Kostenloses eBook

LERNEN Elixir Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#elixir

Inhaltsverzeichnis

Über

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

Kapitel 1: Erste Schritte mit Elixir Language

Bemerkungen

Elixir ist eine dynamische, funktionale Sprache, die zum Erstellen von skalierbaren und wartbaren Anwendungen entwickelt wurde.

Elixir nutzt die Erlang-VM, die dafür bekannt ist, Systeme mit geringer Latenz, verteilte und fehlertolerante Systeme auszuführen, und gleichzeitig erfolgreich in der Webentwicklung und der Embedded-Softwaredomäne eingesetzt wird.

Versionen

Ausführung	Veröffentlichungsdatum
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

Hallo Welt

Installationsanweisungen für elixir check hier , beschreibt Anweisungen für verschiedene Plattformen.

Elixir ist eine Programmiersprache, die mit `erlang` wird und die `BEAM` Laufzeit von `erlang` verwendet (wie `JVM` für Java).

Wir können Elixier in zwei Modi verwenden: interaktiv Shell `iex` oder direkt unter fließendem `elixir` Befehl.

`hello.exs` Folgendes in eine Datei namens `hello.exs` :

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

Geben Sie in der Befehlszeile den folgenden Befehl ein, um die Elixir-Quelldatei auszuführen:

```
$ elixir hello.exs
```

Dies sollte ausgeben:

    Hallo Welt!

Dies ist als *Skriptmodus* von `Elixir` . Tatsächlich können Elixir-Programme auch in Bytecode für die virtuelle BEAM-Maschine kompiliert werden (und dies sind sie im Allgemeinen).

Sie können `iex` für die interaktive Elixier-Shell (empfohlen) verwenden. Führen Sie den Befehl aus, um eine Eingabeaufforderung wie folgt zu erhalten:

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

Hier können Sie Ihre Elixier- `hello world` Beispiele ausprobieren:

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

Sie können Ihre Module auch über `iex` kompilieren und `iex` . Wenn Sie beispielsweise eine `helloworld.ex` haben, die `helloworld.ex` enthält:

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

Durch `iex` :

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

Hallo Welt von IEx

Sie können auch die `IEx` Shell (Interactive Elixir) verwenden, um Ausdrücke auszuwerten und Code auszuführen.

Wenn Sie Linux oder Mac verwenden, geben Sie einfach iex in Ihre Bash ein und drücken Sie die Eingabetaste:

```
$ iex
```

Wenn Sie sich auf einem Windows-Computer befinden, geben Sie Folgendes ein:

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

Dann werden Sie in den IEx REPL (Lesen, Bewerten, Drucken, Wiederholen) eingehen, und Sie können Folgendes eingeben:

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

Wenn Sie ein Skript laden möchten, während Sie eine IEx REPL öffnen, können Sie Folgendes tun:

```
$ iex script.exs
```

Gegebene `script.exs` ist Ihr Skript. Sie können jetzt Funktionen aus dem Skript in der Konsole aufrufen.

Erste Schritte mit Elixir Language online lesen: https://riptutorial.com/de/elixir/topic/954/erste-schritte-mit-elixir-language

Kapitel 2: Aufgabe

Syntax

- Task.async (Spaß)
- Task.await (Aufgabe)

Parameter

Parameter	Einzelheiten
Spaß	Die Funktion, die in einem separaten Prozess ausgeführt werden soll.
Aufgabe	Die von `Task.async` zurückgegebene `Task.async` .

Examples

Arbeit im Hintergrund

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

Parallelverarbeitung

```
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)
```

Aufgabe online lesen: https://riptutorial.com/de/elixir/topic/7588/aufgabe

Kapitel 3: Basic .gitignore für das Elixierprogramm

Basic .gitignore für das Elixierprogramm online lesen:
https://riptutorial.com/de/elixir/topic/6493/basic--gitignore-fur-das-elixierprogramm

Kapitel 4: Basic .gitignore für das Elixierprogramm

Bemerkungen

Beachten Sie, dass der Ordner `/rel` möglicherweise nicht in Ihrer .gitignore-Datei benötigt wird. Dies wird generiert, wenn Sie ein Release-Management-Tool wie z. B. `exrm`

Examples

Ein grundlegender .gitignore für 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
```

Beispiel

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

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

Standalone-Elixier-Anwendung

```
/_build
/cover
```

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

Phoenix-Anwendung

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

Automatisch erzeugter .gitignore

Standardmäßig erzeugt `mix new <projectname>` eine `.gitignore` Datei im Projektstamm, die für Elixir geeignet ist.

```
# 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
```

Basic .gitignore für das Elixierprogramm online lesen:
https://riptutorial.com/de/elixir/topic/6526/basic--gitignore-fur-das-elixierprogramm

Kapitel 5: Besseres Debuggen mit IO.inspect und Labels

Einführung

`IO.inspect` ist sehr nützlich, wenn Sie versuchen, Ihre Ketten des Methodenaufrufs zu debuggen. Es kann jedoch unordentlich werden, wenn Sie es zu oft verwenden.

Ab Elixir 1.4.0 kann die `label` Option von `IO.inspect` helfen

Bemerkungen

Funktioniert nur mit Elixir 1.4+, aber ich kann das noch nicht kennzeichnen.

Examples

Ohne Etiketten

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

Dies führt zu einer Menge Ausgabe ohne Kontext:

```
"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}
```

Mit Etiketten

Die Verwendung der `label` Option zum Hinzufügen von Kontext kann sehr helfen:

```
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}
```

Besseres Debuggen mit IO.inspect und Labels online lesen:
https://riptutorial.com/de/elixir/topic/8725/besseres-debuggen-mit-io-inspect-und-labels

Kapitel 6: Conditionals

Bemerkungen

Beachten Sie, dass die `do...end` Syntax syntaktischer Zucker für reguläre Keyword-Listen ist. Sie können also Folgendes tun:

```
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

Fall

```
 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` wird nur verwendet, um das gegebene Muster der jeweiligen Daten abzugleichen. Hier `{1,2}` mit unterschiedlichen Fallmustern überein, die im Codebeispiel angegeben sind.

wenn und soweit nicht

```
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` `CondClauseError` einen `CondClauseError` wenn keine Bedingungen erfüllt sind.

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

Dies kann vermieden werden, indem eine Bedingung hinzugefügt wird, die immer wahr ist.

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

Es sei denn, es wird nie erwartet, dass der Standardfall erreicht wird, und das Programm sollte tatsächlich zu diesem Zeitpunkt abstürzen.

mit Klausel

`with` Klausel wird verwendet, um übereinstimmende Klauseln zu kombinieren. Anscheinend kombinieren wir anonyme Funktionen oder behandeln Funktionen mit mehreren Körpern (übereinstimmende Klauseln). Beachten Sie den Fall: Wir erstellen einen Benutzer, fügen ihn in die Datenbank ein, erstellen eine Begrüßungs-E-Mail und senden sie an den Benutzer.

Ohne die `with` Klausel könnten wir so etwas schreiben (ich habe Funktionsimplementierungen ausgelassen):

```
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
```

Hier haben wir unsere Geschäftsprozesse des Flusses mit Griff `case` (es könnte sein , `cond` oder `if`). Das führt uns zu einer sogenannten 'Pyramide des Schicksals' , weil wir uns mit möglichen

Bedingungen auseinandersetzen müssen und entscheiden müssen, ob wir uns weiter bewegen oder nicht. Es wäre viel schöner, diesen Code `with` Anweisung neu zu schreiben:

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

In dem obigen Code-Snippet haben wir geschachtelte `case` Klauseln mit `with` umgeschrieben. Innerhalb `with` rufen wir einige Funktionen (entweder anonym oder benannt) und Musterübereinstimmung an ihren Ausgaben auf. Wenn alle aufeinander abgestimmt, `with` Rückkehr `do` Block - Ergebnis oder `else` sonst Block Ergebnis.

Wir können weglassen `else` so `with` kehren entweder `do` Block - Ergebnis oder das erste Ergebnis ausfallen.

Der Wert von `with` Anweisung ist `do` Ergebnis des `do` Blocks.

Conditionals online lesen: https://riptutorial.com/de/elixir/topic/2118/conditionals

Kapitel 7: Datenstrukturen

Syntax

- [Kopf | tail] = [1, 2, 3, true] # man kann Pattern Matching verwenden, um Kons-Zellen aufzubrechen. Dies weist Kopf zu 1 und Schwanz zu [2, 3, wahr] zu.
- % {d: val} =% {d: 1, e: true} # dieser Wert wird 1 zugewiesen; Es wird keine Variable d erstellt, da das d auf dem lhs eigentlich nur ein Symbol ist, das zum Erstellen des Musters% {: d => _} verwendet wird in Rubin)

Bemerkungen

Zu welcher Datenstruktur wir hier einige kurze Anmerkungen machen.

Wenn Sie eine Array-Datenstruktur benötigen, müssen Sie häufig Listen schreiben. Wenn Sie stattdessen viel lesen, sollten Sie Tupel verwenden.

