EBook Gratuito

APPRENDIMENTO

Elixir Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#elixir

Sommario

Di ... 1

Capitolo 1: Iniziare con Elixir Language ... 2

Osservazioni ... 2

Versioni ... 2

Examples ... 2

Ciao mondo ... 2

Ciao mondo da IEx ... 3

Capitolo 2: .Gitignore di base per il programma di elisir ... 5

Capitolo 3: .Gitignore di base per il programma di elisir ... 6

Osservazioni ... 6

Examples ... 6

Un originale .gitignore per Elisir ... 6

Esempio ... 6

Applicazione di elisir standalone ... 6

Applicazione Phoenix ... 7

Auto-generato .gitignore ... 7

Capitolo 4: Compito ... 8

Sintassi ... 8

Parametri ... 8

Examples ... 8

Lavorando in background ... 8

Elaborazione parallela ... 8

Capitolo 5: comportamenti ... 9

Examples ... 9

introduzione ... 9

Capitolo 6: Condizionali ... 10

Osservazioni ... 10

Examples ... 10

Astuccio ... 10

se e a meno ... 10

cond 11
con clausola 11
Capitolo 7: costanti 13
Osservazioni 13
Examples 13
Costanti con scope del modulo 13
Costanti come funzioni 13
Costanti tramite macro 14
Capitolo 8: doctests 16
Examples 16
introduzione 16
Generazione di documentazione HTML basata su doctest 16
Doctest multilinea 16
Capitolo 9: Ecto 18
Examples 18
Aggiunta di un Ecto.Repo in un programma di elisir 18
"e" clausola in un Repo.get_by / 3 18
Interrogazione con campi dinamici 19
Aggiungi tipi di dati personalizzati alla migrazione e allo schema 19
Capitolo 10: elenchi 20
Sintassi 20
Examples 20
Elenchi di parole chiave 20
Char Lists 21
Contro le cellule 22
Liste di mappatura 23
Elenco delle comprensioni 23
Esempio combinato 23
Sommario 24
Lista differenza 24
Elenco membri 24
Convertire le liste in una mappa 24

Capitolo 11: Erlang 26
Examples 26
Utilizzando Erlang 26
Ispeziona un modulo di Erlang 26
Capitolo 12: ExDoc 27
Examples 27
introduzione 27
Capitolo 13: ExUnit 28
Examples 28
Asserire eccezioni 28
Capitolo 14: FASCIO 29
Examples 29
introduzione 29
Capitolo 15: funzioni 30
Examples 30
Funzioni anonime 30
Utilizzando l'operatore di cattura 30
Corpi multipli 31
Elenchi di parole chiave come parametri di funzione 31
Funzioni nominate e funzioni private 31
Pattern Matching 32
Clausole di guardia 32
Parametri predefiniti 33
Cattura le funzioni 33
Capitolo 16: Gestione dello stato in elisir 35
Examples 35
Gestire un pezzo di stato con un agente 35
Capitolo 17: Installazione 36
Examples 36
Installazione di Fedora 36
Installazione OSX 36

homebrew ... 36
macports ... 36
Installazione Debian / Ubuntu ... 36
Installazione di Gentoo / Funtoo ... 36
Capitolo 18: Mappe e elenchi di parole chiave ... 38
Sintassi ... 38
Osservazioni ... 38
Examples ... 38
Creazione di una mappa ... 38
Creazione di un elenco di parole chiave ... 39
Differenza tra mappe e elenchi di parole chiave ... 39
Capitolo 19: Mescolare ... 40
Examples ... 40
Crea un'attività di mix personalizzato ... 40
Attività di missaggio personalizzata con argomenti della riga di comando ... 40
alias ... 40
Ottieni aiuto sulle missioni disponibili ... 41
Capitolo 20: metaprogrammazione ... 42
Examples ... 42
Genera test in fase di compilazione ... 42
Capitolo 21: Migliore debugging con IO.inspect ed etichette ... 43
introduzione ... 43
Osservazioni ... 43
Examples ... 43
Senza etichette ... 43
Con etichette ... 44
Capitolo 22: moduli ... 45
Osservazioni ... 45
Nomi dei moduli ... 45
Examples ... 45
Elenca le funzioni o i macro di un modulo ... 45

