funciones en cada elemento. Esto lo hace más eficiente que `Enum` , que en este caso haría 3 iteraciones, por ejemplo.

Lea Corriente en línea: https://riptutorial.com/es/elixir/topic/2553/corriente

Capítulo 9: Doctests

Examples

Introducción

Cuando documenta su código con `@doc` , puede proporcionar ejemplos de código así:

```
# 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
```

Puede agregar los ejemplos de código como casos de prueba en una de sus suites de prueba:

```
# myproject/test/doc_test.exs

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

Luego, puede ejecutar sus pruebas con la `mix test` .

Generando documentación HTML basada en doctest.

Debido a que la generación de documentación se basa en la reducción, tiene que hacer 2 cosas:

1 / Escriba su doctest y haga que sus doctest sean claros para mejorar la legibilidad (es mejor dar un título, como "ejemplos" o "pruebas"). Cuando escriba sus pruebas, no olvide dar 4 espacios a su código de prueba para que se formatee como código en la documentación HTML.

2 / Luego, ingrese "mix docs" en la consola en la raíz de su proyecto de elixir para generar la documentación HTML en el directorio doc ubicado en la raíz de su proyecto de elixir.

```
$> mix docs
```

Doctests multilínea

Puedes hacer un doctest multilínea usando '...>' para las líneas que siguen a la primera

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

Lea Doctests en línea: https://riptutorial.com/es/elixir/topic/2708/doctests

Capítulo 10: Ecto

Examples

Añadiendo un Ecto.Repo en un programa de elixir

Esto se puede hacer en 3 pasos:

1. Debe definir un módulo de elixir que use Ecto.Repo y registre su aplicación como un otp_app.

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

2. También debe definir alguna configuración para el Repo que le permitirá conectarse a la base de datos. Aquí hay un ejemplo con 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. Antes de utilizar Ecto en su aplicación, debe asegurarse de que Ecto se inicie antes de que se inicie su aplicación. Se puede hacer con el registro de Ecto en lib / custom_app.ex como supervisor.

```
    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
```

"y" cláusula en un Repo.get_by / 3

Si tiene un Ecto.Queryable, llamado Post, que tiene un título y una descripción.

Puede obtener la publicación con el título: "hola" y la descripción: "mundo" ejecutando:

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

Todo esto es posible porque Repo.get_by espera en el segundo argumento una Lista de palabras clave.

Consulta con campos dinámicos.

Para consultar un campo cuyo nombre está contenido en una variable, use la función de campo .

```
some_field = :id
some_value = 10

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

Agregue tipos de datos personalizados a la migración y al esquema

(De esta respuesta)

El siguiente ejemplo agrega un tipo enumerado a una base de datos de postgres.

Primero, edite el **archivo de migración** (creado con la `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
```

Segundo, en el **archivo modelo,** agregue un campo con un tipo de Elixir:

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

o implementar el comportamiento `Ecto.Type` .

Un buen ejemplo para este último es el paquete `ecto_enum` y se puede usar como plantilla. Su uso está bien documentado en su **página github** .

Este compromiso muestra un ejemplo de uso en un proyecto de Phoenix al agregar enum_ecto al proyecto y usar el tipo enumerado en vistas y modelos.

Lea Ecto en línea: https://riptutorial.com/es/elixir/topic/6524/ecto

Capítulo 11: El polimorfismo en el elixir

Introducción

El polimorfismo es la provisión de una interfaz única para entidades de diferentes tipos. Básicamente, permite que diferentes tipos de datos respondan a la misma función. Entonces, la misma función da forma a diferentes tipos de datos para lograr el mismo comportamiento. El lenguaje elixir tiene `protocols` para implementar el polimorfismo de una manera limpia.

Observaciones

Si desea cubrir todos los tipos de datos, puede definir una implementación para `Any` tipo de datos. Por último, si tiene tiempo, verifique el código fuente de Enum y String.Char , que son buenos ejemplos de polimorfismo en el núcleo Elixir.

Examples

Polimorfismo con Protocolos

Implementemos un protocolo básico que convierta las temperaturas de Kelvin y Fahrenheit a Celsius.

```
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
```

Ahora, implementamos nuestros convertidores para los tipos Kelvin y Fahrenheit. Hagamos algunas conversiones:

```
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"}
```

Intentemos convertir cualquier otro tipo de datos que no tenga implementación para la función `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
```

Lea El polimorfismo en el elixir en línea: https://riptutorial.com/es/elixir/topic/9519/el-polimorfismo-en-el-elixir

Capítulo 12: Erlang

Examples

Usando Erlang

Los módulos de Erlang están disponibles como átomos. Por ejemplo, el módulo matemático de Erlang está disponible como `:math` :

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

Inspeccionar un módulo de Erlang

Utilice `module_info` en los módulos de Erlang que desea inspeccionar:

```
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>>]
```

Lea Erlang en línea: https://riptutorial.com/es/elixir/topic/2716/erlang

Capítulo 13: Estructuras de datos

Sintaxis

- [cabeza | cola] = [1, 2, 3, verdadero] # uno puede usar la coincidencia de patrones para dividir las celdas de contras. Esto asigna cabeza a 1 y cola a [2, 3, verdadero]
- % {d: val} =% {d: 1, e: verdadero} # esto asigna val a 1; no se crea una variable d porque la d en lhs es en realidad solo un símbolo que se usa para crear el patrón% {: d => _} (tenga en cuenta que la notación de cohete hash permite que no haya símbolos como claves para mapas como en rubí)

Observaciones

En cuanto a qué estructura de datos para nosotros aquí hay algunos comentarios breves.

Si necesita una estructura de datos de matriz, si va a estar escribiendo muchas listas de uso. Si, por el contrario, va a leer mucho, debería usar tuplas.

En cuanto a los mapas, son simplemente la forma en que usted hace las tiendas de valor clave.