Bei Karten handelt es sich lediglich um das Speichern von Schlüsselwerten.

Examples

Listen

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

Beachten Sie, dass diese als verknüpfte Listen gespeichert werden. Dies ist eine Reihe von Konsumzellen, bei denen der Kopf (List.hd / 1) der Wert des ersten Elements der Liste ist und das Ende (List.tail / 1) der Wert des Restes der Liste ist.

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

Tuples

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

Tupel entsprechen Arrays in anderen Sprachen. Sie werden zusammenhängend im Speicher abgelegt.

Datenstrukturen online lesen: https://riptutorial.com/de/elixir/topic/1607/datenstrukturen

Kapitel 8: Doktests

Examples

Einführung

Wenn Sie Ihren Code mit `@doc` , können Sie Codebeispiele wie `@doc` :

```
# 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
```

Sie können die Codebeispiele als Testfälle in eine Ihrer Testsuiten einfügen:

```
# myproject/test/doc_test.exs

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

Anschließend können Sie Ihre Tests mit dem `mix test` .

HTML-Dokumentation basierend auf doctest erstellen

Da das Generieren von Dokumentation auf Markdown basiert, müssen Sie zwei Dinge tun:

1 / Schreiben Sie Ihr Doctest und machen Sie Ihre Doctest-Beispiele klarer, um die Lesbarkeit zu verbessern (Es ist besser, eine Überschrift wie "Beispiele" oder "Tests" zu geben). Vergessen Sie beim Schreiben Ihrer Tests nicht, Ihrem Testcode 4 Leerzeichen zuzuweisen, damit er in der HTML-Dokumentation als Code formatiert wird.

2 / Geben Sie anschließend in der Konsole im Stammverzeichnis Ihres Elixir-Projekts "mix docs" ein, um die HTML-Dokumentation im doc-Verzeichnis im Stammverzeichnis Ihres elixir-Projekts zu generieren.

```
$> mix docs
```

Mehrzeilige doctests

Sie können einen mehrzeiligen Doctest durchführen, indem Sie für die folgenden Zeilen '...>' verwenden

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

Doktests online lesen: https://riptutorial.com/de/elixir/topic/2708/doktests

Kapitel 9: Ecto

Examples

Hinzufügen eines Ecto.Repo in einem Elixierprogramm

Dies kann in 3 Schritten erfolgen:

1. Sie müssen ein Elixiermodul definieren, das Ecto.Repo verwendet, und Ihre App als otp_app registrieren.

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

2. Sie müssen auch eine Konfiguration für das Repo definieren, mit der Sie eine Verbindung zur Datenbank herstellen können. Hier ist ein Beispiel mit 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. Bevor Sie Ecto in Ihrer Anwendung verwenden, müssen Sie sicherstellen, dass Ecto gestartet wird, bevor Ihre App gestartet wird. Dies kann mit der Registrierung von Ecto in lib / custom_app.ex als Supervisor erfolgen.

```
    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
```

"and" -Klausel in einem Repo.get_by / 3

Wenn Sie eine Ecto.Queryable haben, heißt Post, die einen Titel und eine Beschreibung hat.

Sie können den Beitrag mit Titel: "Hallo" und Beschreibung: "Welt" abrufen, indem Sie Folgendes ausführen:

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

All dies ist möglich, da Repo.get_by im zweiten Argument eine Keyword-Liste erwartet.

Abfragen mit dynamischen Feldern

Um ein Feld abzufragen, dessen Name in einer Variablen enthalten ist, verwenden Sie die Feldfunktion .

```
some_field = :id
some_value = 10

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

Fügen Sie der Migration und dem Schema benutzerdefinierte Datentypen hinzu

(Aus dieser Antwort)

Das folgende Beispiel fügt einer Postgres-Datenbank einen Aufzählungstyp hinzu.

Bearbeiten Sie zuerst die **Migrationsdatei** (erstellt mit `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
```

Zweitens fügen Sie in der **Modelldatei** entweder ein Feld mit einem Elixir-Typ hinzu:

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

oder implementieren Sie das Verhalten von `Ecto.Type` .

Ein gutes Beispiel für Letzteres ist das Paket `ecto_enum` , das als Vorlage verwendet werden kann. Ihre Verwendung ist auf der **Github-Seite** gut dokumentiert.

Dieses Commit zeigt ein Beispiel für die Verwendung in einem Phoenix-Projekt, indem enum_ecto zum Projekt hinzugefügt und der aufgezählte Typ in Ansichten und Modellen verwendet wird.

Ecto online lesen: https://riptutorial.com/de/elixir/topic/6524/ecto

Kapitel 10: Eingebaute Typen

Examples

Zahlen

Elixir kommt mit **Ganzzahlen** und **Fließkommazahlen** . Ein **Integer-Literal** kann in Dezimal-, Binär-, Oktal- und Hexadezimalformaten geschrieben werden.

```
iex> x = 291
291

iex> x = 0b100100011
291

iex> x = 0o443
291

iex> x = 0x123
291
```

Da Elixir Bignum-Arithmetik verwendet, ist **der Bereich der Ganzzahl nur durch den verfügbaren Speicher des Systems begrenzt** .

Fließkommazahlen haben eine doppelte Genauigkeit und entsprechen der IEEE-754-Spezifikation.

```
iex> x = 6.8
6.8

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

Beachten Sie, dass Elixir auch Exponentenform für Floats unterstützt.

```
iex> 1 + 1
2

iex> 1.0 + 1.0
2.0
```

Zuerst haben wir zwei Ganzzahlen hinzugefügt und das Ergebnis ist eine Ganzzahl. Später haben wir zwei Fließkommazahlen hinzugefügt, und das Ergebnis ist eine Fließkommazahl.

Bei der Division in Elixir wird immer eine Fließkommazahl zurückgegeben:

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

Wenn Sie eine Ganzzahl mit einer Gleitkommazahl addieren, subtrahieren oder multiplizieren,

wird das Ergebnis auf die gleiche Weise Fließkomma:

```
iex> 40.0 + 2
42.0

iex> 10 - 5.0
5.0

iex> 3 * 3.0
9.0
```

Für die Ganzzahlteilung kann man die `div/2` Funktion verwenden:

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

Atome

Atome sind Konstanten, die einen Namen für etwas darstellen. Der Wert eines Atoms ist sein Name. Ein Atomname beginnt mit einem Doppelpunkt.

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

Der Name eines Atoms ist einzigartig. Zwei gleichnamige Atome sind immer gleich.

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

iex(2)> b = :atom
:atom

iex(3)> a == b
true

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

Boolean ist `true` und `false` , eigentlich sind Atome.

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

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

Atome werden in der Tabelle der speziellen Atome gespeichert. Es ist sehr wichtig zu wissen, dass dieser Tisch nicht mit Müll gesammelt wird. Wenn Sie also ständig Atome erzeugen wollen (oder aus Versehen eine Tatsache sind), ist dies eine schlechte Idee.

Binaries und Bitstrings

Binaries in Elixir werden mit dem Kernel.SpecialForms-Konstrukt << >> erstellt .

Sie sind ein leistungsfähiges Werkzeug, das Elixir sehr nützlich für die Arbeit mit binären Protokollen und Kodierungen macht.

Binaries und Bitstrings werden durch eine durch Kommas getrennte Liste von Ganzzahlen oder Variablenwerten angegeben, die durch "<<" und ">>" ergänzt werden. Sie bestehen aus "Einheiten", entweder einer Bit-Gruppe oder einer Byte-Gruppe. Die Standardgruppierung ist ein einzelnes Byte (8 Bit), das mit einer Ganzzahl angegeben wird:

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

Elixir-Strings werden auch direkt in Binärdateien konvertiert:

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

Sie können jedem "Segment" einer Binärdatei "Spezifizierer" hinzufügen, sodass Sie Folgendes kodieren können:

- Datentyp
- Größe
- Endianness

Diese Bezeichner werden durch Folgen jedes Werts oder jeder Variablen mit dem Operator "::" codiert:

```
<<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
```

Die verfügbaren Datentypen, die Sie verwenden können, sind:

- ganze Zahl
- schweben
- Bits (Alias für Bitstring)
- Bitstring
- binär
- Bytes (Alias für binär)
- utf8
- utf16
- utf32

Beachten Sie, dass die Angabe der 'Größe' des binären Segments je nach dem im Segmentbezeichner ausgewählten 'Typ' variiert:

- Ganzzahl (Standard) 1 Bit

- Float 1 Bit
- binär 8 Bit

Eingebaute Typen online lesen: https://riptutorial.com/de/elixir/topic/1774/eingebaute-typen

Kapitel 11: Erlang

Examples

Erlang verwenden

Erlang-Module sind als Atome erhältlich. Das Erlang-Mathematikmodul ist beispielsweise verfügbar als `:math` :

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

Überprüfen Sie ein Erlang-Modul

Verwenden Sie `module_info` für Erlang-Module, die Sie untersuchen möchten:

```
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>>]
```

