無料電子ブック

学習

Elixir Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#elixir

1: Elixirをいめる

Elixirはスケーラブルでなアプリケーションをするためにされたでなです。

Elixirは、レイテンシのおよびフォールトトレラントシステムですることでられているErlang VMをしており、Webやみみソフトウェアでもうまくされています。

バージョン

バージョン	
0.9	2013-05-23
1.0	2014-09-18
1.1	2015-09-28
1.2	2016-01-03
1.3	2016621
1.4	2017-01-05

Examples

こんにちは

エリキシルチェックのインストールについては、ここでは、なるプラットフォームにしたをします。

Elixirは`erlang`をしてされたプログラミングで、erlangの`BEAM`ランタイムJavaの`JVM`などをします。

たちはエリクシールを2つのモードでうことができますシェル`iex`か、 `elixir`コマンドをってします。

`hello.exs`というのファイルにのファイルをします。

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

コマンドラインで、のコマンドをしてElixirソースファイルをします。

```
$ elixir hello.exs
```

これはするがあります

こんにちは

これは、 `Elixir`のスクリプトモードとばれます 。、Elixirプログラムは、BEAMマシンのバイトコードにコンパイルすることもできますには、それらのプログラムをコンパイルすることもできます。

あなたはまた、インタラクティブなエリクシールシェルに`iex`をうこともできます。コマンドをすると、のようなプロンプトがされます

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

ここであなたのエリクサーの`hello world`をすことができます

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

`iex`をしてモジュールをコンパイルしてすることもできます。たとえば、 `helloworld.ex`ものがまれているとします。

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

`iex`をじて、

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

IExからのHello World

`IEx` Interactive Elixirシエルをして、をしてコードをすることもできます。

LinuxまたはMacのは、bashにiexとしてEnterキーをします。

```
$ iex
```

Windowsマシンのは、のようにします。

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

に、IEx REPLみみ、、、ループをし、のようにするだけです。

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

IEx REPLをいているときにスクリプトをロードするには、のようにします。

```
$ iex script.exs
```

えられた`script.exs`はあなたのスクリプトです。これでコンソールのスクリプトからをびすことができます。

オンラインでElixirをいめるをむ https://riptutorial.com/ja/elixir/topic/954/elixirをいめる

2: Doctests

Examples

き

`@doc`でコードをするときは、のようなコードをできます。

```
# 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
```

テストケースとしてコードをテストスイートのいずれかにすることができます。

```
# myproject/test/doc_test.exs

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

その、 `mix test`テストをできます。

doctestについたHTMLの

ドキュメントのはマークダウンについているため、2つのことをうがあります。

1 /あなたのdoctestをいて、をさせるためのdoctestのをにしてください "examples"や "tests"のようなしをけるがい。テストをくときには、HTMLドキュメントのコードとしてされるように、テストコードに4つのスペースをけることをれないでください。

2 /に、あなたのエリクシールプロジェクトのルートにあるコンソールに "mix docs"として、あなたのエリクシールプロジェクトのルートにあるdocディレクトリにHTMLドキュメントをします。

```
$> mix docs
```

のdoctest

のにくには '...>'をってのdoctestをできます

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

オンラインでDoctestsをむ https://riptutorial.com/ja/elixir/topic/2708/doctests

3: ExDoc

Examples

き

でドキュメントをするため`HTML`からフォーマット`@doc`と`@moduledoc`ソースコードの、`ex_doc`とげプロセッサを、ExDocはサポートしていてる 、 Pandoc 、 ホーダウンとCmarkをあなたへのとして 、 `mix.exs`ファイル

```
# config/mix.exs

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

のMarkdownプロセッサをするは、 「Markdownツールの」のセクションをしてください。

Elixir `@doc`と`@moduledoc`でMarkdownをうことができます。

に、 `mix docs`します。

ExDocでは、のようなパラメータをできます。

```
  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
```

このオプションにするは、 `mix help docs`をしてください

オンラインでExDocをむ https://riptutorial.com/ja/elixir/topic/3582/exdoc

4: ExUnit

Examples

の

assert_raiseをして、がしたかどうかをテストします。 assert_raiseはExceptionとされるを assert_raiseます。

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

テストするコードをにラップし、 assert_raiseます。

オンラインでExUnitをむ https://riptutorial.com/ja/elixir/topic/3583/exunit

5: IExコンソールでのヘルプの

き

IExはElixirのドキュメントにアクセスします。 Elixirがシステムにインストールされているは、`iex`コマンドをですることができます。に、IExコマンドラインで`h`コマンドをし、そのモジュールのにをけてします。たとえば、 `h List.foldr`

Examples

Elixirモジュールとのリスト

Elixirモジュールのリストをするには、

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

[TAB]をすと、モジュールとがします。この、すべてのモジュールがリストされます。モジュールのすべてのをつけるには`List` using

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

オンラインでIExコンソールでのヘルプのをむ https://riptutorial.com/ja/elixir/topic/10780/iexコンソールでのヘルプの

6: IExコンソールのヒントとコツ

Examples

`コンパイル`をってプロジェクトをコンパイルする

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

`h`のドキュメントを

```
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
```

のコマンドのを`v`でる

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

`v`でのをする

のコマンドのを`v`でする

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

のインデックスをすのをすることができます

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

のになインデックスをすることもできます。

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

はのでできます。

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

しないをすると、 `IEx`はエラーをさせます。

```
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
```

IExコンソールをする

1. するにはCtrl + C、Ctrl + Cをいます

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

2. `Ctrl+ \`を`Ctrl+ \`てすぐにする

「i」のを

```
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の

これはのコマンドからPIDをしていないときにです