Utilizzando i moduli 45
Delega di funzioni a un altro modulo 46
Capitolo 23: nodi 47
Examples 47
Elencare tutti i nodi visibili nel sistema 47
Collegamento di nodi sulla stessa macchina 47
Collegamento di nodi su macchine diverse 47
Capitolo 24: operatori 49
Examples 49
L'operatore del tubo 49
Operatore del tubo e parentesi 49
Operatori booleani 50
Operatori di confronto 51
Unisciti agli operatori 51
Operatore "In" 52
Capitolo 25: Ottenere aiuto nella console IEx 53
introduzione 53
Examples 53
Elenco dei moduli e delle funzioni di Elixir 53
Capitolo 26: Ottimizzazione 54
Examples 54
Misura sempre per primo! 54
Capitolo 27: Pattern matching 55
Examples 55
Funzioni di corrispondenza del modello 55
Pattern matching su una mappa 55
Pattern matching su una lista 55
Ottieni la somma di un elenco usando la corrispondenza del modello 56
Funzioni anonime 56
Le tuple 57
Leggere un file 57
Pattern che corrisponde alle funzioni anonime 57

Capitolo 28: Polimorfismo in elisir 59
introduzione 59
Osservazioni 59
Examples 59
Polimorfismo con protocolli 59
Capitolo 29: Processi 61
Examples 61
Generare un processo semplice 61
Invio e ricezione di messaggi 61
Ricorsione e ricezione 61
Capitolo 30: Programmazione funzionale in elisir 63
introduzione 63
Examples 63
Carta geografica 63
Ridurre 63
Capitolo 31: protocolli 65
Osservazioni 65
Examples 65
introduzione 65
Capitolo 32: ruscello 66
Osservazioni 66
Examples 66
Concatenare più operazioni 66
Capitolo 33: sigilli 67
Examples 67
Costruisci un elenco di stringhe 67
Costruisci una lista di atomi 67
Sigilli personalizzati 67
Capitolo 34: stringhe 68
Osservazioni 68
Examples 68

Converti in stringa 68
Ottieni una sottostringa 68
Dividere una stringa 68
Interpolazione a stringa 68
Controlla se String contiene la sottostringa 68
Unisci le stringhe 69
Capitolo 35: Strutture dati 70
Sintassi 70
Osservazioni 70
Examples 70
elenchi 70
Le tuple 70
Capitolo 36: Suggerimenti e trucchi 71
introduzione 71
Examples 71
Creazione di sigilli personalizzati e documentazione 71
Multiplo [OR] 71
Configurazione personalizzata iex - Decorazione iex 71
Capitolo 37: Suggerimenti e trucchi della console IEx 73
Examples 73
Ricompilare il progetto con `ricompilare` 73
Vedi la documentazione con `h` 73
Ottieni valore dall'ultimo comando con `v` 73
Ottieni il valore di un comando precedente con `v` 73
Esci dalla console IEx 74
Vedi le informazioni con `i` 74
Creazione di PID 75
Prepara i tuoi alias quando avvii IEx 75
Storia persistente 75
Quando la console Elixir è bloccata 75
rompere di espressione incompleta 76
Carica un modulo o uno script nella sessione IEx 77

Capitolo 38: Suggerimenti per il debug 78
Examples 78
Debugging con IEX.pry / 0 78
Debugging con IO.inspect / 1 78
Debug in pipe 79
Fare leva nel tubo 79
Capitolo 39: Tipi incorporati 81
Examples 81
Numeri 81
atomi 82
Binari e Bitstring 82
Capitolo 40: Unisci le stringhe 85
Examples 85
Utilizzo dell'interpolazione stringa 85
Uso dell'elenco IO 85
Utilizzando Enum.join 85
Capitolo 41: uso di base delle clausole di guardia 86
Examples 86
usi di base delle clausole di guardia 86
Titoli di coda 88

Di

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

Capitolo 1: Iniziare con Elixir Language

Osservazioni

Elixir è un linguaggio dinamico e funzionale progettato per creare applicazioni scalabili e manutenibili.

Elixir sfrutta la VM di Erlang, nota per l'esecuzione di sistemi a bassa latenza, distribuiti e fault-tolerant, e allo stesso tempo utilizzata con successo nello sviluppo web e nel dominio del software embedded.

Versioni

Versione	Data di rilascio
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

Ciao mondo

Per le istruzioni di installazione su elisir controllare qui , descrive le istruzioni relative alle diverse piattaforme.