Erlang online lesen: https://riptutorial.com/de/elixir/topic/2716/erlang

Kapitel 12: ExDoc

Examples

Einführung

So generiert Dokumentation im `HTML` - Format von `@doc` und `@moduledoc` Attributen in Ihrem Quellcode, fügt `ex_doc` und einen Abschlag Prozessor, gerade jetzt ExDoc unterstützt Earmark , Pandoc , Hoedown und Cmark , als Abhängigkeiten in Ihre `mix.exs` Datei:

```
# config/mix.exs

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

Wenn Sie einen anderen Markdown-Prozessor verwenden möchten, finden Sie weitere Informationen im Abschnitt Ändern des Markdown-Werkzeugs .

Sie können Markdown in den Attributen Elixir `@doc` und `@moduledoc` .

Führen Sie dann `mix docs` .

Dabei ist zu beachten, dass ExDoc Konfigurationsparameter zulässt, beispielsweise:

```
  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
```

Weitere Informationen zu diesen Konfigurationsoptionen finden Sie in den Hilfedokumenten zu `mix help docs`

ExDoc online lesen: https://riptutorial.com/de/elixir/topic/3582/exdoc

Kapitel 13: ExUnit

Examples

Ausnahmen geltend machen

Verwenden Sie `assert_raise` , um zu testen, ob eine Ausnahme `assert_raise` wurde. `assert_raise` nimmt eine Exception und eine `assert_raise` Funktion auf.

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

Packen Sie den Code, den Sie testen möchten, in eine anonyme Funktion und übergeben Sie ihn an `assert_raise` .

ExUnit online lesen: https://riptutorial.com/de/elixir/topic/3583/exunit

Kapitel 14: Funktionale Programmierung in Elixir

Einführung

Versuchen wir, die grundlegenden Funktionen höherer Ordnung, wie z. B. Zuordnen und Reduzieren, mit Elixir zu implementieren

Examples

Karte

Map ist eine Funktion, die ein Array und eine Funktion übernimmt und ein Array zurückgibt, nachdem diese Funktion auf ***jedes Element*** in dieser Liste angewendet wurde

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

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

Einfügen in `iex` und Ausführen:

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

`MyList.map [1,2,3], &(&1 * 5)` ist `MyList.map [1,2,3], &(&1 * 5)`

Reduzieren

Reduce ist eine Funktion, die ein Array, eine Funktion und einen Akkumulator benötigt und einen ***Akkumulator als Startwert verwendet, um die Iteration mit dem ersten Element zu starten, um den nächsten Akkumulator zu erhalten, und die Iteration wird für alle Elemente im Array fortgesetzt*** (siehe Beispiel unten).

```
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
```

Kopieren Sie den obigen Ausschnitt in iex:

1. Um alle Zahlen in einem Array `MyList.reduce [1,2,3,4], fn acc, element -> acc + element end, 0` : `MyList.reduce [1,2,3,4], fn acc, element -> acc + element end, 0`
2. Um alle Zahlen in einem Array zu `MyList.reduce [1,2,3,4], fn acc, element -> acc * element end, 1` : `MyList.reduce [1,2,3,4], fn acc, element -> acc * element end, 1`

Erklärung für Beispiel 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)
```

Filtern Sie die Liste mit verkleinern

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

Funktionale Programmierung in Elixir online lesen:
https://riptutorial.com/de/elixir/topic/10186/funktionale-programmierung-in-elixir

Kapitel 15: Funktionen

Examples

Anonyme Funktionen

In Elixir ist es üblich, anonyme Funktionen zu verwenden. Das Erstellen einer anonymen Funktion ist einfach:

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

Die allgemeine Syntax lautet:

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

Zur besseren Lesbarkeit können Sie die Argumente in Klammern setzen:

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

Um eine anonyme Funktion aufzurufen, rufen Sie sie mit dem zugewiesenen Namen auf und fügen Sie hinzu `.` zwischen dem Namen und den Argumenten.

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

Es ist möglich, anonyme Funktionen ohne Argumente zu deklarieren:

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

Verwenden des Erfassungsoperators

Um anonyme Funktionen übersichtlicher zu gestalten, können Sie den **Capture-Operator** `&` . Zum Beispiel anstelle von:

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

Du kannst schreiben:

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

Verwenden Sie bei mehreren Parametern die Nummer, die jedem Argument entspricht, von `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
```

Mehrere Körper

Eine anonyme Funktion kann auch mehrere Körper enthalten (als Ergebnis eines Musterabgleichs):

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

Wenn Sie eine Funktion mit mehreren Körpern aufrufen, versucht Elixir, die von Ihnen angegebenen Parameter mit dem richtigen Körperteil der Funktion abzugleichen.

Schlüsselwortlisten als Funktionsparameter

Verwenden Sie Schlüsselwortlisten für 'options'-style Parameter, die mehrere Schlüssel-Wert-Paare enthalten:

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

Wir können die Funktion wie folgt aufrufen:

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

was äquivalent ist zu:

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

Die Argumentwerte sind als `opts.pizza` bzw. `opts.soda` verfügbar.
Alternativ können Sie Atome verwenden: `opts[:pizza]` und `opts[:soda]` .

Benannte Funktionen und private Funktionen

Benannte Funktionen

```
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
```

Private Funktionen

```
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
```

Musterabgleich

Elixir ordnet einen Funktionsaufruf seinem Körper zu, basierend auf dem Wert seiner Argumente.

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

Hier stimmt die Fakultät positiver Zahlen mit der zweiten Klausel überein, während `factorial(0)` mit der ersten übereinstimmt. (Negative Zahlen werden der Einfachheit halber ignoriert). Elixir versucht die Funktionen von oben nach unten abzugleichen. Wenn die zweite Funktion über der ersten geschrieben wird, wird ein unerwartetes Ergebnis erwartet, da sie zu einer endlosen Rekursion führt. Weil `factorial(0)` zu `factorial(n)` passt

Leitsätze

Guard-Klauseln ermöglichen es uns, die Argumente vor der Ausführung der Funktion zu überprüfen. Guard-Klauseln werden aufgrund ihrer Lesbarkeit normalerweise `if` und `cond` vorgezogen, um dem Compiler eine bestimmte Optimierungstechnik zu erleichtern. Die erste

Funktionsdefinition, bei der alle Wächter übereinstimmen, wird ausgeführt.

Hier ist eine Beispielimplementierung der Faktorialfunktion unter Verwendung von Guards und Pattern Matching.

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

Das erste Muster stimmt überein, wenn (und nur wenn) das Argument `0` . Wenn das Argument nicht `0` , schlägt die Musterübereinstimmung fehl und die nächste Funktion wird geprüft.

Diese zweite Funktionsdefinition hat eine Schutzklausel: `when n > 0` . Das bedeutet, dass diese Funktion nur übereinstimmt, wenn das Argument `n` größer als `0` . Schließlich ist die mathematische Faktorfunktion für negative ganze Zahlen nicht definiert.

Wenn keine Funktionsdefinition (einschließlich ihrer Pattern Matching- und Guard-Klauseln) übereinstimmt, wird ein `FunctionClauseError` ausgelöst. Dies geschieht für diese Funktion, wenn wir eine negative Zahl als Argument übergeben, da sie nicht für negative Zahlen definiert ist.

Beachten Sie, dass dieser `FunctionClauseError` selbst kein Fehler ist. Die Rückgabe von `-1` oder `0` oder eines anderen "Fehlerwerts", wie er in anderen Sprachen üblich ist, würde die Tatsache verdecken, dass Sie eine undefinierte Funktion aufgerufen haben, die die Fehlerquelle verbirgt und möglicherweise einen sehr schmerzhaften Fehler für zukünftige Entwickler verursacht.

Standardparameter

Sie können Standardparameter mit der folgenden Syntax an jede benannte Funktion übergeben: `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]
```

Capture-Funktionen

Verwenden Sie `&` , um Funktionen von anderen Modulen zu erfassen. Sie können die erfassten Funktionen direkt als Funktionsparameter oder in anonymen Funktionen verwenden.

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

Kann mit `&` noch übersichtlicher gestaltet werden:

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

Um Funktionen erfassen zu können, ohne Argumente zu übergeben, müssen Sie deren Art explizit angeben, z. B. `&String.capitalize/1` :

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

Funktionen online lesen: https://riptutorial.com/de/elixir/topic/2442/funktionen

Kapitel 16: grundlegende Verwendung von Schutzklauseln

Examples

grundlegende Verwendungen von Schutzklauseln

In Elixir kann man mehrere Implementierungen einer Funktion mit demselben Namen erstellen und Regeln festlegen, die auf die Parameter der Funktion angewendet werden, *bevor die Funktion* aufgerufen wird, um zu bestimmen, welche Implementierung ausgeführt werden soll.