```
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>
```

IExのにエイリアスをする

よくうエイリアスをアプリのルートにある`.iex.exs`ファイルにれると、に`.iex.exs`がそれらをみみます。

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

な

デフォルトでは、 `IEx`ユーザーはなるセッションでされません。

`erlang-history`はErlangシエルと`IEx`にサポートをします

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

`IEx`なるセッションでも、のキーをしてのにアクセスできるようになりました。

エリクシールのコンソールがついているとき...

によっては、ってシエルでかをして、にってシエルをブロックすることもあります。

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

そのは、Ctrl + gをします。わかるでしょ

```
User switch command
```

のコマンドをにします。

- `k` シエルプロセスをする
- `s` しいシエルプロセスをする
- `c` しいシエルプロセスにするため

あなたはしいErlangシエルでわるでしょう

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

Elixirシエルをするには、のようにします。

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

のドットをれないでください

そして、しいElixirシエルプロセスがするでしょう

```
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)>
```

でまれていないかっこなどのために「ち」モーードからするには、 **`#iex:break`** 、そのにキャリッジリターーン␍ をします。

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

iex(1)>
```

は、きなスニペットをコピーペーストしてコンソールを「ち」モーードにするときににです。

なからけす

のなどのがな`iex`すると、 `iex`では、プロンプトを`iex`ではなく `...` にしてをっていることをすようにされます。

をするのをっているが、をするがあるかどうかわからないや、このをしたいは、コンソールとして`#iex:break`とします。これにより、 `TokenMissingError`は`TokenMissingError`をスローし、それのをってキャンセルし、の「」のコンソールにります。

```
iex:1> "foo"
"foo"
iex:2> "bar
...:2> #iex:break
```

```
** (TokenMissingError) iex:2: incomplete expression
```

は、IExのドキュメントをしてください 。

モジュールまたはスクリプトをIExセッションにロードする

あなたがエリクシルファイルをっている。スクリプトまたはモジュールをして、のIExセッションにロードしたいは、 `c/1`メソッドをできます。

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

これにより、モジュールがコンパイルされ、IExでモジュールがロードされ、すべてのパブリックメソッドをびすことができます。

スクリプトの、スクリプトのはちにされます。

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

オンラインでIExコンソールのヒントとコツをむ https://riptutorial.com/ja/elixir/topic/1283/iexコンソールのヒントとコツ

7: IO.inspectとラベルによるデバッグのき

`IO.inspect`は、メソッドびしのチェーンをデバッグしようとするとにです。あまりにもにすると、になることがあります。

Elixir 1.4.0、 `IO.inspect`の`label`オプションがちます

Elixir 1.4+でのみしますが、まだタグけできません。

Examples

ラベルなし

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

これはコンテキストなしでくのをもたらします

```
"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}
```

ラベルき

`label`オプションをしてコンテキストをすると、くのにちます。

```
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}
```

オンラインでIO.inspectとラベルによるデバッグのをむ
https://riptutorial.com/ja/elixir/topic/8725/io-inspectとラベルによるデバッグの

8: Sigils

Examples

のリストをする

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

のリストをする

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

カスタムサイン

カスタム`sigil_X`は、 `sigil_X`というメソッドをしてすることができます。ここで、Xはするですこれは1のみです。

```
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
```

`options`は、sigilのにえられたのバイナリです。たとえば、のようになります。

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

オンラインでSigilsをむ https://riptutorial.com/ja/elixir/topic/2204/sigils

9: アーラン

Examples

Erlangをう

Erlangモジュールはとしてできます。たとえば、Erlangモジュールはのようにできます:`math`

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

Erlangモジュールをする

したいErlangモジュールで`module_info`をしてください

```
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>>]
```

オンラインでアーランをむ https://riptutorial.com/ja/elixir/topic/2716/アーラン

10: インストール

Examples

Fedoraのインストール

```
dnf install erlang elixir
```

OSXのインストール

OS XとMacOSでは、Elixirはのパッケージマネージャーをってインストールできます

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

Macports

```
 $ sudo port install elixir
```

Debian / Ubuntu インストール

```
# 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のインストール

Elixirはメインのパッケージリポジトリでです。
パッケージをインストールするにパッケージリストをしてください

```
emerge --sync
```

これはワンステップインストールです。

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

オンラインでインストールをむ https://riptutorial.com/ja/elixir/topic/4208/インストール

11: エクト

Examples

エリクサープログラムでEcto.Repoをする

これは3つのステップでうことができます

1. あなたはEcto.Repoをするエリクシールモジュールをし、あなたのアプリをotp_appとしてするがあります。

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

2. また、データベースにできるようにするRepoのをいくつかするがあります。ここにはポストグルのがあります。

```
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. アプリケーションでEctoをするに、Ectoをしてからアプリをするがあります。これは、スーパーユーザーとしてlib / custom_app.exにEctoをすることでうことができます。

```
    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
```

Repo.get_by / 3の

あなたがEcto.Queryableをっていれば、タイトルとをつPostというです。

のようにして、タイトルが「hello」、が「world」のPostをできます。

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

Repo.get_byは2のにキーワードリストをしているので、これはすべてです。

フィールドをしたクエリ

にまれるのフィールドをするには、 fieldをします 。

```
some_field = :id
some_value = 10

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

カスタムデータをおよびスキーマにする

このから

のでは、 をpostgresデータベースにしています。

まず、 ファイル `mix ecto.gen.migration`をします 。