Elixir è un linguaggio di programmazione creato usando `erlang` e utilizza il runtime `BEAM` di erlang (come `JVM` per java).

Possiamo usare elixir in due modalità: `iex` shell interattivo o in esecuzione diretta utilizzando il comando `elixir` .

Inserire quanto segue in un file denominato `hello.exs` :

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

Dalla riga di comando, digitare il comando seguente per eseguire il file di origine Elixir:

```
$ elixir hello.exs
```

Questo dovrebbe produrre:

    Ciao mondo!

Questo è noto come la *modalità script* di `Elixir` . In effetti, i programmi Elixir possono anche essere compilati (e generalmente sono) in bytecode per la macchina virtuale BEAM.

Puoi anche usare `iex` per la shell interattiva di elisir (consigliato), esegui il comando riceverai un prompt come questo:

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

Qui puoi provare i tuoi esempi di elisir `hello world` :

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

Puoi anche compilare ed eseguire i tuoi moduli tramite `iex` . Ad esempio, se hai un `helloworld.ex` che contiene:

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

Attraverso `iex` , fai:

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

Ciao mondo da IEx

È inoltre possibile utilizzare la `IEx` (Interactive Elixir) per valutare espressioni ed eseguire codice.

Se sei su Linux o Mac, digita iex sul tuo bash e premi invio:

```
$ iex
```

Se sei su un computer Windows, digita:

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

Quindi entrerai in IEx REPL (leggi, valuta, stampa, loop) e puoi semplicemente digitare qualcosa come:

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

Se vuoi caricare uno script mentre apri un REPL IEx, puoi farlo:

```
$ iex script.exs
```

Dato `script.exs` è il tuo script. È ora possibile chiamare le funzioni dallo script nella console.

Leggi Iniziare con Elixir Language online: https://riptutorial.com/it/elixir/topic/954/iniziare-con-elixir-language

Capitolo 2: .Gitignore di base per il programma di elisir

Leggi .Gitignore di base per il programma di elisir online: https://riptutorial.com/it/elixir/topic/6493/-gitignore-di-base-per-il-programma-di-elisir

Capitolo 3: .Gitignore di base per il programma di elisir

Osservazioni

Nota che la cartella `/rel` potrebbe non essere necessaria nel tuo file .gitignore. Questo viene generato se si utilizza uno strumento di gestione del rilascio come `exrm`

Examples

Un originale .gitignore per Elisir

```
/_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
```

Esempio

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

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

Applicazione di elisir standalone

```
/_build
/cover
```

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

Applicazione Phoenix

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

Auto-generato .gitignore

Per impostazione predefinita, `mix new <projectname>` genererà un file `.gitignore` nella radice del progetto adatto per Elixir.

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

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

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

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

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

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

Leggi .Gitignore di base per il programma di elisir online: https://riptutorial.com/it/elixir/topic/6526/-gitignore-di-base-per-il-programma-di-elisir

Capitolo 4: Compito

Sintassi

- Task.async (divertente)
- Task.await (task)

Parametri

Parametro	Dettagli
divertimento	La funzione che dovrebbe essere eseguita in un processo separato.
compito	L'attività restituita da `Task.async` .

Examples

Lavorando in background

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

Elaborazione parallela

```
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)
```

Leggi Compito online: https://riptutorial.com/it/elixir/topic/7588/compito

Capitolo 5: comportamenti

Examples

introduzione

I comportamenti sono un elenco di specifiche delle funzioni che possono essere implementate da un altro modulo. Sono simili alle interfacce in altre lingue.

Ecco un esempio di comportamento:

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

E un modulo che lo implementa:

```
defmodule JSONParser do
  @behaviour Parser

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

L' `@behaviour` modulo `@behaviour` alto indica che questo modulo dovrebbe definire ogni funzione definita nel modulo Parser. Le funzioni mancanti comporteranno errori di compilazione delle funzioni comportamentali non definite.

I moduli possono avere più attributi `@behaviour` .

Leggi comportamenti online: https://riptutorial.com/it/elixir/topic/3558/comportamenti

Capitolo 6: Condizionali

Osservazioni

Tieni presente che la sintassi del `do...end` è lo zucchero sintattico per gli elenchi di parole chiave regolari, quindi puoi effettivamente eseguire questa operazione:

```
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

Astuccio

```
 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` è usato solo per abbinare il modello dato dei dati particolari. Qui, `{1,2}` corrisponde a un diverso modello di caso che viene fornito nell'esempio di codice.

se e a meno

```
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` un `CondClauseError` se nessuna condizione è vera.