Examples

Liza

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

Tenga en cuenta que estos se almacenan en la memoria como listas vinculadas. Esta es una serie de celdas de contras donde la cabecera (List.hd / 1) es el valor del primer elemento de la lista y la cola (List.tail / 1) es el valor del resto de la lista.

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

Tuplas

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

Las tuplas son el equivalente de matrices en otros idiomas. Se almacenan de forma contigua en la memoria.

Lea Estructuras de datos en línea: https://riptutorial.com/es/elixir/topic/1607/estructuras-de-datos

Capítulo 14: Exdoc

Examples

Introducción

Para generar la documentación en `HTML` formato de `@doc` y `@moduledoc` atributos en el código fuente, añadir `ex_doc` y un procesador de reducción del precio, en este momento ExDoc apoya Earmark , Pandoc , Hoedown y Cmark , como dependencias en su `mix.exs` archivo:

```
# config/mix.exs

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

Si desea utilizar otro procesador Markdown, puede encontrar más información en la sección Cambiar la herramienta Markdown .

Puede utilizar Markdown dentro de los `@doc` Elixir `@doc` y `@moduledoc` .

Luego, ejecute `mix docs` .

Una cosa a tener en cuenta es que ExDoc permite parámetros de configuración, tales como:

```
  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
```

Puede ver más información sobre estas opciones de configuración con `mix help docs`

Lea Exdoc en línea: https://riptutorial.com/es/elixir/topic/3582/exdoc

Capítulo 15: ExUnidad

Examples

Afirmación de excepciones

Use `assert_raise` para probar si se generó una excepción. `assert_raise` toma una excepción y una función para ser ejecutada.

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

Envuelva el código que desee probar en una función anónima y páselo a `assert_raise` .

Lea ExUnidad en línea: https://riptutorial.com/es/elixir/topic/3583/exunidad

Capítulo 16: Funciones

Examples

Funciones anónimas

En Elixir, una práctica común es usar funciones anónimas. Crear una función anónima es simple:

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

La sintaxis general es:

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

Para facilitar la lectura, puede poner paréntesis alrededor de los argumentos:

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

Para invocar una función anónima, llámela por el nombre asignado y agregue `.` entre el nombre y los argumentos.

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

Es posible declarar funciones anónimas sin argumentos:

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

Usando el operador de captura

Para hacer que las funciones anónimas sean más concisas, puede utilizar el **operador de captura** `&` . Por ejemplo, en lugar de:

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

Puedes escribir:

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

Con múltiples parámetros, use el número correspondiente a cada argumento, contando desde `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
```

Cuerpos múltiples

Una función anónima también puede tener varios cuerpos (como resultado de la coincidencia de patrones):

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

Cuando llama a una función con varios cuerpos, Elixir intenta hacer coincidir los parámetros que ha proporcionado con el cuerpo de función adecuado.

Listas de palabras clave como parámetros de función

Utilice listas de palabras clave para los parámetros de estilo 'opciones' que contienen varios pares clave-valor:

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

Podemos llamar a la función anterior así:

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

que es equivalente a:

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

Los valores de argumento están disponibles como `opts.pizza` y `opts.soda` respectivamente. Alternativamente, puedes usar los átomos: `opts[:pizza]` y `opts[:soda]` .

Funciones con nombre y funciones privadas

Funciones nombradas

```
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
```

Funciones privadas

```
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
```

La coincidencia de patrones

Elixir hace coincidir una llamada de función con su cuerpo en función del valor de sus argumentos.

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

Aquí, el factorial de los números positivos coincide con la segunda cláusula, mientras que el `factorial(0)` coincide con la primera. (ignorando los números negativos por simplicidad). Elixir intenta hacer coincidir las funciones de arriba a abajo. Si la segunda función se escribe arriba de la primera, obtendremos un resultado inesperado ya que pasa a una recursión sin fin. Porque `factorial(0)` coincide con `factorial(n)`

Cláusulas de guardia

Las cláusulas de guardia nos permiten verificar los argumentos antes de ejecutar la función. Cláusulas de guardia son generalmente preferidos para `if` y `cond` debido a su facilidad de lectura, y para hacer una cierta técnica de optimización más fácil para el compilador. Se ejecuta la

primera definición de función donde coinciden todos los guardias.

Aquí hay un ejemplo de implementación de la función factorial utilizando guardas y la coincidencia de patrones.

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

El primer patrón coincide si (y solo si) el argumento es `0` . Si el argumento no es `0` , la coincidencia del patrón falla y se comprueba la siguiente función a continuación.

Esa segunda definición de función tiene una cláusula de protección: `when n > 0` . Esto significa que esta función solo coincide si el argumento `n` es mayor que `0` . Después de todo, la función factorial matemática no está definida para enteros negativos.

Si ninguna de las definiciones de funciones (incluidas sus cláusulas de coincidencia de patrones y guardas) coinciden, se generará un `FunctionClauseError` . Esto sucede para esta función cuando pasamos un número negativo como argumento, ya que no está definido para números negativos.

Tenga en cuenta que este `FunctionClauseError` sí, no es un error. Si devuelve `-1` o `0` o algún otro "valor de error", como es común en otros idiomas, ocultará el hecho de que llamó a una función indefinida, ocultando la fuente del error, posiblemente creando un error muy doloroso para un futuro desarrollador.