Diese Regeln werden durch das Schlüsselwort `when` markiert und `def function_name(params) do` in der Funktionsdefinition zwischen `def function_name(params)` und `do` . Ein triviales Beispiel:

```
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
```

`Math.is_even(2)` ich `Math.is_even(2)` mit diesem Beispiel `Math.is_even(2)` . Es gibt zwei Implementierungen von `is_even` mit unterschiedlichen Guard-Klauseln. Das System prüft sie der Reihe nach und führt die erste Implementierung aus, bei der die Parameter die Guard-Klausel erfüllen. Der erste gibt an, dass `num === 1` nicht wahr ist, also geht es zum nächsten weiter. Die zweite gibt an, dass `num === 2` , was wahr ist. Dies ist also die verwendete Implementierung, und der Rückgabewert ist `true` .

Was ist, wenn ich `Math.is_odd(1)` ? Das System betrachtet die erste Implementierung und stellt fest, dass seit `num 1` die Schutzklausel der ersten Implementierung erfüllt ist. Es wird dann diese Implementierung verwenden und `true` , ohne sich um andere Implementierungen zu kümmern.

Wachen sind in den Arten von Operationen, die sie ausführen können, eingeschränkt. Die Elixir-Dokumentation listet jede zulässige Operation auf . `is_atom` gesagt, sie ermöglichen Vergleiche, mathematische Operationen, binäre Operationen, Typprüfungen (z. B. `is_atom`) und eine Handvoll kleiner Komfortfunktionen (z. B. `length`). Es ist möglich, benutzerdefinierte Schutzklauseln zu definieren, es erfordert jedoch das Erstellen von Makros und sollte am besten für eine erweiterte

Anleitung verwendet werden.

Beachten Sie, dass Wachen keine Fehler werfen. Sie werden als normale Fehler der Guard-Klausel behandelt, und das System betrachtet die nächste Implementierung. Wenn Sie feststellen, dass Sie `(FunctionClauseError) no function clause matching` wenn Sie eine überwachte Funktion mit den von Ihnen erwarteten Params aufrufen, kann es sein, dass eine von Ihnen erwartete Guard-Klausel einen Fehler ausgibt, der verschluckt wird.

Um dies für sich selbst zu sehen, erstellen Sie eine Funktion und rufen Sie sie dann mit einem Wächter auf, der keinen Sinn hat, wie z. B. dieser, die versucht, durch Null zu teilen:

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

Durch den Aufruf von `BadMath.divide("anything")` wird der etwas wenig hilfreiche Fehler `(FunctionClauseError) no function clause matching in BadMath.divide/1` Wenn Sie jedoch versucht haben, `"anything" / 0` direkt auszuführen, erhalten Sie hilfreiche `(FunctionClauseError) no function clause matching in BadMath.divide/1` Fehler: `(ArithmeticError) bad argument in arithmetic expression` .

grundlegende Verwendung von Schutzklauseln online lesen:
https://riptutorial.com/de/elixir/topic/6121/grundlegende-verwendung-von-schutzklauseln

Kapitel 17: Hilfe in der IEx-Konsole erhalten

Einführung

IEx bietet Zugriff auf die Elixir-Dokumentation. Wenn Elixir auf Ihrem System installiert ist, können Sie IEx zB mit `iex` Befehl `iex` in einem Terminal starten. Geben Sie dann den Befehl `h` in die IEx-Befehlszeile ein, gefolgt von dem Funktionsnamen, dem der `h List.foldr` vorangestellt ist, z

Examples

Auflisten von Elixir-Modulen und -Funktionen

Um die Liste der Elixir-Module abzurufen, geben Sie einfach ein

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

Durch Drücken von [TAB] werden die Namen der Module und Funktionen automatisch vervollständigt. In diesem Fall werden alle Module aufgelistet. Um alle Funktionen in einem Modul zu finden, z. B. `List` verwenden

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

Hilfe in der IEx-Konsole erhalten online lesen: https://riptutorial.com/de/elixir/topic/10780/hilfe-in-der-iex-konsole-erhalten

Kapitel 18: Installation

Examples

Fedora-Installation

```
dnf install erlang elixir
```

OSX-Installation

Unter OS X und MacOS kann Elixir über die üblichen Paketmanager installiert werden:

Homebrew

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

Macports

```
 $ sudo port install elixir
```

Debian / Ubuntu-Installation

```
# 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
```

Gentoo / Funtoo-Installation

Elixir ist im Hauptpaket-Repository verfügbar.
Aktualisieren Sie die Paketliste, bevor Sie ein Paket installieren:

```
emerge --sync
```

Dies ist eine einstufige Installation:

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

Installation online lesen: https://riptutorial.com/de/elixir/topic/4208/installation

Kapitel 19: Karten und Keyword-Listen

Syntax

- map =% {} // erstellt eine leere Map
- map =% {: a => 1,: b => 2} // erstellt eine nicht leere Map
- list = [] // erstellt eine leere Liste
- list = [{: a, 1}, {: b, 2}] // erstellt eine nicht leere Keyword-Liste

Bemerkungen

Elixir bietet zwei assoziative Datenstrukturen: *Karten* und *Keyword-Listen* .

Karten sind der Elixir-Schlüsselwert (in anderen Sprachen auch als Wörterbuch oder Hash bezeichnet).

Schlüsselwortlisten sind Schlüssel- / Wertetupel, die einem bestimmten Schlüssel einen Wert zuordnen. Sie werden im Allgemeinen als Optionen für einen Funktionsaufruf verwendet.

Examples

Karte erstellen

Karten sind der Elixir-Schlüsselwert (in anderen Sprachen auch als Wörterbuch oder Hash bezeichnet). Sie erstellen eine Map mit der `%w{}` -Syntax:

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

Schlüssel und Werte können beliebig sein:

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

Außerdem können Sie Karten mit gemischten Typen für Schlüssel und Werte erstellen ":

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

Wenn alle Schlüssel in einer Map aus Atomen bestehen, können Sie zur Vereinfachung die Schlüsselwortsyntax verwenden:

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

Erstellen einer Keyword-Liste

Schlüsselwortlisten sind Schlüssel / Wert-Tupel, die normalerweise als Optionen für einen Funktionsaufruf verwendet werden.

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

Bei Schlüsselwortlisten kann derselbe Schlüssel mehrmals wiederholt werden.

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

Schlüssel und Werte können beliebig sein:

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

Unterschied zwischen Karten und Keyword-Listen

Karten und Keyword-Listen haben unterschiedliche Anwendung. Zum Beispiel kann eine Karte nicht zwei Schlüssel mit demselben Wert haben und sie ist nicht geordnet. Umgekehrt kann die Verwendung einer Keyword-Liste in bestimmten Fällen ein wenig schwierig sein.

Hier sind einige Anwendungsfälle für Karten- und Keyword-Listen.

Verwenden Sie Stichwortlisten, wenn:

- Sie müssen die Elemente bestellen
- Sie benötigen mehr als ein Element mit demselben Schlüssel

Karten verwenden, wenn:

- Sie möchten mit einigen Schlüsseln / Werten ein Pattern-Match durchführen
- Sie benötigen nicht mehr als ein Element mit demselben Schlüssel
- wann immer Sie nicht explizit eine Keyword-Liste benötigen

Karten und Keyword-Listen online lesen: https://riptutorial.com/de/elixir/topic/2706/karten-und-keyword-listen

Kapitel 20: Knoten

Examples

Alle sichtbaren Knoten im System auflisten

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

Knoten auf demselben Rechner verbinden

Starten Sie zwei benannte Knoten in zwei Terminalfenstern:

```
>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)>
```

Verbinden Sie zwei Knoten, indem Sie einen Knoten anweisen, eine Verbindung herzustellen:

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

Die beiden Knoten sind jetzt miteinander verbunden und wissen voneinander:

```
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"]
```

Sie können Code auf anderen Knoten ausführen:

```
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
```

Knoten auf verschiedenen Maschinen verbinden

Starten Sie einen benannten Prozess mit einer IP-Adresse:

```
$ 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"]
```

Starten Sie einen anderen Prozess mit einer anderen IP-Adresse:

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

Knoten online lesen: https://riptutorial.com/de/elixir/topic/2065/knoten

Kapitel 21: Konstanten

Bemerkungen

Dies ist also eine zusammenfassende Analyse, die ich anhand der unter Wie definiere ich Konstanten in Elixir-Modulen aufgelisteten Methoden durchgeführt habe . . Ich poste es aus mehreren Gründen:

- Die meisten Elixir-Dokumentationen sind ziemlich gründlich, aber ich fand diese wichtige architektonische Entscheidung ohne Anleitung - daher hätte ich sie als Thema gefordert.
- Ich wollte ein wenig Sichtbarkeit und Kommentare von anderen zum Thema bekommen.
- Ich wollte auch den neuen Workflow für die SO-Dokumentation testen. ;)

Ich habe auch den gesamten Code in das GitHub-Repo- Elixier-Konstanten-Konzept hochgeladen.

Examples

Konstanten für Module

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

Diese sind nur innerhalb dieses Moduls zugänglich.

Konstanten als Funktionen

Erklären:

```
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
```

Mit konsumieren benötigen:

```
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
```

Bei Import einnehmen:

```
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
```

Mit dieser Methode können Konstanten projektübergreifend wiederverwendet werden. Sie können jedoch nicht in Guard-Funktionen verwendet werden, für die Konstanten zur Kompilierungszeit erforderlich sind.