```
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
```

に、 モデルファイルで Elixirのフィールドをするか、

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

または`Ecto.Type`ビヘイビアをします。

のいは`ecto_enum`パッケージで、テンプレートとしてできます。そのは**github**ページでよくされています 。

このコミットは、Phoenixプロジェクトでenum_ectoをプロジェクトにし、ビューとモデルでをするをしています。

オンラインでエクトをむ https://riptutorial.com/ja/elixir/topic/6524/エクト

12: エリキシルの

き

は、なるタイプのエンティティへののインタフェースをすることです。には、なるデータがじにすることができます。したがって、じがじデータでじをします。エリクシールには、をきれいに`protocols`ためのの`protocols`があり`protocols`。

すべてのデータをカバーするは、`Any`データのをできます。に、があれば、 EnumとString.Charのソースコードをしてください。これは、コアのElixirののいです。

Examples

プロトコルによる

ケルビンとのをにするなプロトコルをしましょう。

```
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
```

は、KelvinとFahrenheitタイプのコンバータをしました。コンバージョンをいくつかろう

```
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"}
```

`to_celsius`のがないのデータをしようとしましょう

```
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
```

オンラインでエリキシルのをむ https://riptutorial.com/ja/elixir/topic/9519/エリキシルの

13: エリクシールの

Examples

エージェントをってをする

ラップしてにアクセスするもなは`Agent`です。このモジュールでは、のデータをし、そのをみんでするためのメッセージをするプロセスをすることができます。このため、へのアクセスは、プロセスがに1つのメッセージしかしないため、にシリアルされます。

```
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}
```

オンラインでエリクシールのをむ https://riptutorial.com/ja/elixir/topic/6596/エリクシールの

14: エリクシルのプログラミング

き

Elixirをってmapやreduceのようななをしようとしましょう

Examples

Mapはとをとり、そのをそのリストのにしたにをすです

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

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

ペーストを`iex`コピーしてします

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

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

らす

Reduceは、、、アキュムレータをとり、アキュムレータをシードとしてしてのでのアキュムレータをえ、のすべてのにしてりしをするですのをしてください

```
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
```

のスニペットをiexにりけてコピーします

1. すべてのをにするには`MyList.reduce [1,2,3,4], fn acc, element -> acc + element end, 0`
2. のすべてのをmutliplyするには `MyList.reduce [1,2,3,4], fn acc, element -> acc * element end, 1`

1の

```
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)
```

reduceをしてリストをフィルタリングする

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

オンラインでエリクシルのプログラミングをむ https://riptutorial.com/ja/elixir/topic/10186/エリクシルのプログラミング

15: エリクシルプログラムのない

オンラインでエリクシルプログラムのないをむ https://riptutorial.com/ja/elixir/topic/6493/エリクシルプログラムのない

16: エリクシルプログラムのない

.gitignoreファイルに`/rel`フォルダがないがあることにしてください。これは、 `exrm`などのリリースツールをしているにされ`exrm`

Examples

エリクシルのなgitignore

```
/_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
```

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

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

スタンドアロンエリクサー

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

Phoenixアプリケーション

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

.gitignore

デフォルトでは、 `mix new <projectname>`を`mix new <projectname>`と、プロジェクトルートにElixirにした`.gitignore`ファイルがされます。

```
# 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
```

オンラインでエリクシルプログラムのないをむ https://riptutorial.com/ja/elixir/topic/6526/エリクシルプログラムのない

17: ガードのな

Examples

ガードのない

Elixirでは、じののののをし、どのをするかをするためにをびすにのパラメータにされるをすることができます。

これらのルールはキーワード`when`でマークされ、 `def function_name(params)`との`do`にります。な

```
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
```

こので`Math.is_even(2)`をするとします。 `is_even`には、ガードがなる2つのがあります。システムはそれらをにべ、パラメータがガードをたすのをします。のものは、 `num === 1`がではないので、のものにります。 2つは`num === 2`します。これはtrueです。これはされるであり、りは`true`になり`true`。

`Math.is_odd(1)`をするとどうなりますかシステムはのをて、 `num`が`1`あるため、ののガードがたされていることをします。そのをして`true`をし、のをにするはありません。

ガードは、できるのがられています。 Elixirのドキュメントには、されたすべてのがリストされていますにえば、、、バイナリ、チェック `is_atom` 、さな `length` をにします。カスタムガードをすることはですが、マクロをするがあります。

ガードはエラーをげないことにしてください。それらはガードののとしてわれ、システムはのをるためにします。あなたがしているparamsでされたをびすときに`(FunctionClauseError) no function clause matching`ことがわかったは、するとされるガードがみまれているエラーをげているがあります。

これをでるには、ガードをつをしてびしてください。このは、0でりろうとします。

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

`BadMath.divide("anything")`をびすと`(FunctionClauseError) no function clause matching in BadMath.divide/1`がにたないエラー`(FunctionClauseError) no function clause matching in BadMath.divide/1`ます。、 `"anything" / 0`しようとすると、エラー `(ArithmeticError) bad argument in arithmetic expression`です。

オンラインでガードのなをむ https://riptutorial.com/ja/elixir/topic/6121/ガードのな

18: ストリーム

ストリームはで、したです。

そののために、ストリームはきなまたはのコレクションでするときにです。 `Enum`でくのをさせる、リストがされ、 `Stream`はでされるのレシピをします。

Examples

のの

`Stream`は、コレクションにしてのをするににです。これは、 `Stream`がで、りしが1だけである `Enum`はのりしをうなどためです。