Parámetros predeterminados

Puede pasar los parámetros predeterminados a cualquier función nombrada usando la sintaxis: `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]
```

Funciones de captura

Utilice `&` para capturar funciones de otros módulos. Puede utilizar las funciones capturadas directamente como parámetros de función o dentro de funciones anónimas.

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

Puede hacerse más conciso usando `&` :

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

Las funciones de captura sin pasar ningún argumento requieren que especifique explícitamente su aridad, por ejemplo, `&String.capitalize/1` :

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

Lea Funciones en línea: https://riptutorial.com/es/elixir/topic/2442/funciones

Capítulo 17: HAZ

Examples

Introducción

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

`:observer.start` abre la interfaz del observador de la GUI, que muestra la falla de la CPU, el uso de la memoria y otra información crítica para comprender los patrones de uso de sus aplicaciones.

Lea HAZ en línea: https://riptutorial.com/es/elixir/topic/3587/haz

Capítulo 18: Instalación

Examples

Instalación de Fedora

```
dnf install erlang elixir
```

Instalacion OSX

En OS X y MacOS, Elixir se puede instalar a través de los administradores de paquetes comunes:

Homebrew

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

Macports

```
 $ sudo port install elixir
```

Instalación de 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
```

Instalación Gentoo / Funtoo

Elixir está disponible en el repositorio de paquetes principales.
Actualice la lista de paquetes antes de instalar cualquier paquete:

```
emerge --sync
```

Este es un paso de instalación:

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

Lea Instalación en línea: https://riptutorial.com/es/elixir/topic/4208/instalacion

Capítulo 19: Instrumentos de cuerda

Observaciones

Una `String` en Elixir es un binario codificado en `UTF-8` .

Examples

Convertir a cadena

Usa Kernel.inspect para convertir cualquier cosa en una cadena.

```
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\"}"
```

Obtener una subcadena

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

Dividir una cadena

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

Interpolación de cuerdas

```
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."
```

Compruebe si la cadena contiene subcadena

```
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
```

Unir cuerdas

Puede concatenar cadenas en Elixir utilizando el operador `<>` :

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

Para una `List` de cadenas, puede utilizar `Enum.join/2` :

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

Lea Instrumentos de cuerda en línea: https://riptutorial.com/es/elixir/topic/2618/instrumentos-de-cuerda

Capítulo 20: La coincidencia de patrones

Examples

Funciones de coincidencia de patrones

```
#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
```

Patrón de coincidencia en un mapa

```
%{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"}
```

Coincidencia de patrones en una lista

También puede hacer un patrón de coincidencia en las estructuras de datos de Elixir, como las listas.

Liza

Coincidencia en una lista es bastante simple.

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

Esto funciona al hacer coincidir los primeros (o más) elementos de la lista con el lado izquierdo de la `|` (tubería) y el resto de la lista a la derecha de la variable `|` .

También podemos coincidir en valores específicos de una lista:

```
[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]
```

Enlace de múltiples valores consecutivos a la izquierda de la `|` También está permitido:

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

Más complejo aún: podemos hacer coincidir un valor específico y compararlo con una variable:

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

Obtén la suma de una lista usando la coincidencia de patrones

```
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
```

Funciones anonimas

```
f = fn
  {:a, :b} -> IO.puts "Tuple {:a, :b}"
```

```
  [] -> IO.puts "Empty list"
end

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

Tuplas

```
{ 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" }
```

Leyendo un archivo

La coincidencia de patrones es útil para una operación como la lectura de archivos que devuelve una tupla.

Si el archivo `sample.txt` contiene `This is a sample text` , entonces:

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

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

De lo contrario, si el archivo no existe:

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

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

Funciones anónimas que coinciden con el patrón

```
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
```

Lea La coincidencia de patrones en línea: https://riptutorial.com/es/elixir/topic/1602/la-coincidencia-de-patrones

Capítulo 21: Liza

Sintaxis

- []
- [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]
- [cabeza | cola]
- [1 | [2, 3, 4]] # -> [1,2,3,4]
- [1 | [2 | [3 | [4 | []]]]] -> [1,2,3,4]
- 'hola' = [? h,? e,? l,? l,? o]
- keyword_list = [a: 123, b: 456, c: 789]
- keyword_list [: a] # -> 123

Examples

Listas de palabras clave

Las listas de palabras clave son listas donde cada elemento de la lista es una tupla de un átomo seguida de un valor.

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

Una notación abreviada para escribir listas de palabras clave es la siguiente:

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

Las listas de palabras clave son útiles para crear estructuras de datos ordenadas de pares clave-valor, donde pueden existir múltiples elementos para una clave determinada.

El primer elemento en una lista de palabras clave para una clave dada se puede obtener así:

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

Un caso de uso para listas de palabras clave podría ser una secuencia de tareas nombradas para ejecutar:

```
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
```

Se puede llamar a este código con una lista de palabras clave así:

```
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"...
```

Listas de Char

Las cuerdas en el elixir son "binarios". Sin embargo, en el código de Erlang, las cadenas son tradicionalmente "listas de caracteres", por lo que al llamar a las funciones de Erlang, es posible que tenga que usar listas de caracteres en lugar de cadenas de Elixir normales.

Mientras que las cadenas normales se escriben con comillas dobles `"` , las listas de caracteres se escriben con comillas simples `'` :

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

Las listas de caracteres son simplemente listas de números enteros que representan los puntos de código de cada carácter.

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

Una cadena se puede convertir en una lista de caracteres con `to_charlist/1` :

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

Y lo contrario se puede hacer con `to_string/1` :

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

Llamar a una función de Erlang y convertir la salida a una cadena de Elixir regular:

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

Contras células

Las listas en Elixir son listas enlazadas. Esto significa que cada elemento de una lista consta de un valor, seguido de un puntero al siguiente elemento de la lista. Esto se implementa en el elixir utilizando células contras.

Las celdas en contra son estructuras de datos simples con un valor "izquierdo" y un valor "correcto", o una "cabeza" y una "cola".

Un `|` Se puede agregar un símbolo antes del último elemento de una lista para anotar una lista (impropia) con una cabeza y cola dadas. Lo siguiente es una única celda de contras con `1` como la cabeza y `2` como la cola:

```
[1 | 2]
```

La sintaxis estándar de Elixir para una lista es en realidad equivalente a escribir una cadena de celdas de contras anidadas:

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

La lista vacía `[]` se usa como la cola de una celda de contras para representar el final de una lista.

Todas las listas en Elixir son equivalentes a la forma `[head | tail]` , donde `head` es el primer elemento de la lista y `tail` es el resto de la lista, menos la cabeza.

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

Usando la `[head | tail]` notación de `[head | tail]` es útil para la coincidencia de patrones en funciones recursivas:

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

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

Listas de mapas

`map` es una función en la programación funcional que, dada una lista y una función, devuelve una nueva lista con la función aplicada a cada elemento de esa lista. En Elixir, la función `map/2` está en el módulo Enum .

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

Usando la sintaxis de captura alternativa para funciones anónimas:

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

Haciendo referencia a una función con sintaxis de captura:

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

Encadenamiento de operaciones de lista utilizando el operador de tubería:

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

Lista de Comprensiones

Elixir no tiene bucles. En lugar de ellos, para listas, hay excelentes módulos `Enum` y `List` , pero también hay Comprensiones de listas.

List Comprehensions puede ser útil para:

- crear nuevas listas

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

- filtrando listas, usando expresiones de `guard` pero las usas sin `when` palabra clave.

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

- crear mapa personalizado, usando `into` palabra clave:

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

Ejemplo combinado

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

Resumen

Lista de Comprensiones:

- se usa `for..do` sintaxis de ... `for..do` con guardias adicionales después de comas y `into` palabras clave cuando se devuelve otra estructura que las listas, es decir. mapa.
- En otros casos devuelven nuevas listas.
- no soporta acumuladores
- no se puede detener el procesamiento cuando se cumple cierta condición
- `guard` declaraciones de `guard` deben ser las primeras en orden después `for` antes y antes de `do` o `into` símbolos. El orden de los símbolos no importa.

De acuerdo con estas restricciones, las Comprensiones de lista están limitadas solo para uso simple. En casos más avanzados, usar las funciones de los módulos `Enum` y `List` sería la mejor idea.

Diferencia de lista

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

`--` elimina la primera aparición de un elemento en la lista izquierda para cada elemento a la derecha.

Lista de miembros

Use `in` operador para verificar si un elemento es miembro de una lista.

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

Convertir listas a un mapa

Use `Enum.chunk/2` para agrupar elementos en sub-listas, y `Map.new/2` para convertirlo en un Mapa:

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

Daría:

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

Lea Liza en línea: https://riptutorial.com/es/elixir/topic/1279/liza

Capítulo 22: Los operadores

Examples

El operador de tubería

El operador de tubería `|>` toma el resultado de una expresión de la izquierda y la alimenta como primer parámetro a una función de la derecha.

```
expression |> function
```

Utilice el operador de tubería para encadenar expresiones y documentar visualmente el flujo de una serie de funciones.

Considera lo siguiente:

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

En el ejemplo, `Oven.bake` aparece antes que `Ingredients.mix` , pero se ejecuta en último lugar. Además, puede que no sea obvio que `:temperature` es un parámetro de `Oven.bake`

Reescribiendo este ejemplo usando el operador de tubería:

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

Da el mismo resultado, pero el orden de ejecución es más claro. Además, está claro que `:temperature` es un parámetro para la llamada `Oven.bake` .

Tenga en cuenta que cuando se usa el operador de tuberías, el primer parámetro para cada función se reubica antes que el operador de tuberías, por lo que la función a la que se llama parece tener un parámetro menos. Por ejemplo:

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

es lo mismo que:

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

Operador de tuberías y paréntesis.

Se necesitan paréntesis para evitar la ambigüedad:

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

Debe escribirse como:

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

operadores booleanos

Hay dos tipos de operadores booleanos en Elixir:

- operadores booleanos (esperan que sea `true` o `false` como su primer argumento)

```
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
```

Todos los operadores booleanos generarán `ArgumentError` si el primer argumento no es estrictamente un valor booleano, lo que significa que solo es `true` o `false` (`nil` no es booleano).

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

- Operadores booleanos relajados (trabajo con cualquier tipo, todo lo que ni `false` ni `nil` se considera `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
```

Operador `||` siempre devolverá el primer argumento si es veraz (Elixir trata todo excepto `nil` y `false` para que sea cierto en las comparaciones), de lo contrario, devolverá el segundo.

```
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
```

Operator `&&` siempre devolverá el segundo argumento si es veraz. De lo contrario, volverá respectivamente a los argumentos, `false` o `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
```

Ambos `&&` y `||` Son operadores de cortocircuito. Solo ejecutan el lado derecho si el lado izquierdo no es suficiente para determinar el resultado.

Operador `!` devolverá el valor booleano de negación del término actual:

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

La forma simple de obtener el valor booleano del término seleccionado es simplemente duplicar este operador:

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

Operadores de comparación

Igualdad:

- igualdad de valor `x == y` (`1 == 1.0 # true`)
- desigualdad de valor `x == y` (`1 != 1.0 # false`)
- igualdad estricta `x === y` (`1 === 1.0 # false`)
- desigualdad estricta `x === y` (`1 !== 1.0 # true`)

Comparación:

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

Si los tipos son compatibles, la comparación usa orden natural. De lo contrario hay regla general de comparación de tipos:

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

Unirse a los operadores

Puede unir (concatenar) binarios (incluidas cadenas) y listas:

```
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"
```

Operador 'In'

`in` operador le permite verificar si una lista o un rango incluye un elemento:

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

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

Lea Los operadores en línea: https://riptutorial.com/es/elixir/topic/1161/los-operadores

Capítulo 23: Manejo de estado en elixir

Examples

Manejar un estado con un agente

La forma más sencilla de envolver y acceder a un estado es `Agent` . El módulo permite generar un proceso que mantiene una estructura de datos arbitraria y permite enviar mensajes para leer y actualizar esa estructura. Gracias a esto, el acceso a la estructura se serializa automáticamente, ya que el proceso solo maneja un mensaje a la vez.