Konstanten über Makros

Erklären:

```
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
```

Mit `use` verbrauchen:

```
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
```

Mit dieser Methode können Sie die `@some_constant` innerhalb der Guards verwenden. Ich bin mir nicht mal sicher, ob die Funktionen unbedingt notwendig sind.

Konstanten online lesen: https://riptutorial.com/de/elixir/topic/6614/konstanten

Kapitel 22: Listen

Syntax

- []
- [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]
- [Kopf | Schwanz]
- [1 | [2, 3, 4]] # -> [1,2,3,4]
- [1 | [2 | [3 | [4 | []]]]] -> [1,2,3,4]
- 'Hallo' = [? h,? e,? l,? l,? o]
- keyword_list = [a: 123, b: 456, c: 789]
- Schlüsselwortliste [: a] # -> 123

Examples

Keyword-Listen

Schlüsselwortlisten sind Listen, bei denen jedes Element in der Liste ein Tupel eines Atoms ist, gefolgt von einem Wert.

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

Eine Kurzschreibweise zum Schreiben von Schlüsselwortlisten lautet wie folgt:

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

Schlüsselwortlisten sind zum Erstellen von geordneten Datenpaaren mit Schlüsselwertpaaren nützlich, in denen mehrere Elemente für einen bestimmten Schlüssel vorhanden sein können.

Das erste Element in einer Keyword-Liste für einen bestimmten Schlüssel kann wie folgt abgerufen werden:

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

Ein Anwendungsfall für Keyword-Listen kann eine Folge von benannten Aufgaben sein, die ausgeführt werden sollen:

```
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
```

Dieser Code kann mit einer Keyword-Liste aufgerufen werden:

```
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"...
```

Char-Listen

Zeichenketten in Elixir sind "binaries". Im Erlang-Code sind Zeichenfolgen jedoch traditionell "Zeichenlisten". Wenn Sie Erlang-Funktionen aufrufen, müssen Sie möglicherweise Zeichenlisten anstelle von regulären Elixir-Zeichenfolgen verwenden.

Während regelmäßige Strings geschrieben werden mit doppelten Anführungszeichen `"` werden char - Listen mit einfachen Anführungszeichen geschrieben `'` :

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

Zeichenlisten sind einfach Listen von ganzen Zahlen, die die Codepunkte jedes Zeichens darstellen.

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

Eine Zeichenfolge kann mit `to_charlist/1` in eine Char-Liste `to_charlist/1` :

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

Und umgekehrt kann mit `to_string/1` :

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

Aufrufen einer Erlang-Funktion und Konvertieren der Ausgabe in eine reguläre Elixir-Zeichenfolge:

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

Cons Cells

Listen in Elixir sind verknüpfte Listen. Dies bedeutet, dass jedes Element in einer Liste aus einem Wert besteht, gefolgt von einem Zeiger auf das nächste Element in der Liste. Dies wird in Elixir unter Verwendung von cons-Zellen implementiert.

Cons-Zellen sind einfache Datenstrukturen mit einem "linken" und einem "rechten" Wert oder einem "Kopf" und einem "Schwanz".

A `|` Ein Symbol kann vor dem letzten Eintrag in einer Liste hinzugefügt werden, um eine (unangemessene) Liste mit einem bestimmten Kopf und einem bestimmten Ende zu notieren. Das Folgende ist eine einzelne Cons-Zelle mit `1` als Kopf und `2` als Schwanz:

```
[1 | 2]
```

Die Standardsyntax von Elixir für eine Liste entspricht tatsächlich dem Schreiben einer Kette verschachtelter Konsumzellen:

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

Die leere Liste `[]` wird als Ende einer Cons-Zelle verwendet, um das Ende einer Liste darzustellen.

Alle Listen in Elixir entsprechen der Form `[head | tail]` , wobei `head` der erste Eintrag der Liste ist und `tail` der Rest der Liste, abzüglich des Heads.

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

Verwendung des `[head | tail]` -Notation ist nützlich für die Mustererkennung in rekursiven Funktionen:

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

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

Zuordnungslisten

`map` ist eine Funktion in der Funktionsprogrammierung, die bei einer Liste und einer Funktion eine neue Liste mit der auf jedes Element dieser Liste angewendeten Funktion zurückgibt. In Elixir befindet sich die `map/2` Funktion im Enum- Modul.

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

Verwenden der alternativen Erfassungssyntax für anonyme Funktionen:

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

Verweis auf eine Funktion mit Erfassungssyntax:

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

Verkettungslistenoperationen mit dem Pipe-Operator:

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

Listenverständnisse

Elixir hat keine Schleifen. Anstatt sie für Listen gibt es große `Enum` und `List` Module, aber es gibt auch Listenkomprehension.

Listenkomplexe können nützlich sein für:

- Neue Listen erstellen

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

- Filtern von Listen mit `guard` Ausdrücken, aber Sie verwenden sie ohne das Schlüsselwort `when` .

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

- Erstellen von benutzerdefinierten Karte, mit `into` Stichworten:

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

Kombiniertes Beispiel

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

Zusammenfassung

Listenverständnisse:

- verwendet `for..do` Syntax mit zusätzlichen Guards nach Kommas und `into` Keyword, wenn andere Strukturen als Listen zurückgegeben werden, z. Karte.
- In anderen Fällen werden neue Listen zurückgegeben
- unterstützt keine akkus
- Die Verarbeitung kann nicht abgebrochen werden, wenn eine bestimmte Bedingung erfüllt ist
- `guard` Aussagen sein müssen zuerst , um nach `for` und vor `do` oder `into` Symbolen. Die Reihenfolge der Symbole spielt keine Rolle

Entsprechend diesen Einschränkungen sind List Comprehensions nur für die einfache Verwendung begrenzt. In fortgeschrittenen Fällen wäre die Verwendung von Funktionen aus den Modulen `Enum` und `List` die beste Idee.

Unterschied auflisten

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

`--` entfernt das erste Vorkommen eines Elements in der linken Liste für jedes Element auf der rechten Seite.

Mitgliedschaft auflisten

Verwenden Sie `in` Operator, um zu prüfen, ob ein Element Mitglied einer Liste ist.

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

Konvertieren von Listen in eine Karte

Verwenden Sie `Enum.chunk/2` , um Elemente in Unterlisten zu gruppieren, und `Map.new/2` , um es in eine Map zu konvertieren:

```
[1, 2, 3, 4, 5, 6]
|> Enum.chunk(2)
```

```
|> Map.new(fn [k, v] -> {k, v} end)
```

Würde geben:

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

Listen online lesen: https://riptutorial.com/de/elixir/topic/1279/listen

Kapitel 23: Metaprogrammierung

Examples

Tests zur Kompilierzeit generieren

```
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
```

Ausgabe:

```
.

  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
```

Metaprogrammierung online lesen: https://riptutorial.com/de/elixir/topic/4069/metaprogrammierung

Kapitel 24: Mischen

Examples

Erstellen Sie eine benutzerdefinierte Mix-Aufgabe

```
# 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
```

Kompilieren und ausführen:

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

Benutzerdefinierte Mischaufgabe mit Befehlszeilenargumenten

In einer Basisimplementierung muss das Taskmodul eine `run/1` Funktion definieren, die eine Liste von Argumenten verwendet. ZB `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
```

Kompilieren und ausführen:

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

Aliase

Mit Elixir können Sie Aliase für Ihre Mischbefehle hinzufügen. Coole Sache, wenn Sie sich etwas Tipparbeit sparen wollen.

Öffnen Sie `mix.exs` in Ihrem Elixir-Projekt.

`aliases/0` zunächst der Keyword-Liste, die die `project` zurückgibt, eine `aliases/0` Funktion hinzu.

Durch das Hinzufügen von `()` am Ende der Aliase-Funktion wird verhindert, dass der Compiler eine Warnung auslöst.

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

Definieren Sie dann Ihre `aliases/0` Funktion (z. B. am Ende Ihrer `mix.exs` Datei).

```
...

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

Sie können jetzt `$ mix go` , um Ihren Phoenix-Server auszuführen (wenn Sie eine Phoenix-Anwendung ausführen). Verwenden Sie `$ mix trident` , um mix mitzuteilen, dass alle Abhängigkeiten abgerufen, der Server kompiliert und ausgeführt wird.

Hilfe zu verfügbaren Mix-Aufgaben erhalten

Um verfügbare Mix-Aufgaben aufzulisten, verwenden Sie:

```
mix help
```

Um Hilfe zu einer bestimmten Aufgabe zu erhalten, verwenden Sie die `mix help task` z.

```
mix help cmd
```

Mischen online lesen: https://riptutorial.com/de/elixir/topic/3585/mischen

Kapitel 25: Module

Bemerkungen

Modulnamen

In Elixir sind `:"Elixir.ModuleName"` wie `IO` oder `String` nur Atome unter der Haube und werden zur Kompilierungszeit in die Form `:"Elixir.ModuleName"` konvertiert.