```
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]
```

ここでは、3つの `map` 、 `filter` 、 `take_every` をチェーンしましたが、のりしは`Enum.to_list`がびされたにのみわれ`Enum.to_list`た。

`Stream`がでうことは、のがになるまですることです。そのに、すべてののリストがされますが、がなは、すべてののすべてのをしてコレクションをします。これにより、 `Enum`よりもになります。この、このは3のがわれます。

オンラインでストリームをむ https://riptutorial.com/ja/elixir/topic/2553/ストリーム

19: データ

- [ヘッド| tail] = [1,2,3、true]パターンマッチングをしてコンスセルをすることができます。これはheadを1に、tailを[2、3、true]にりてます。
- {dval} ={d1、etrue}valに1をします。 lhsのdはにパターン{d => _}をするためにされるシンボルなので、dはされませんハッシュロケットは、マップのキーとしてシンボルをキーとしてつことができますルビーで

ここでたちにどのようなデータがあるかについてながあります。

のデータがなは、たくさんのみリストをすることになります。あなたがんでたくさんんでいるのであれば、タプルをうべきです。

マップにしては、にキーバリューストアをどのようにうのかということだけです。

Examples

リスト

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

これらはリンクリストとしてメモリにされることにしてください。 ld estこれはのコンスセルで、headList.hd / 1はリストののののであり、tailList.tail / 1はりのリストのです。

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

タプル

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

タプルは、ののとです。それらはメモリにしてされます。

オンラインでデータをむ https://riptutorial.com/ja/elixir/topic/1607/データ

20: デバッグのヒント

Examples

IEX.pry / 0によるデバッグ

`IEx.pry/0`デバッグはにです。

1. モジュールに`require IEx`が`require IEx`
2. したいコードのをします
3. ラインのに`IEx.pry`をする

すぐあなたのプロジェクトをしてくださいえば、 `iex -S mix`。

`IEx.pry/0`にすると、プログラムはし、するがあります。のデバッガのブレークポイントにています。

したら、コンソールに`respawn`してください。

```
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
```

IO.inspect / 1でのデバッグ

エリクシールプログラムをデバッグするツールとしてIO.inspect / 1をすることはです。

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

それは、1と2をコンソールにします。 `IO.inspect/1`はそのをすので、びしやパイプラインにそれをめるのはにです。

```
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))
```

パイプでデバッグする

```
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]
```

でげる

```
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)>
```

オンラインでデバッグのヒントをむ https://riptutorial.com/ja/elixir/topic/2719/デバッグのヒント

21: ノード

Examples

システムのすべてのノードをする

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

じマシンのノードの

2つのターミナルウィンドウで2つのきノードをします。

```
>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)>
```

1つのノードにをして2つのノードをします。

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

2つのノードがされ、いにしています。

```
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"]
```

のノードでコードをすることができます

```
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
```

なるマシンのノードの

1つのIPアドレスできプロセスをする

```
$ 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"]
```

なるIPアドレスでのきプロセスをする

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

オンラインでノードをむ https://riptutorial.com/ja/elixir/topic/2065/ノード

22: パターンマッチング

Examples

パターンマッチング

```
#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
```

マップのパターンマッチング

```
%{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"}
```

リストのパターンマッチング

リストのようなエリクシールデータにしてもパターンマッチをうことができます。

リスト

リストのマッチングはにです。

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

これは、リストののまたはそれのを| パイプとリストのりののののに| 。

リストののについてもさせることができます。

```
[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]
```

| にのしたをバインドするまたされています

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

さらにな - のとし、それをとすることができます

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

パターンマッチングをってリストのをる

```
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
```

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

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

タプル

```
{ 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" }
```

ファイルをむ

パターンマッチングは、タプルをすファイルのみみのようなにです。

`sample.txt`に`This is a sample text`まれている`This is a sample text`

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

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

それの、ファイルがしない

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

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

とするパターン

```
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
```

オンラインでパターンマッチングをむ https://riptutorial.com/ja/elixir/topic/1602/パターンマッチング

23: ビーム

Examples

き

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

`:observer.start`はGUIオブザーバーインターフェースをき、CPUの、メモリー、およびアプリケーションのパターンをするでなそののをします。

オンラインでビームをむ https://riptutorial.com/ja/elixir/topic/3587/ビーム

24: ビルトインタイプ

Examples

エリクシールにはとがしています。 リテラルは、10、2、8、16のでできます。

```
iex> x = 291
291

iex> x = 0b100100011
291

iex> x = 0o443
291

iex> x = 0x123
291
```

Elixirはbignumをするため、のはシステムのなメモリによってのみされます。

はで、IEEE-754にしています。

```
iex> x = 6.8
6.8

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

Elixirはのもサポートしています。

```
iex> 1 + 1
2

iex> 1.0 + 1.0
2.0
```

に2つのをし、はになります。で2つのをし、はになります。

エリクシールをするとにがされます

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

に、でを、、またはけわせると、はになります。

```
iex> 40.0 + 2
42.0

iex> 10 - 5.0
5.0
```

```
iex> 3 * 3.0
9.0
```

の、 `div/2`をできます。

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

はかのをすです。のはそのです。はコロンでまります。

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

のはユニークです。じの2つのはにしい。

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

iex(2)> b = :atom
:atom

iex(3)> a == b
true

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

`true`と`false`ブーリアンは、にです。

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

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

はにされます。このテーブルがガベージコレクションされていないことをることはにです。だから、あなたがあるいははいなくであるえずをりたいなら、それはいえです。

バイナリとビットストリング

エリクシルのバイナリは、Kernel.SpecialFormsの<< >>をしてされます 。

らはElixirをバイナリプロトコルとエンコーディングのににつなツールです。

バイナリとビットストリングは、 "<<"と ">>"がいたまたはのカンマりリストをしてします。それらは「」でされ、ビットのグループまたはバイトのグループがわれます。デフォルトのグループは1バイト8ビットで、をしてされます。