```
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}
```

Lea Manejo de estado en elixir en línea: https://riptutorial.com/es/elixir/topic/6596/manejo-de-estado-en-elixir

Capítulo 24: Mapas y listas de palabras clave

Sintaxis

- map =% {} // crea un mapa vacío
- map =% {: a => 1,: b => 2} // crea un mapa no vacío
- list = [] // crea una lista vacía
- list = [{: a, 1}, {: b, 2}] // crea una lista de palabras clave no vacía

Observaciones

Elixir proporciona dos estructuras de datos asociativos: *mapas* y *listas de palabras clave* .

Los mapas son el tipo clave-valor Elixir (también llamado diccionario o hash en otros idiomas).

Las listas de palabras clave son tuplas de clave / valor que asocian un valor a una clave determinada. Generalmente se utilizan como opciones para una llamada de función.

Examples

Creando un Mapa

Los mapas son el tipo clave-valor Elixir (también llamado diccionario o hash en otros idiomas). Usted crea un mapa usando la sintaxis `%w{}` :

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

Las claves y valores pueden ser de cualquier tipo:

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

Además, puedes tener mapas con tipos mixtos tanto para claves como para valores ":

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

Cuando todas las claves de un mapa son átomos, puede usar la sintaxis de las palabras clave para su comodidad:

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

Creación de una lista de palabras clave

Las listas de palabras clave son tuplas de clave / valor, generalmente utilizadas como opciones para una llamada de función.

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

Las listas de palabras clave pueden tener la misma clave repetida más de una vez.

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

Las claves y valores pueden ser de cualquier tipo:

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

Diferencia entre mapas y listas de palabras clave

Los mapas y las listas de palabras clave tienen diferentes aplicaciones. Por ejemplo, un mapa no puede tener dos claves con el mismo valor y no está ordenado. Por el contrario, una lista de palabras clave puede ser un poco difícil de usar en la coincidencia de patrones en algunos casos.

Aquí hay algunos casos de uso para mapas y listas de palabras clave.

Use listas de palabras clave cuando:

- Necesitas los elementos a ordenar.
- Necesitas más de un elemento con la misma llave.

Usa mapas cuando:

- quieres un patrón de coincidencia con algunas claves / valores
- No necesitas más de un elemento con la misma llave.
- siempre que no necesite explícitamente una lista de palabras clave

Lea Mapas y listas de palabras clave en línea: https://riptutorial.com/es/elixir/topic/2706/mapas-y-listas-de-palabras-clave

Capítulo 25: Mejor depuración con IO. Inspección y etiquetas.

Introducción

`IO.inspect` es muy útil cuando intenta depurar sus cadenas de llamadas a métodos. Se puede ensuciar si lo usas demasiado a menudo.

Desde Elixir 1.4.0, la opción de `label` de `IO.inspect` puede ayudar

Observaciones

Solo funciona con Elixir 1.4+, pero no puedo etiquetar eso todavía.

Examples

Sin etiquetas

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

Esto dará como resultado una gran cantidad de resultados sin contexto:

```
"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}
```

Con etiquetas

Usar la opción de `label` para agregar contexto puede ayudar mucho:

```
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}
```

Lea Mejor depuración con IO. Inspección y etiquetas. en línea:
https://riptutorial.com/es/elixir/topic/8725/mejor-depuracion-con-io--inspeccion-y-etiquetas-

Capítulo 26: Mejoramiento

Examples

¡Siempre mide primero!

Estos son consejos generales que en general mejoran el rendimiento. Si su código es lento, siempre es importante perfilarlo para averiguar qué partes son lentas. Adivinar **nunca** es suficiente. Mejorar la velocidad de ejecución de algo que solo ocupa el 1% del tiempo de ejecución probablemente no valga la pena. Busque los grandes sumideros de tiempo.

Para obtener números un tanto precisos, asegúrese de que el código que está optimizando se ejecute durante al menos un segundo al perfilar. Si gasta el 10% del tiempo de ejecución en esa función, asegúrese de que la ejecución completa del programa tome al menos 10 segundos y asegúrese de que puede ejecutar los mismos datos exactos a través del código varias veces, para obtener números repetibles.

ExProf es simple para comenzar.

Lea Mejoramiento en línea: https://riptutorial.com/es/elixir/topic/6062/mejoramiento

Capítulo 27: Metaprogramacion

Examples

Generar pruebas en tiempo de compilación.

```
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
```

Salida:

```
.

  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
```

Lea Metaprogramacion en línea: https://riptutorial.com/es/elixir/topic/4069/metaprogramacion

Capítulo 28: Mezcla

Examples

Crear una tarea de mezcla personalizada

```
# 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
```

Compilar y ejecutar:

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

Tarea de mezcla personalizada con argumentos de línea de comando

En una implementación básica, el módulo de tareas debe definir una función `run/1` que tome una lista de argumentos. Ej. `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
```

Compilar y ejecutar:

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

Alias

Elixir le permite agregar alias para sus comandos de mezcla. Algo genial si quieres ahorrarte algo de mecanografía.

Abra `mix.exs` en su proyecto Elixir.