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

Examples

Listen Sie die Funktionen oder Makros eines Moduls auf

Die Funktion `__info__/1` hat eines der folgenden Atome:

- `:functions` - Gibt eine Liste der Schlüsselwörter öffentlicher Funktionen mit ihren Aritäten zurück
- `:macros` - Gibt eine Liste der Schlüsselwörter öffentlicher Makros mit ihren Eigenschaften zurück

So listen Sie die Funktionen des `Kernel` Moduls auf:

```
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, ...]
```

Ersetzen Sie den `Kernel` durch ein beliebiges Modul Ihrer Wahl.

Verwendung von Modulen

Für Module stehen vier Schlüsselwörter zur Verfügung, um sie in anderen Modulen verwenden zu können: `alias` , `import` , `use` und `require` .

`alias` registriert ein Modul unter einem anderen (normalerweise kürzeren) Namen:

```
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` werden alle Funktionen des Moduls ohne Namen vor ihnen zur Verfügung gestellt:

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

`use` kann ein Modul Code in das aktuelle Modul einfügen. Dies geschieht normalerweise als Teil eines Frameworks, das seine eigenen Funktionen erstellt, um Ihr Modul dazu zu bringen, ein bestimmtes Verhalten zu bestätigen.

`require` Lademakros vom Modul, damit sie verwendet werden können.

Delegieren von Funktionen an ein anderes Modul

Verwenden Sie `defdelegate` , um Funktionen zu definieren, die an Funktionen mit demselben Namen delegieren, die in einem anderen Modul definiert sind:

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


iex> Math.pi
3.141592653589793
```

Module online lesen: https://riptutorial.com/de/elixir/topic/2721/module

Kapitel 26: Musterabgleich

Examples

Pattern-Matching-Funktionen

```
#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
```

Musterabgleich auf einer Karte

```
%{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"}
```

Mustervergleich auf einer Liste

Sie können auch Muster für Elixir-Datenstrukturen (z. B. Listen) zuordnen.

Listen

Das Übereinstimmen auf einer Liste ist ziemlich einfach.

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

Dies funktioniert, indem die ersten (oder mehr) Elemente in der Liste auf der linken Seite von `|` (Pipe) und der Rest der Liste auf der rechten Seite der Variable `|` .

Wir können auch bestimmte Werte einer Liste abgleichen:

```
[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]
```

Binden mehrerer aufeinanderfolgender Werte links von `|` ist auch erlaubt:

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

Noch komplexer - wir können einen bestimmten Wert abgleichen und dies mit einer Variablen abgleichen:

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

Holen Sie sich die Summe einer Liste mit Hilfe des Mustervergleichs

```
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
```

Anonyme Funktionen

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

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

Tuples

```
{ 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" }
```

Eine Datei lesen

Der Musterabgleich ist nützlich für eine Operation wie das Lesen einer Datei, die ein Tupel zurückgibt.

Wenn die Datei `sample.txt` enthält, `This is a sample text` :

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

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

Andernfalls, wenn die Datei nicht vorhanden ist:

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

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

Musteranpassung anonymer Funktionen

```
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
```

Musterabgleich online lesen: https://riptutorial.com/de/elixir/topic/1602/musterabgleich

Kapitel 27: Operatoren

Examples

Der Pipe Operator

Der Pipe-Operator `|>` nimmt das Ergebnis eines Ausdrucks auf der linken Seite und führt ihn als ersten Parameter einer Funktion auf der rechten Seite zu.

```
expression |> function
```

Verwenden Sie den Pipe Operator, um Ausdrücke miteinander zu verketten und den Fluss einer Reihe von Funktionen visuell zu dokumentieren.

Folgendes berücksichtigen:

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

In dem Beispiel steht `Oven.bake` vor `Ingredients.mix` , wird jedoch zuletzt ausgeführt. Es kann auch nicht offensichtlich sein, dass `:temperature` ein Parameter von `Oven.bake`

Dieses Beispiel mit dem Pipe Operator neu schreiben:

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

gibt das gleiche Ergebnis, aber die Reihenfolge der Ausführung ist klarer. Außerdem ist klar `:temperature` ist ein Parameter für den Aufruf von `Oven.bake` .

Beachten Sie, dass bei Verwendung des Pipe-Operators der erste Parameter für jede Funktion vor dem Pipe-Operator verschoben wird, und die aufgerufene Funktion scheint einen Parameter weniger zu haben. Zum Beispiel:

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

ist das gleiche wie:

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

Rohrbediener und Klammern

Klammern sind erforderlich, um Mehrdeutigkeiten zu vermeiden:

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

Sollte geschrieben werden als:

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

boolesche Operatoren

Es gibt zwei Arten von booleschen Operatoren in Elixir:

- boolesche Operatoren (sie erwarten als erstes Argument entweder `true` oder `false`)

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

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

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

Alle booleschen Operatoren lösen `ArgumentError` wenn das erste Argument kein strikt boolescher Wert ist, dh nur `true` oder `false` (`nil` ist nicht boolean).

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

- Entspannte boolesche Operatoren (arbeiten mit jedem Typ, alles, was weder `false` noch `nil` ist, wird als `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
```

Betreiber `||` wird immer das erste Argument zurückgeben, wenn es wahr ist (Elixir behandelt alles außer `nil` und `false` , um bei Vergleichen wahr zu sein), andernfalls wird das zweite zurückgegeben.

```
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 `&&` gibt immer das zweite Argument zurück, wenn es wahr ist. Andernfalls kehren die Argumente `false` bzw. `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
```

Sowohl `&&` als auch `||` sind Kurzschlussoperatoren. Sie führen die rechte Seite nur dann aus, wenn die linke Seite nicht ausreicht, um das Ergebnis zu bestimmen.

Betreiber `!` gibt den booleschen Wert der Negation des aktuellen Begriffs zurück:

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

Ein einfacher Weg, um den booleschen Wert des ausgewählten Begriffs zu erhalten, besteht darin, diesen Operator einfach zu verdoppeln:

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

Vergleichsoperatoren

Gleichberechtigung:

- Wertgleichheit `x == y` (`1 == 1.0 # true`)
- Wertungleichheit `x == y` (`1 != 1.0 # false`)
- strikte Gleichheit `x === y` (`1 === 1.0 # false`)
- strikte Ungleichung `x === y` (`1 !== 1.0 # true`)

Vergleich:

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

Wenn Typen kompatibel sind, verwendet der Vergleich eine natürliche Reihenfolge. Ansonsten gibt es eine allgemeine Typenvergleichsregel:

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

Operator beitreten

Sie können Binärdateien (einschließlich Zeichenfolgen) und Listen verknüpfen (verketten):

```
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"
```

'In' Operator

`in` Operator können Sie prüfen, ob eine Liste oder ein Bereich ein Element enthält:

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

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

Operatoren online lesen: https://riptutorial.com/de/elixir/topic/1161/operatoren

Kapitel 28: Optimierung

Examples

Messen Sie immer zuerst!

Dies sind allgemeine Tipps, die im Allgemeinen die Leistung verbessern. Wenn Ihr Code langsam ist, ist es immer wichtig, ein Profil zu erstellen, um herauszufinden, welche Teile langsam sind. Raten ist **nie** genug. Die Verbesserung der Ausführungsgeschwindigkeit von etwas, das nur 1% der Ausführungszeit in Anspruch nimmt, ist den Aufwand wahrscheinlich nicht wert. Suchen Sie nach den großen Senken.

Um etwas genaue Zahlen zu erhalten, stellen Sie sicher, dass der zu optimierende Code beim Profiling mindestens eine Sekunde lang ausgeführt wird. Wenn Sie 10% der Ausführungszeit in dieser Funktion verbringen, stellen Sie sicher, dass die vollständige Programmausführung mindestens 10 Sekunden dauert, und stellen Sie sicher, dass Sie dieselben exakten Daten mehrmals durch den Code laufen lassen, um wiederholbare Zahlen zu erhalten.

ExProf ist einfach zu beginnen.

Optimierung online lesen: https://riptutorial.com/de/elixir/topic/6062/optimierung

Kapitel 29: Polymorphismus in Elixier

Einführung

Polymorphismus ist die Bereitstellung einer einzigen Schnittstelle für Entitäten verschiedener Typen. Grundsätzlich können verschiedene Datentypen auf dieselbe Funktion reagieren. Dieselbe Funktion ist also für verschiedene Datentypen geeignet, um dasselbe Verhalten zu erreichen. Elixiersprache verfügt über `protocols` , um Polymorphismus auf saubere Weise zu implementieren.

Bemerkungen

Wenn Sie alle Datentypen abdecken möchten, können Sie eine Implementierung für den Datentyp `Any` definieren. Wenn Sie Zeit haben, überprüfen Sie den Quellcode von Enum und String.Char. Dies sind gute Beispiele für Polymorphismen in Core-Elixir.

Examples

Polymorphismus mit Protokollen

Implementieren wir ein Basisprotokoll, das die Temperaturen in Kelvin und Fahrenheit in Celsius umwandelt.