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

エリクシールもバイナリにされます

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

バイナリの「セグメント」に「」をすると、エンコードすることができます。

- データ·タイプ
- サイズ
- エンディアン

これらのは、 "::"をしてまたはのにエンコードされます。

```
<<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
```

できるデータはのとおりです。

-
- く
- ビットビットストリングのエイリアス
- ビットストリング
- バイナリ
- バイトバイナリのエイリアス
- utf8
- utf16
- utf32

バイナリセグメントの 'サイズ'をするときは、セグメントでされた 'タイプ'によってなります。

- デフォルト1ビット
- 1ビット
- バイナリ8ビット

オンラインでビルトインタイプをむ https://riptutorial.com/ja/elixir/topic/1774/ビルトインタイプ

25: ヒントとテクニック

き

エリクシールコーディングにをするなヒントとテクニック。

Examples

カスタムサインとドキュメントの

x sigilはそれぞれのsigil_xをびします

カスタムサインの

```
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
```

の[OR]

これは、のORをくもう1つのです。これはアプローチではありません。のでは、がであるとされるため、リストのすべてのをにするこのアプローチとはなり、をするりののをします。これはちょうどいですが、にはいことです。

```
# 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カスタム - iexデコレーション

コンテンツをファイルにコピーし、ファイルを.iex.exsとしてのホームディレクトリにし、そのをてください。 ここでファイルをダウンロードすることもできます

```
# 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

)
```

オンラインでヒントとテクニックをむ https://riptutorial.com/ja/elixir/topic/10623/ヒントとテクニック

26: プロセス

Examples

なプロセスの

のでは、 `Greeter`モジュールの`greet`がのプロセスでされます。

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

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

ここで`#PID<0.122.0>`は、 されたプロセスのプロセスです。

メッセージの

```
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}
```

と

をしてのメッセージをすることができます

```
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は、びしがこののようにでにするり、テールコールをします。

オンラインでプロセスをむ https://riptutorial.com/ja/elixir/topic/3173/プロセス

27: プロトコル

にする

マップでプロトコルをするわりに、にはのプロトコルがです。

Examples

き

プロトコルはエリキシルでをにする。 `defprotocol`プロトコルをする

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

`defimpl`プロトコルをする

```
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
```

のでは、をうことができます。

```
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
```

プロトコルは、プロトコルをしているり、のデ―タにディスパッチさせることができます。これには、 `Atom` 、 `BitString` 、 `BitString`などの`Tuples`がまれます。

オンラインでプロトコルをむ https://riptutorial.com/ja/elixir/topic/3487/プロトコル

28: マップとキーワードリスト

- map ={} //のマップをする
- map ={a => 1、b => 2} //でないマップをする
- list = [] //のリストをする
- list = [{a、1}、{b、2}] //でないキーワードリストをする

Elixirはマップとキーワードリストという2つのデータをします 。

マップはElixirのKey-Valueののまたはハッシュともばれますタイプです。

キーワードリストは、のキーにをけるキー/のタプルです。これらはに、びしのオプションとしてされます。

Examples

マップの

マップはElixirのKey-Valueののまたはハッシュともばれますタイプです。 `%w{}`をしてマップをします。

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

キーとはのをできます

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

さらに、キーとのにタイプがするマップをつことができます "

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

マップのすべてのキーがアトムの、キーワードをできます。

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

キーワードリストの

キーワードリストはキー/のタプルであり、にびしのオプションとしてされます。

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

キーワードリストには、じキーがりされることがあります。

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

キーとにはのをできます。

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

マップとキーワードリストのい

とキーワードリストにはなるアプリケーションがあります。たとえば、マップにはじのキーが2つあり、けされていません。に、キーワードリストはパターンマッチングでするのがししいもあります。

マップとキーワードリストのをいくつかします。

のにキーワードリストをします。

- するがです
- じキーをつのがです

のにマップをします。

- いくつかのキー/にしてパターンマッチしたい
- じキーをつのはありません
- あなたがにキーワードリストをとしないときはいつでも

オンラインでマップとキーワードリストをむ https://riptutorial.com/ja/elixir/topic/2706/マップとキーワードリスト

29: メタプログラミング

Examples

コンパイルにテストをする

```
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
```

```
.

  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
```

オンラインでメタプログラミングをむ https://riptutorial.com/ja/elixir/topic/4069/メタプログラミング

30: モジュール

モジュール

Elixirでは、 `IO`や`String`などのモジュールは、フードののにぎず、コンパイルに`:"Elixir.ModuleName"`にされます。

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

Examples

モジュールのまたはマクロをする

`__info__/1`は、のアトムのいずれかをとります。

- `:functions` - パブリックのキーワードリストをそのアリティとともにします
- `:macros` - パブリックマクロのキーワードリストをそのアリティとともにします

`Kernel`モジュールのをするには

```
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, ...]
```

`Kernel`をあなたがんだモジュールにきえてください。

モジュールの

モジュールには、 `alias` 、 `import` 、 `use` 、および`require`のモジュールでするための4つのキーワードがあり`require`。

`alias`はなるはよりいでモジュールをします

```
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`は、モジュールのすべてのをのににします。

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

`use`は、モジュールがのモジュールにコードをできるようにします。これは、、モジュールがらかのをするためののをするフレームワークのとしてされます。

それらをできるようにモジュールからロードマクロを`require`する。

のモジュールへのの

`defdelegate`をして、のモジュールでされたじのにするをします。

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


iex> Math.pi
3.141592653589793
```