Primero, agregue la función `aliases/0` a la lista de palabras clave que devuelve la función del

`project` . *Agregar* `()` *al final de la función de alias evitará que el compilador emita una advertencia.*

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

Luego, defina su función `aliases/0` (por ejemplo, en la parte inferior de su archivo `mix.exs`).

```
  ...

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

Ahora puede usar `$ mix go` para ejecutar su servidor Phoenix (si está ejecutando una aplicación Phoenix). Y use `$ mix trident` para indicar a la mezcla que capte todas las dependencias, compile y ejecute el servidor.

Obtenga ayuda sobre las tareas de mezcla disponibles

Para listar las tareas de mezcla disponibles use:

```
mix help
```

Para obtener ayuda sobre una tarea específica, use la `mix help task` por ejemplo:

```
mix help cmd
```

Lea Mezcla en línea: https://riptutorial.com/es/elixir/topic/3585/mezcla

Capítulo 29: Módulos

Observaciones

Nombres de módulos

En Elixir, los nombres de módulos como `IO` o `String` son solo átomos debajo del capó y se convierten a la forma `:"Elixir.ModuleName"` en el momento de la compilación.

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

Examples

Listar las funciones o macros de un módulo.

La función `__info__/1` toma uno de los siguientes átomos:

- `:functions` - Devuelve una lista de palabras clave de funciones públicas junto con sus aridades
- `:macros` - Devuelve una lista de palabras clave de macros públicas junto con sus aridades

Para enumerar las funciones del módulo `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, ...]
```

Reemplace el `Kernel` con cualquier módulo de su elección.

Utilizando modulos

Los módulos tienen cuatro palabras clave asociadas para usarlos en otros módulos: `alias` , `import` , `use` y `require` .

`alias` registrará un módulo con un nombre diferente (generalmente más corto):

```
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` hará que todas las funciones en el módulo estén disponibles sin nombre delante de ellas:

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

`use` permite a un módulo inyectar código en el módulo actual; esto generalmente se realiza como parte de un marco que crea sus propias funciones para que su módulo confirme algún comportamiento.

`require` cargar macros desde el módulo para que puedan ser utilizadas.

Delegar funciones a otro módulo.

Utilice `defdelegate` para definir funciones que delegan funciones del mismo nombre definidas en otro módulo:

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


iex> Math.pi
3.141592653589793
```

Lea Módulos en línea: https://riptutorial.com/es/elixir/topic/2721/modulos

Capítulo 30: Nodos

Examples

Listar todos los nodos visibles en el sistema

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

Conexión de nodos en la misma máquina.

Inicie dos nodos con nombre en dos ventanas de terminal:

```
>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)>
```

Conecte dos nodos indicando a un nodo que se conecte:

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

Los dos nodos ahora están conectados y son conscientes uno del otro:

```
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"]
```

Puede ejecutar código en otros nodos:

```
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
```

Conexión de nodos en diferentes máquinas.

Iniciar un proceso con nombre en una dirección IP:

```
$ 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"]
```

Comience otro proceso con nombre en una dirección IP diferente:

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

Lea Nodos en línea: https://riptutorial.com/es/elixir/topic/2065/nodos

Capítulo 31: Obteniendo ayuda en la consola IEx

Introducción

IEx proporciona acceso a la documentación de Elixir. Cuando Elixir está instalado en su sistema, puede iniciar IEx, por ejemplo, con el comando `iex` en un terminal. Luego escriba el comando `h` en la línea de comandos IEx seguido del nombre de la función precedido por el nombre de su módulo, por ejemplo, `h List.foldr`

Examples

Listado de módulos y funciones de Elixir

Para obtener la lista de módulos de Elixir simplemente escriba

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

Al presionar [TAB] se autocompletan los módulos y nombres de funciones. En este caso enumera todos los módulos. Para encontrar todas las funciones en un módulo, por ejemplo, uso de `List`

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

Lea Obteniendo ayuda en la consola IEx en línea:
https://riptutorial.com/es/elixir/topic/10780/obteniendo-ayuda-en-la-consola-iex

Capítulo 32: Procesos

Examples

Generando un proceso simple

En el siguiente ejemplo, la función de `greet` dentro del módulo `Greeter` se ejecuta en un proceso separado:

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

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

Aquí `#PID<0.122.0>` es el *identificador de proceso* para el proceso generado.

Enviando y recibiendo mensajes

```
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}
```

Recursion y Recibir

La recursión puede usarse para recibir múltiples mensajes.

```
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
```

Elixir utilizará una optimización de recursión de llamada de cola siempre y cuando la llamada de función sea lo último que suceda en la función como en el ejemplo.

Lea Procesos en línea: https://riptutorial.com/es/elixir/topic/3173/procesos

Capítulo 33: Programa básico .gitignore para elixir

Lea Programa básico .gitignore para elixir en línea:
https://riptutorial.com/es/elixir/topic/6493/programa-basico--gitignore-para-elixir

Capítulo 34: Programa básico .gitignore para elixir

Observaciones

Tenga en cuenta que la carpeta `/rel` puede no ser necesaria en su archivo .gitignore. Esto se genera si está utilizando una herramienta de administración de versiones como `exrm`

Examples

Un .gitignore básico para 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
```

Ejemplo

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

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

Aplicación de elixir independiente

```
/_build
/cover
```

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

Solicitud de Phoenix

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

Auto-generado .gitignore

De forma predeterminada, la `mix new <projectname>` generará un archivo `.gitignore` en la raíz del proyecto que es adecuado para 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
```

Lea Programa básico .gitignore para elixir en línea:
https://riptutorial.com/es/elixir/topic/6526/programa-basico--gitignore-para-elixir

Capítulo 35: Programación funcional en el elixir.

Introducción

Intentemos implementar las funciones básicas de órdenes superiores como mapear y reducir usando Elixir

Examples

Mapa

El mapa es una función que tomará una matriz y una función y devolverá una matriz después de aplicar esa función a ***cada elemento*** de esa lista

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

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

Copia pegar en `iex` y ejecuta:

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

La sintaxis `MyList.map [1,2,3], &(&1 * 5)` es `MyList.map [1,2,3], &(&1 * 5)`

Reducir

Reducir es una función que tomará una matriz, función y acumulador y usará el ***acumulador como semilla para iniciar la iteración con el primer elemento para dar el siguiente acumulador y la iteración continúa para todos los elementos de la matriz*** (consulte el ejemplo a continuación)

```
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
```

Copie y pegue el fragmento anterior en iex:

1. Para agregar todos los números en una matriz: `MyList.reduce [1,2,3,4], fn acc, element -> acc + element end, 0`
2. Para mutliply todos los números en una matriz: `MyList.reduce [1,2,3,4], fn acc, element -> acc * element end, 1`

Explicación para el ejemplo 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)
```

Filtra la lista usando reducir

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

Lea Programación funcional en el elixir. en línea:
https://riptutorial.com/es/elixir/topic/10186/programacion-funcional-en-el-elixir-

Capítulo 36: Protocolos

Observaciones

Una nota sobre las estructuras.

En lugar de compartir la implementación del protocolo con mapas, las estructuras requieren su propia implementación del protocolo.

Examples

Introducción

Los protocolos permiten el polimorfismo en el elixir. Definir protocolos con `defprotocol` :

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

Implementar un protocolo con `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
```

Con las implementaciones anteriores, podemos hacer:

```
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
```

Los protocolos le permiten enviar a cualquier tipo de datos, siempre que implemente el protocolo. Esto incluye algunos tipos incorporadas tales como `Atom` , `BitString` , `Tuples` , y otros.

Lea Protocolos en línea: https://riptutorial.com/es/elixir/topic/3487/protocolos

Capítulo 37: Sigilos

Examples

Construir una lista de cadenas

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

Construye una lista de átomos

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

Sigilos personalizados

Se pueden hacer sigilos personalizados creando un método `sigil_X` donde X es la letra que desea usar (esto solo puede ser una sola letra).

```
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
```

El argumento de `options` es un binario de los argumentos dados al final del sigilo, por ejemplo:

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

Lea Sigilos en línea: https://riptutorial.com/es/elixir/topic/2204/sigilos

Capítulo 38: Tarea

Sintaxis

- Task.async (divertido)
- Task.await (tarea)

Parámetros

Parámetro	Detalles
divertido	La función que se debe ejecutar en un proceso separado.
tarea	La tarea devuelta por `Task.async` .

Examples

Haciendo trabajos de fondo.

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

Procesamiento en paralelo

```
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)
```

Lea Tarea en línea: https://riptutorial.com/es/elixir/topic/7588/tarea

Capítulo 39: Tipos incorporados

Examples

Números

El elixir viene con **números enteros** y **números de punto flotante** . Un **literal entero** se puede escribir en formatos decimal, binario, octal y hexadecimal.

```
iex> x = 291
291

iex> x = 0b100100011
291

iex> x = 0o443
291

iex> x = 0x123
291
```

Como Elixir usa la aritmética bignum, **el rango de enteros solo está limitado por la memoria disponible en el sistema** .

Los números de punto flotante son de doble precisión y siguen la especificación IEEE-754.

```
iex> x = 6.8
6.8

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

Tenga en cuenta que el elixir también es compatible con la forma exponencial para flotadores.

```
iex> 1 + 1
2

iex> 1.0 + 1.0
2.0
```

Primero agregamos dos números enteros, y el resultado es un entero. Luego, agregamos dos números de punto flotante, y el resultado es un número de punto flotante.

La división en Elixir siempre devuelve un número de punto flotante:

```
iex> 10 / 2
5.0
```

De la misma manera, si sumas, restas o multiplicas un número entero por un número de punto flotante, el resultado será un punto flotante:

```
iex> 40.0 + 2
42.0

iex> 10 - 5.0
5.0

iex> 3 * 3.0
9.0
```

Para la división entera, se puede usar la función `div/2` :

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

Los átomos

Los átomos son constantes que representan un nombre de alguna cosa. El valor de un átomo es su nombre. Un nombre de átomo comienza con dos puntos.

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

El nombre de un átomo es único. Dos átomos con los mismos nombres siempre son iguales.

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

iex(2)> b = :atom
:atom

iex(3)> a == b
true

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

Los booleanos `true` y `false` , en realidad son átomos.

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

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

Los átomos se almacenan en la tabla de átomos especiales. Es muy importante saber que esta tabla no es recogida de basura. Entonces, si quieres (o accidentalmente es un hecho) constantemente crear átomos, es una mala idea.

Binarios y cadenas de bits

Los binarios en elixir se crean utilizando la construcción Kernel.SpecialForms << >> .

Son una herramienta poderosa que hace que Elixir sea muy útil para trabajar con protocolos

binarios y codificaciones.

Los binarios y las cadenas de bits se especifican mediante una lista de valores enteros o variables delimitados por comas, incluidos en "<<" y ">>". Se componen de 'unidades', ya sea una agrupación de bits o una agrupación de bytes. La agrupación predeterminada es un byte único (8 bits), especificado mediante un entero:

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

Las cadenas de elixir también se convierten directamente a binarios:

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

Puede agregar "especificadores" a cada "segmento" de un binario, lo que le permite codificar:

- Tipo de datos
- tamaño
- Endianness

Estos especificadores se codifican siguiendo cada valor o variable con el operador "::":

```
<<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
```

Los tipos de datos disponibles que puede utilizar son:

- entero
- flotador
- bits (alias para cadena de bits)
- cadena de bits
- binario
- bytes (alias para binario)
- utf8
- utf16
- utf32

Tenga en cuenta que al especificar el 'tamaño' del segmento binario, varía de acuerdo con el 'tipo' elegido en el especificador de segmento:

- entero (predeterminado) 1 bit
- flotar 1 bit
- binarios de 8 bits

Lea Tipos incorporados en línea: https://riptutorial.com/es/elixir/topic/1774/tipos-incorporados

Capítulo 40: Unir cuerdas

Examples

Usando la interpolación de cadenas

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

Usando la lista IO

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

Esto le dará algunas mejoras de rendimiento como cadenas no duplicadas en la memoria.
Método alternativo:

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

Usando Enum.join

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

Lea Unir cuerdas en línea: https://riptutorial.com/es/elixir/topic/9202/unir-cuerdas

Capítulo 41: uso básico de cláusulas de guardia

Examples

Usos básicos de las cláusulas de guardia.

En Elixir, se pueden crear múltiples implementaciones de una función con el mismo nombre y especificar reglas que se aplicarán a los parámetros de la función *antes de llamar a la función* para determinar qué implementación ejecutar.

Estas reglas están marcadas por la palabra clave `when` , y van entre `def function_name(params)` y `do` en la definición de la función. Un ejemplo trivial:

```
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
```

Digamos que ejecuto `Math.is_even(2)` con este ejemplo. Hay dos implementaciones de `is_even` , con diferentes cláusulas de protección. El sistema los examinará en orden y ejecutará la primera implementación donde los parámetros satisfacen la cláusula de protección. El primero especifica que `num === 1` que no es verdadero, por lo que pasa al siguiente. El segundo especifica que `num === 2` , que es verdadero, por lo que esta es la implementación que se usa, y el valor de retorno será `true` .

¿Qué pasa si ejecuto `Math.is_odd(1)` ? El sistema analiza la primera implementación y ve que, dado que `num` es `1` se cumple la cláusula de protección de la primera implementación. Luego usará esa implementación y devolverá la `true` , y no se molestará en mirar cualquier otra implementación.

Los guardias están limitados en los tipos de operaciones que pueden ejecutar. La documentación de Elixir enumera todas las operaciones permitidas ; en pocas palabras, permiten comparaciones, operaciones matemáticas, operaciones binarias, verificación de tipos (por ejemplo, `is_atom`) y un puñado de pequeñas funciones de conveniencia (por ejemplo, `length`). Es posible definir

cláusulas de protección personalizadas, pero requiere la creación de macros y es mejor dejarlas para una guía más avanzada.

Tenga en cuenta que los guardias no lanzan errores; se tratan como fallas normales de la cláusula de protección, y el sistema avanza para ver la siguiente implementación. Si descubre que no está `(FunctionClauseError) no function clause matching` al llamar a una función protegida con los parámetros que espera que funcionen, es posible que una cláusula de protección con la que espera trabajar genere un error que se está tragando.

Para ver esto por ti mismo, crea y luego llama a una función con un guardia que no tiene sentido, como este, que trata de dividir por cero:

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

Llamar a `BadMath.divide("anything")` proporcionará el error algo inútil `(FunctionClauseError) no function clause matching in BadMath.divide/1` - mientras que si hubiera intentado ejecutar `"anything" / 0` directamente, obtendría una más útil error: `(ArithmeticError) bad argument in arithmetic expression`.

Lea uso básico de cláusulas de guardia en línea: https://riptutorial.com/es/elixir/topic/6121/uso-basico-de-clausulas-de-guardia

Creditos

S. No	Capítulos	Contributors
1	Empezando con 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	Comportamientos	Yos Riady
3	Condicionales	Andrey Chernykh, evuez, javanut13, Musfiqur Rahman, Paweł Obrok
4	Consejos de depuración	javanut13, Paweł Obrok, Pfitz, Philippe-Arnaud de MANGOU, sbs
5	Consejos y trucos	Ankanna
6	Consejos y trucos de la consola IEx	alxndr, Cifer, fahrradflucht, legoscia, mudasobwa, muttonlamb, PatNowak, Paweł Obrok, sbs, Sheharyar, Simone Carletti, Stephan Rodemeier, Uniaika, Vincent, Yos Riady
7	Constantes	ibgib
8	Corriente	Oskar
9	Doctests	aholt, milmazz, Philippe-Arnaud de MANGOU, Yos Riady
10	Ecto	fgutierr, Philippe-Arnaud de MANGOU, toraritte
11	El polimorfismo en el elixir	mustafaturan
12	Erlang	4444, Yos Riady
13	Estructuras de datos	Sam Mercier, Simone Carletti, Stephan Rodemeier, Yos Riady
14	Exdoc	milmazz, Yos Riady
15	ExUnidad	Yos Riady
16	Funciones	Andrey Chernykh, cwc, Dair, Eiji, Filip Haglund, PatNowak, rainteller, Simone Carletti, Stephan Rodemeier, Yedhu Krishnan , Yos Riady
17	HAZ	Yos Riady

18	Instalación	cwc, Douglas Correa, Eiji, JAlberto, MattW.
19	Instrumentos de cuerda	Alex G, Sheharyar, Yos Riady
20	La coincidencia de patrones	Alex Anderson, Dair, Danny Rosenblatt, evuez, Gabriel C, gmile , Harrison Lucas, javanut13, Oskar, PatNowak, theIV, Thomas, Yedhu Krishnan
21	Liza	Ben Bals, Candy Gumdrop, emoragaf, PatNowak, Sheharyar, Yos Riady
22	Los operadores	alxndr, Andrey Chernykh, Dair, Gazler, Mitkins, nirev, PatNowak
23	Manejo de estado en elixir	Paweł Obrok
24	Mapas y listas de palabras clave	Sam Mercier, Simone Carletti, Yos Riady
25	Mejor depuración con IO. Inspección y etiquetas.	leifg
26	Mejoramiento	Filip Haglund, legoscia
27	Metaprogramacion	4444, Paweł Obrok
28	Mezcla	4444, helcim, rainteller, Slava.K, Yos Riady
29	Módulos	Alex G, javanut13, Yos Riady
30	Nodos	Yos Riady
31	Obteniendo ayuda en la consola IEx	helcim
32	Procesos	Alex G, Yedhu Krishnan
33	Programa básico .gitignore para elixir	Yos Riady
34	Programación funcional en el elixir.	Dinesh Balasubramanian
35	Protocolos	Yos Riady
36	Sigilos	javanut13, Yos Riady
37	Tarea	mario

38	Tipos incorporados	Andrey Chernykh, Arithmeticbird, Oskar, TreyE, Vinicius Quaiato
39	Unir cuerdas	Agung Santoso
40	uso básico de cláusulas de guardia	alxndr