```
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
```

Jetzt haben wir unsere Konverter für die Typen Kelvin und Fahrenheit implementiert. Lassen Sie uns einige Konvertierungen vornehmen:

```
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"}
```

Versuchen wir, einen anderen Datentyp zu konvertieren, für den die `to_celsius` Funktion nicht implementiert ist:

```
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
```

Polymorphismus in Elixier online lesen: https://riptutorial.com/de/elixir/topic/9519/polymorphismus-in-elixier

Kapitel 30: Protokolle

Bemerkungen

Ein Hinweis zu structs

Anstatt die Protokollimplementierung mit Karten gemeinsam zu nutzen, erfordern structs eine eigene Protokollimplementierung.

Examples

Einführung

Protokolle ermöglichen Polymorphismus in Elixir. Protokolle mit `defprotocol` :

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

Implementiere ein Protokoll mit `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
```

Mit den obigen Implementierungen können wir Folgendes tun:

```
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
```

Protokolle ermöglichen das Versenden an einen beliebigen Datentyp, sofern das Protokoll implementiert wird. Dies umfasst einige integrierte Typen wie `Atom` , `BitString` , `Tuples` und andere.

Protokolle online lesen: https://riptutorial.com/de/elixir/topic/3487/protokolle

Kapitel 31: Prozesse

Examples

Einen einfachen Prozess starten

Im folgenden Beispiel wird die `greet` Funktion im `Greeter` Modul in einem separaten Prozess ausgeführt:

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

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

Hier ist `#PID<0.122.0>` die *Prozesskennung* für den erzeugten Prozess.

Nachrichten senden und empfangen

```
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}
```

Rekursion und Empfang

Rekursion kann verwendet werden, um mehrere Nachrichten zu empfangen

```
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 verwendet eine Rekursionsoptimierung für den Rückruf, solange der Funktionsaufruf das letzte ist, was in der Funktion wie im Beispiel geschieht.

Prozesse online lesen: https://riptutorial.com/de/elixir/topic/3173/prozesse

Kapitel 32: Sigils

Examples

Erstellen Sie eine Liste mit Zeichenfolgen

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

Erstellen Sie eine Liste von Atomen

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

Kundenspezifische Siegel

Benutzerdefinierte Siegel können durch Erstellen einer Methode `sigil_X` wobei X der Buchstabe ist, den Sie verwenden möchten (dies kann nur ein einzelner Buchstabe sein).

```
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
```

Das `options` ist eine binäre der am Ende des Sigils angegebenen Argumente, zum Beispiel:

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

Sigils online lesen: https://riptutorial.com/de/elixir/topic/2204/sigils

Kapitel 33: STRAHL

Examples

Einführung

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

`:observer.start` öffnet die GUI-Observer-Oberfläche und zeigt Ihnen CPU-Ausfall, Speicherauslastung und andere Informationen an, die für das Verständnis der Nutzungsmuster Ihrer Anwendungen wichtig sind.

STRAHL online lesen: https://riptutorial.com/de/elixir/topic/3587/strahl

Kapitel 34: Strings verbinden

Examples

String-Interpolation verwenden

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

E / A-Liste verwenden

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

Dies führt zu einigen Leistungssteigerungen, da Zeichenfolgen nicht im Speicher vorhanden sind. Alternative Methode:

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

Enum.join verwenden

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

Strings verbinden online lesen: https://riptutorial.com/de/elixir/topic/9202/strings-verbinden

Kapitel 35: Strom

Bemerkungen

Streams sind komponierbare, faule Enumerables.

Streams sind aufgrund ihrer Faulheit nützlich, wenn Sie mit großen (oder sogar unendlichen) Sammlungen arbeiten. Wenn Sie viele Vorgänge mit `Enum` verketten, werden Zwischenlisten erstellt, während `Stream` ein Rezept von Berechnungen erstellt, die zu einem späteren Zeitpunkt ausgeführt werden.

Examples

Verketten mehrerer Operationen

`Stream` ist besonders nützlich, wenn Sie mehrere Vorgänge für eine Sammlung ausführen möchten. Dies liegt daran, dass `Stream` faul ist und nur eine Iteration durchführt (während `Enum` beispielsweise mehrere Iterationen `Enum` würde).

```
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]
```

Hier haben wir 3 Operationen verkettet (`map` , `filter` und `take_every`), aber die letzte Iteration wurde erst ausgeführt, nachdem `Enum.to_list` aufgerufen wurde.

Was `Stream` intern tut, ist, dass er wartet, bis eine tatsächliche Auswertung erforderlich ist. Vorher erstellt es eine Liste aller Funktionen. Sobald eine Auswertung erforderlich ist, wird sie einmal durch die Sammlung geführt und führt alle Funktionen für jedes Element aus. Dies macht es effizienter als `Enum` , das in diesem Fall beispielsweise 3 Iterationen durchführen würde.

Strom online lesen: https://riptutorial.com/de/elixir/topic/2553/strom

Kapitel 36: Tipps und Tricks

Einführung

Elixir Fortgeschrittene Tipps und Tricks, die beim Codieren Zeit sparen.

Examples

Benutzerdefinierte Siegel erstellen und dokumentieren

Jedes x-Sigel ruft die jeweilige sigil_x-Definition auf

Benutzerdefinierte Siegel definieren

```
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
```

Mehrere [ODER]

Dies ist nur die andere Art, mehrere ODER-Bedingungen zu schreiben. Dies ist nicht der empfohlene Ansatz, da die Bedingung bei regelmäßiger Annäherung, wenn sie als wahr ausgewertet wird, die Ausführung der verbleibenden Bedingungen beendet, die die Bewertungszeit sparen, im Gegensatz zu diesem Ansatz, bei dem alle Bedingungen, die an erster Stelle in der Liste stehen, ausgewertet werden. Das ist einfach nur schlecht, aber gut für Entdeckungen.

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

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

iex Benutzerdefinierte Konfiguration - iex Dekoration

Kopieren Sie den Inhalt in eine Datei und speichern Sie die Datei unter .iex.exs in Ihrem ~ Home-Verzeichnis, um die Magie zu sehen. Sie können die Datei auch HIER herunterladen

```
# 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

)
```

Tipps und Tricks online lesen: https://riptutorial.com/de/elixir/topic/10623/tipps-und-tricks

Kapitel 37: Tipps und Tricks für die IEx-Konsole

Examples

Rekompilieren Sie das Projekt mit "Rekompilieren"

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

Siehe Dokumentation mit "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
```

Holen Sie sich den Wert vom letzten Befehl mit `v`

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

Siehe auch: Holen Sie sich den Wert einer Zeile mit `v`

Holen Sie sich den Wert eines vorherigen Befehls mit `v`

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

Sie können eine bestimmte Zeile über den Index der Zeile übergeben:

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

Sie können auch einen Index relativ zur aktuellen Zeile angeben:

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

Der Wert kann in anderen Berechnungen wiederverwendet werden:

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

Wenn Sie eine nicht vorhandene Zeile angeben, gibt `IEx` einen Fehler aus:

```
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
```

Beenden Sie die IEx-Konsole

1. Verwenden Sie zum Verlassen Strg + C, Strg + C

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

2. Verwenden Sie `Ctrl+ \` um sofort zu beenden

Siehe Information mit `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
```

PID erstellen

Dies ist nützlich, wenn Sie die PID eines vorherigen Befehls nicht gespeichert haben

```
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>
```

Halten Sie Ihre Aliase bereit, wenn Sie IEx starten

Wenn Sie Ihre häufig verwendeten Aliase in eine `.iex.exs` Datei im Stammverzeichnis Ihrer App `.iex.exs` IEx diese beim Start für Sie.

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

Anhaltende Geschichte

Standardmäßig wird der Benutzereingabeverlauf in `IEx` nicht für verschiedene Sitzungen `IEx` .

`erlang-history` fügt der Erlang-Shell und dem `IEx` `erlang-history` hinzu:

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

Sie können jetzt mit den Aufwärts- und `IEx` auf Ihre vorherigen Eingaben zugreifen, auch in verschiedenen `IEx` Sitzungen.

Wenn die Elixir-Konsole feststeckt ...

Manchmal laufen Sie aus Versehen etwas in der Shell, das für immer wartet und die Shell blockiert:

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

In diesem Fall drücken Sie Strg-g. Du wirst sehen:

```
User switch command
```

Geben Sie diese Befehle in der angegebenen Reihenfolge ein:

- `k` (um den Shell-Prozess zu beenden)
- `s` (um einen neuen Shell-Prozess zu starten)
- `c` (um eine Verbindung zum neuen Shell-Prozess herzustellen)

Sie landen in einer neuen Erlang-Shell:

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

Um eine Elixir-Shell zu starten, geben Sie Folgendes ein:

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

(den letzten Punkt nicht vergessen!)

Dann sehen Sie einen neuen Elixir-Shell-Prozess:

```
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)>
```

Um aus dem "Warten auf weitere Eingabe" -Modus (aufgrund nicht geschlossener Anführungszeichen, Klammern usw.) zu **`#iex:break`** , geben Sie **`#iex:break`** , gefolgt von Wagenrücklauf (▯):

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

iex(1)>
```

Dies ist besonders nützlich, wenn ein relativ großes Snippet durch Kopieren und Einfügen in den Modus "Warten auf weitere Eingaben" versetzt wird.

brechen Sie aus unvollständigem Ausdruck aus

Wenn Sie in IEx etwas eingegeben haben, das eine Vervollständigung erwartet, wie z. B. eine mehrzeilige Zeichenfolge, ändert IEx die Eingabeaufforderung, um anzuzeigen, dass Sie auf den Abschluss warten, indem Sie die Eingabeaufforderung so ändern, dass sie eine Auslassungszeichen (`...`) anstelle von `iex` .