オンラインでモジュールをむ https://riptutorial.com/ja/elixir/topic/2721/モジュール

31: リスト

- []
- [1,2,3,4]
- [1,2] ++ [3,4] - > [1,2,3,4]
- hd[1、2、3、4]→1
- t1[1,2,3,4]→[2,3,4]
- [ヘッド|]
- [1 | [2,3,4]]→[1,2,3,4]
- [1 | [2 | [3 | [4 | []]]]] - > [1,2,3,4]
- 'hello' = [h、e、l、l、o]
- keyword_list = [a123、b456、c789]
- keyword_list [a] - > 123

Examples

キーワードリスト

キーワードリストは、リストのがアトムのタプルのにがくリストです。

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

キーワードリストをするためのは、のとおりです。

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

キーワードリストは、けされたキーとのペアのデータをするのにです。ここでは、されたキーにしてのがすることがあります。

のキーのキーワードリストののは、のようにできます。

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

キーワードリストのユースケースは、するきタスクのシーケンスです。

```
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
```

このコードはのようなキーワードリストでびすことができます

```
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"...
```

リスト

Elixirのは "バイナリ"です。しかし、Erlangコードでは、はに "char list"なので、Erlangをびすときは、のElixirのわりにcharリストをするがあります。

のはをしてかれていますが" 、charのリストは、をしてかれています' 。

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

リストは、のコードポイントをすのなるリストです。

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

は、 `to_charlist/1`を`to_charlist/1`してcharリストにできます。

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

そしては`to_string/1`うことができます

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

Erlangをびし、をのElixirにする

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

コンスセル

エリクシールのリストはリンクされたリストです。これは、リストのがでされ、そのにリストののへのポインタがくことをします。これはコンスセルをってElixirにされています。

コンスセルは、「」と「」の、または「」と「」のをつなデータです。

は、リストののののにして、されたとをつなリストにすることができます。はが、がコンスセルです。

```
[1 | 2]
```

リストのなElixirは、にネストされたコンスセルのチェーンをくこととじです

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

リスト[]は、リストのわりをすコンスセルのとしてされます。

Elixirのすべてのリストは、`[head | tail]`、`head`はリストのの、`tail`はリストのりのからをいたものです。

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

`[head | tail]`は、のパターンマッチングにちます。

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

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

マッピングリスト

`map`は、とをえられたプログラミングのであり、そのをそのリストのにしてしいリストをします。Elixirでは、`map/2`はEnumモジュールにあります。

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

のキャプチャの

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

キャプチャをつをする

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

パイプをしたリストの

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

リストの

エリクサーにはループがありません。リストのわりにらしい`Enum`と`List`モジュールがありますが、List Comprehensionsもあります。

リストのは、のにちます。

- しいリストをする

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

- `guard`をしてリストをフィルタリングしますが、キーワードをしない`when guard`をします。

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

- キーワードをっ`into`カスタムマップをする

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

された

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

リスト

- `for..do`をし、コンマのにガードが`into`、リストのをすとき`into`キーワードに`into`ます。。
- それのはしいリストをす
- アキュムレータはサポートしていません
-

のがたされたときにをできません
- `guard`は、のでにするが`for`とに`do`か`into`シンボル。のはありません

これらのによれば、リストはなのためにのみされています。よりなケースでは、 `Enum`と`List`モジュールのをすることがです。

リストのい

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

`--`のアイテムのリストのアイテムののをします。

メンバーシップの

がリストのメンバーであるかどうかをするに`in`で`in`ます。

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

リストをマップにする

`Enum.chunk/2`サブリストにグループに、そして`Map.new/2`にします

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

える

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

オンラインでリストをむ https://riptutorial.com/ja/elixir/topic/1279/リスト

32:

- Task.asyncしい
- Task.awaitタスク

パラ メーター

パラ メーター	
しい	のプロセスでするがある。
	`Task.async`によってされたタスク。

Examples

バック グラウンドでをしている

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

```
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)
```

オンラインでをむ https://riptutorial.com/ja/elixir/topic/7588/

33:

これは、がElixirモジュールでどのようにをするのかにされているについてったです。 。はいくつかのでそれをしています

- ほとんどのElixirのドキュメントはかなりしていますが、このなアーキテクチャーのにガイダンスがけていることがわかりました。
- はしのをて、トピックについてのからコメントしたかったのです。
- また、しいSOドキュメンテーションワークフローをしてみたかったのです。 ;

また、コードをGitHub repo elixir-constants-conceptにアップロードしました。

Examples

モジュールスコープ

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

これらは、このモジュールからのみアクセスできます。

としての

```
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
```

する

```
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
```

でする

```
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
```

このでは、プロジェクトでをできますが、コンパイルをとするガードではできません。

マクロによる

```
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
```

`use`に`use`

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

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

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

このメソッドでは、 `@some_constant`に`@some_constant`をできます。はそのがにであるかどうかもかりません。

オンラインでをむ https://riptutorial.com/ja/elixir/topic/6614/

34:

Elixirの`String`は、 `UTF-8`エンコードされたバイナリです。

Examples

にする

Kernel.inspectをしてかをにします。

```
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\"}"
```

をする

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

をする

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

の

```
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."
```

Stringにサブストリングがまれている

```
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
```

をする

`<>`をってElixirのをすることができます

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

`Enum.join/2` `List`では、 `Enum.join/2`できます。

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

オンラインでをむ https://riptutorial.com/ja/elixir/topic/2618/

35: をする

Examples

のをう

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

IOリストの

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

これは、メモリでしないとしてのパフォーマンスをさせます。

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

Enum.joinの

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

オンラインでをするをむ https://riptutorial.com/ja/elixir/topic/9202/をする

36:

Examples

ずにしてください

これらはにパフォーマンスをさせるなヒントです。コードがいは、いをするためにプロファイルをすることがにです。はしてではありません。おそらくの1をめるだけのもののをすることは、そのにするものではありません。きなのシンクをしてください。

なをするには、プロファイリングにしているコードがなくとも1されていることをしてください。そのでの10をやしたは、プログラムのが10かかることをして、じなデータをしてりしなをできることをしてください。

ExProfはにいめることができます。

オンラインでをむ https://riptutorial.com/ja/elixir/topic/6062/

37:

`do...end`は、のキーワードリストのななので、にはこれをうことができます

```
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

```
 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`は、のデータのされたパターンをするためにのみされます。ここで、 `{1,2}`はコードでえられたなるケースパターンとしています。

ifとunless

```
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 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)
```

がでない、 `cond`は`CondClauseError`ます。

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

これは、にであるをすることでできます。

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

デフォルトのケースにすることはしてされないり、プログラムはにそのでクラッシュすべきです。

with

`with`は、するをするためにされます。をみわせたり、をのするとみわせるようにえます。このをえてみましょう。ユーザーをし、DBにしてから、メールをしてユーザーにします。

`with`がなければ、のようにくことができますはのをしました。

```
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
```

ここでは、ビジネスプロセスのれを`case`します `cond`または`if`であるがあります。それは、たちがなにし、よりくのをするかどうかをしなければならないので、いわゆる「のピラミッド」につながります。このコード`with`ステートメントできすほうがはるかにいでしょう。

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

のコードスニペットでは、ネストされたcaseをwithきしました。 withでwithいくつかのまたはきとそののパターンマッチをびします。すべてがマッチしたは、 withリターンdoブロックの、またはelseそれのブロックのを。

たちは、することができelseように、 withいずれかをしますdoブロックのをまたはのをします。

ですから、 withのはdoブロックのです。

オンラインでをむ https://riptutorial.com/ja/elixir/topic/2118/

38:

Examples

カスタムミックスタスクをする

```
# 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
```

コンパイルしてする

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

コマンドラインをしたカスタムミックスタスク

なでは、タスクモジュールはのリストをる`run/1`をしなければなりません。えば`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
```

コンパイルしてする

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

エイリアス

Elixirでは、ミックスコマンドのエイリアスをすることができます。クールなことは、あなたのいくつかのをしたい。

あなたのElixirプロジェクトで`mix.exs`をきます。

まず、 `project`がすキーワードリストに`aliases/0`をします。 *aliases*のに*Adding* `()`をすると、コンパイラはをスローすることができません。

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

に、 `aliases/0`をします `mix.exs`ファイルの。

```
...

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

Phoenixアプリケーションをしているは、 `$ mix go`ってPhoenixサーバをできます。また、 `$ mix trident`をして、すべてのをフェッチし、コンパイルし、サーバーをするようにmixにします。

なミックスタスクのヘルプをする

なミックスタスクをするには、

```
mix help
```

のタスクにするヘルプをるには、 `mix help task`してください

```
mix help cmd
```

オンラインでをむ https://riptutorial.com/ja/elixir/topic/3585/

39:

Examples

パイプオペレータ

Pipe Operator `|>`はののをけり、のパラメータとしてのにります。

```
expression |> function
```

をさせ、ののフロをにするには、パイプをします。

のをしてください。

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

このでは、 `Oven.bake`は`Ingredients.mix`にありますが、にされます。また、それはらかではないかもしれません`:temperature`は`Oven.bake`パラメータです

パイプをしてこのをきす

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

じがられますが、がになります。さらに、 `:temperature`が`Oven.bake`コールのパラメータであることはらかです。

パイプをする、ののパラメータはパイプのにされるため、びされるのパラメータは1つなくなります。えば

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

のものとじです

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

パイプとかっこ

あいまいさをけるためにはかっこがです。

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

のようにするがあります。

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

ブーール

エリクシーールには2のブーールがあります

- ブーールのとして`true`または`false`いずれかが`true`

```
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
```

すべてのブーールは、のがにブーールではなく、 `true`または`false nil`はブーールではないのみをする、`ArgumentError`させます。

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

- リラックスブーールのでし、 `false`でもなくても`nil`でないものはすべて`true`とみなされ`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
```

オペレーター`||`であればのをにしますElixirは`nil`と`false`をくすべてをでとみなします。そうでなければ2のをします。

```
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
```

`&&`はならに2をします。それのは、それぞれ`false`または`nil`ります。

```
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
```

`&&`と`||`オペレータです。がをするのにでないにのみ、をします。

オペレーター!ののののブールをします

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

されたのブールをるなは、にこのを2にすることです。

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

- のしい`x == y 1 == 1.0 # true`
- の`x == y 1 != 1.0 # false`
- な`x === y 1 === 1.0 # false`
- な`x === y 1 !== 1.0 # true`

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

のがある、ではけがされます。それのは、なのルールがあります。

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

バイナリをむとリストをすることができます

```
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'

`in`をすると、リストまたはにアイテムがまれているかどうかをできます。

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

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

オンラインでをむ https://riptutorial.com/ja/elixir/topic/1161/

40:

Examples

き

ビヘイビアは、のモジュールができるのリストです。それらはののインタフェースにています。

にをします。

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

そしてそれをするモジュール

```
defmodule JSONParser do
  @behaviour Parser

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

の`@behaviour`モジュールは、このモジュールがParserモジュールでされたすべてのをすることがされることをします。がすると、されていないのコンパイルエラーがします。

モジュールは、つことができます`@behaviour`を。

オンラインでをむ https://riptutorial.com/ja/elixir/topic/3558/

41:

Examples

Elixirでは、なとして、をします。のはです

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

なはのとおりです。

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

みやすくするために、のにかっこをれることができます

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

をびすには、りてられたでびしてし.とのに

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

なしでをすることはです

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

キャプチャの

をよりにするために、 キャプチャ &することができます。たとえば、のわりに

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

あなたはける

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

のパラメータをするは、にするを&1からえてします。

```
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
```

の

は、 パターンマッチングのとしてのボディをつこともできます

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

のボディをつをびすときElixirは、したパラメータをなとしようとします。

リストとしてのキーワードリスト

のKey-Valueペアをむ 'options'のパラメータにキーワードリストをする

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

のをのようにびすことができます

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

これはとです

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

のは、それぞれ`opts.pizza`および`opts.soda`としてできます。
あるいは、atoms `opts[:pizza]`と`opts[:soda]`することもできます。

きとプライベート

き

```
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
```

プライベート

```
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
```

パターンマッチング

Elixirは、のについてのびしをします。

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

ここで、のは2とし、 `factorial(0)` はのとします。 のためにのをする。 Elixirはからにをマッチさせようとします。 2のが1ののにかれている、のにくのでせぬになります。 `factorial(0)` は `factorial(n)` するので、

ガード

ガードをすると、をするにをチェックできます。ガードは、、みやすさのために `if` と `cond` よりされ、コンパイラにとってのをにします。すべてのガードがするのがされます。

ガードとパターンマッチングをしたのをにします。

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

のパターンは、が0にのみし0 。が0でなければ、パターンのはし、ののがチェックされます。

その2のには、ガードがありますwhen n > 0です。 つまり、このは、nが0よりきいにのみします。のところ、はのにしてされていません。

パターンマッチングとガードをむがしない、 FunctionClauseErrorがします。これは、のがされていないため、のをとしてすときにこのでします。

このFunctionClauseErrorはいではないことにしてください。のいくつかのでよくられるように、 -1や0などの「エラー」をすと、のをびしてエラーのをしてしまい、のにとってなバグがするがあります。

デフォルトパラメータ

をして、のきにデフォルトのパラメータをすことができます 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]
```

キャプチャ

のモジュールからをキャプチャするには、 &をします。キャプチャされたは、パラメータとして、またはでできます。

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

&をしてよりにすることができます

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

をさずにをキャプチャするには、そのをにするがあります &String.capitalize/1

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

オンラインでをむ https://riptutorial.com/ja/elixir/topic/2442/

クレジット

S. No		Contributors
1	Elixirをいめる	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	Doctests	aholt, milmazz, Philippe-Arnaud de MANGOU, Yos Riady
3	ExDoc	milmazz, Yos Riady
4	ExUnit	Yos Riady
5	IExコンソールでのヘルプの	helcim
6	IExコンソールのヒントとコツ	alxndr, Cifer, fahrradflucht, legoscia, mudasobwa, muttonlamb, PatNowak, Paweł Obrok, sbs, Sheharyar, Simone Carletti, Stephan Rodemeier, Uniaika, Vincent, Yos Riady
7	IO.inspectとラベルによるデバッグの	leifg
8	Sigils	javanut13, Yos Riady
9	アーラン	4444, Yos Riady
10	インストール	cwc, Douglas Correa, Eiji, JAlberto, MattW.
11	エクト	fgutierr, Philippe-Arnaud de MANGOU, toraritte
12	エリキシルの	mustafaturan
13	エリクシールの	Paweł Obrok
14	エリクシルのプログラミング	Dinesh Balasubramanian
15	エリクシルプログラムのない	Yos Riady
16	ガードのな	alxndr
17	ストリーム	Oskar

18	データ	Sam Mercier, Simone Carletti, Stephan Rodemeier, Yos Riady
19	デバッグのヒント	javanut13, Paweł Obrok, Pfitz, Philippe-Arnaud de MANGOU, sbs
20	ノード	Yos Riady
21	パターンマッチング	Alex Anderson, Dair, Danny Rosenblatt, evuez, Gabriel C, gmile, Harrison Lucas, javanut13, Oskar, PatNowak, theIV, Thomas, Yedhu Krishnan
22	ビーム	Yos Riady
23	ビルトインタイプ	Andrey Chernykh, Arithmeticbird, Oskar, TreyE, Vinicius Quaiato
24	ヒントとテクニック	Ankanna
25	プロセス	Alex G, Yedhu Krishnan
26	プロトコル	Yos Riady
27	マップとキーワードリスト	Sam Mercier, Simone Carletti, Yos Riady
28	メタプログラミング	4444, Paweł Obrok
29	モジュール	Alex G, javanut13, Yos Riady
30	リスト	Ben Bals, Candy Gumdrop, emoragaf, PatNowak, Sheharyar, Yos Riady
31		mario
32		ibgib
33		Alex G, Sheharyar, Yos Riady
34	をする	Agung Santoso
35		Filip Haglund, legoscia
36		Andrey Chernykh, evuez, javanut13, Musfiqur Rahman, Paweł Obrok
37		4444, helcim, rainteller, Slava.K, Yos Riady
38		alxndr, Andrey Chernykh, Dair, Gazler, Mitkins, nirev, PatNowak
39		Yos Riady

40		Andrey Chernykh, cwc, Dair, Eiji, Filip Haglund, PatNowak, rainteller, Simone Carletti, Stephan Rodemeier, Yedhu Krishnan , Yos Riady