Wenn Sie feststellen, dass IEx auf das Beenden eines Ausdrucks wartet, Sie jedoch nicht sicher sind, was zum Beenden des Ausdrucks erforderlich ist, oder möchten Sie diese Eingabezeile einfach abbrechen, geben `#iex:break` als Konsoleneingabe `#iex:break` . Dies bewirkt, dass IEx einen `TokenMissingError` und das Warten auf weitere Eingaben `TokenMissingError` , wodurch Sie zu einer Standardkonsoleingabe der obersten Ebene zurückkehren.

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

Weitere Informationen finden Sie in der IEx-Dokumentation .

Laden Sie ein Modul oder Skript in die IEx-Sitzung

Wenn Sie eine Elixier-Datei haben; Wenn Sie ein Skript oder ein Modul verwenden und es in die aktuelle IEx-Sitzung laden möchten, können Sie die Methode `c/1` verwenden:

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

Dadurch wird das Modul in IEx kompiliert und geladen, und Sie können alle öffentlichen Methoden aufrufen.

Bei Skripten wird der Inhalt des Skripts sofort ausgeführt:

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

Tipps und Tricks für die IEx-Konsole online lesen: https://riptutorial.com/de/elixir/topic/1283/tipps-und-tricks-fur-die-iex-konsole

Kapitel 38: Tipps zum Debuggen

Examples

Debuggen mit IEX.pry / 0

Das Debuggen mit `IEx.pry/0` ist recht einfach.

1. `require IEx` in Ihrem Modul
2. Suchen Sie die Codezeile, die Sie untersuchen möchten
3. Fügen `IEx.pry` nach der Zeile `IEx.pry` hinzu

Starten Sie nun Ihr Projekt (zB `iex -S mix`).

Wenn die Zeile mit `IEx.pry/0` erreicht ist, wird das Programm `IEx.pry/0` und Sie haben die Möglichkeit zu prüfen. Es ist wie ein Haltepunkt in einem traditionellen Debugger.

Wenn Sie fertig sind, tippen Sie einfach `respawn` in die Konsole.

```
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
```

Debuggen mit IO.inspect / 1

Es ist möglich, IO.inspect / 1 als Werkzeug zum Debuggen eines Elixierprogramms zu verwenden.

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

Es werden Argumente_1 und Argumente_2 an die Konsole ausgegeben. Da `IO.inspect/1` sein Argument zurückgibt, ist es sehr einfach, es in Funktionsaufrufe oder Pipelines aufzunehmen, ohne den Fluss zu `IO.inspect/1` :

```
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))
```

In Rohrleitung debuggen

```
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]
```

In Pfeife hebeln

```
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)>
```

Tipps zum Debuggen online lesen: https://riptutorial.com/de/elixir/topic/2719/tipps-zum-debuggen

Kapitel 39: Verhaltensweisen

Examples

Einführung

Verhalten sind eine Liste von Funktionsspezifikationen, die ein anderes Modul implementieren kann. Sie ähneln den Schnittstellen in anderen Sprachen.

Hier ist ein Beispiel für ein Verhalten:

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

Und ein Modul, das es implementiert:

```
defmodule JSONParser do
  @behaviour Parser

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

Das `@behaviour` Attribut-Attribut zeigt an, dass dieses Modul jede im Parser-Modul definierte Funktion definieren soll. Fehlende Funktionen führen zu undefinierten Verhaltensfunktionskompilierungsfehlern.

Module können mehrere `@behaviour` Attribute haben.

Verhaltensweisen online lesen: https://riptutorial.com/de/elixir/topic/3558/verhaltensweisen

Kapitel 40: Zeichenketten

Bemerkungen

Ein `String` in Elixir ist eine `UTF-8` codierte Binärdatei.

Examples

Konvertieren Sie in einen String

Verwenden Sie Kernel.inspect, um alles in eine Zeichenfolge zu konvertieren.

```
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\"}"
```

Holen Sie sich einen Teilstring

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

Einen String teilen

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

String Interpolation

```
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."
```

Prüfen Sie, ob String Substrate enthält

```
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
```

Strings verbinden

Sie können Zeichenfolgen in Elixir mithilfe des Operators `<>` verketten:

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

Für eine `List` von Strings können Sie `Enum.join/2` :

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

Zeichenketten online lesen: https://riptutorial.com/de/elixir/topic/2618/zeichenketten

Kapitel 41: Zustandsbehandlung in Elixir

Examples

Verwalten eines Zustands mit einem Agenten

Der einfachste Weg, einen Status zu umschließen und auf ihn zuzugreifen, ist `Agent` . Mit dem Modul kann ein Prozess erzeugt werden, der eine beliebige Datenstruktur enthält und Nachrichten zum Lesen und Aktualisieren dieser Struktur sendet. Dadurch wird der Zugriff auf die Struktur automatisch serialisiert, da der Prozess jeweils nur eine Nachricht verarbeitet.

```
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}
```

Zustandsbehandlung in Elixir online lesen:
https://riptutorial.com/de/elixir/topic/6596/zustandsbehandlung-in-elixir

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit 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	Aufgabe	mario
3	Basic .gitignore für das Elixierprogramm	Yos Riady
4	Besseres Debuggen mit IO.inspect und Labels	leifg
5	Conditionals	Andrey Chernykh, evuez, javanut13, Musfiqur Rahman, Paweł Obrok
6	Datenstrukturen	Sam Mercier, Simone Carletti, Stephan Rodemeier, Yos Riady
7	Doktests	aholt, milmazz, Philippe-Arnaud de MANGOU, Yos Riady
8	Ecto	fgutierr, Philippe-Arnaud de MANGOU, toraritte
9	Eingebaute Typen	Andrey Chernykh, Arithmeticbird, Oskar, TreyE, Vinicius Quaiato
10	Erlang	4444, Yos Riady
11	ExDoc	milmazz, Yos Riady
12	ExUnit	Yos Riady
13	Funktionale Programmierung in Elixir	Dinesh Balasubramanian
14	Funktionen	Andrey Chernykh, cwc, Dair, Eiji, Filip Haglund, PatNowak, rainteller, Simone Carletti, Stephan Rodemeier, Yedhu Krishnan , Yos Riady
15	grundlegende Verwendung von Schutzklauseln	alxndr

16	Hilfe in der IEx-Konsole erhalten	helcim
17	Installation	cwc, Douglas Correa, Eiji, JAlberto, MattW.
18	Karten und Keyword-Listen	Sam Mercier, Simone Carletti, Yos Riady
19	Knoten	Yos Riady
20	Konstanten	ibgib
21	Listen	Ben Bals, Candy Gumdrop, emoragaf, PatNowak, Sheharyar, Yos Riady
22	Metaprogrammierung	4444, Paweł Obrok
23	Mischen	4444, helcim, rainteller, Slava.K, Yos Riady
24	Module	Alex G, javanut13, Yos Riady
25	Musterabgleich	Alex Anderson, Dair, Danny Rosenblatt, evuez, Gabriel C, gmile , Harrison Lucas, javanut13, Oskar, PatNowak, theIV, Thomas, Yedhu Krishnan
26	Operatoren	alxndr, Andrey Chernykh, Dair, Gazler, Mitkins, nirev, PatNowak
27	Optimierung	Filip Haglund, legoscia
28	Polymorphismus in Elixier	mustafaturan
29	Protokolle	Yos Riady
30	Prozesse	Alex G, Yedhu Krishnan
31	Sigils	javanut13, Yos Riady
32	STRAHL	Yos Riady
33	Strings verbinden	Agung Santoso
34	Strom	Oskar
35	Tipps und Tricks	Ankanna
36	Tipps und Tricks für die IEx-Konsole	alxndr, Cifer, fahrradflucht, legoscia, mudasobwa, muttonlamb, PatNowak, Paweł Obrok, sbs, Sheharyar, Simone Carletti, Stephan Rodemeier, Uniaika, Vincent, Yos Riady
37	Tipps zum Debuggen	javanut13, Paweł Obrok, Pfitz, Philippe-Arnaud de MANGOU,

		sbs
38	Verhaltensweisen	Yos Riady
39	Zeichenketten	Alex G, Sheharyar, Yos Riady
40	Zustandsbehandlung in Elixir	Paweł Obrok