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

Questo può essere evitato aggiungendo una condizione che sarà sempre vera.

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

A meno che non sia previsto il raggiungimento del caso predefinito, e il programma dovrebbe in effetti bloccarsi in quel punto.

con clausola

`with` clausola viene utilizzata per combinare le clausole di corrispondenza. Sembra che combiniamo funzioni anonime o gestiamo funzioni con più corpi (clausole di corrispondenza). Considera il caso: creiamo un utente, lo inseriamo nel DB, quindi creiamo e-mail di saluto e quindi lo inviamo all'utente.

Senza la clausola `with` potremmo scrivere qualcosa di simile (ho omesso le implementazioni di funzioni):

```
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
```

Qui gestiamo il flusso del processo aziendale con il `case` (potrebbe essere `cond` o `if`). Questo ci porta alla cosiddetta "piramide del destino" , perché dobbiamo affrontare le possibili condizioni e

decidere: se avanzare o meno. Sarebbe molto più bello riscrivere questo codice `with` istruzione:

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

Nello snippet di codice sopra abbiamo riscritto le clausole dei `case` annidate con `with` . Entro `with` invochiamo alcune funzioni (sia in forma anonima o nome) e pattern matching sulle loro uscite. Se tutto abbinato, `with` ritorno `do` risultato del blocco, o `else` risultato del blocco in caso contrario.

Siamo in grado di omettere `else` modo `with` restituirà o `do` risultato del blocco o il risultato prima fallire.

Così, il valore di `with` dichiarazione è il suo `do` risultato del blocco.

Leggi Condizionali online: https://riptutorial.com/it/elixir/topic/2118/condizionali

Capitolo 7: costanti

Osservazioni

Quindi questa è un'analisi riassuntiva che ho fatto sulla base dei metodi elencati in Come si definiscono le costanti nei moduli Elixir? . Lo sto postando per un paio di motivi:

- La maggior parte della documentazione su Elixir è piuttosto approfondita, ma ho trovato questa fondamentale decisione architettonica priva di indicazioni - quindi l'avrei richiesta come argomento.
- Volevo ottenere un po 'di visibilità e commenti da altri sull'argomento.
- Volevo anche testare il nuovo flusso di lavoro della documentazione SO. ;)

Ho anche caricato l'intero codice sul GitHub repo elixir-constants-concept .

Examples

Costanti con scope del modulo

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

Questi sono accessibili solo da questo modulo.

Costanti come funzioni

Dichiarare:

```
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
```

Consumare con richiedono:

```
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
```

Consuma con l'importazione:

```
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
```

Questo metodo consente il riutilizzo di costanti attraverso i progetti, ma non saranno utilizzabili all'interno di funzioni di protezione che richiedono costanti in fase di compilazione.

Costanti tramite macro

Dichiarare:

```
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
```

Consumare con l' `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
```

Questo metodo ti consente di usare `@some_constant` all'interno di guardie. Non sono nemmeno sicuro che le funzioni sarebbero strettamente necessarie.

Leggi costanti online: https://riptutorial.com/it/elixir/topic/6614/costanti

Capitolo 8: doctests

Examples

introduzione

Quando documenti il tuo codice con `@doc` , puoi fornire esempi di codice come:

```
# 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
```

Puoi aggiungere gli esempi di codice come casi di test in una delle tue suite di test:

```
# myproject/test/doc_test.exs

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

Quindi, è possibile eseguire i test con il test del `mix test` .

Generazione di documentazione HTML basata su doctest

Poiché la generazione della documentazione si basa sul markdown, devi fare 2 cose:

1 / Scrivi il tuo doctest e rendi chiari i tuoi esempi di doctest per migliorare la leggibilità (è meglio dare un titolo, come "esempi" o "test"). Quando scrivi i tuoi test, non dimenticare di dare 4 spazi al tuo codice di test in modo che sia formattato come codice nella documentazione HTML.

2 / Quindi, inserisci "mix docs" nella console alla radice del tuo progetto elixir per generare la documentazione HTML nella directory doc situata nella radice del tuo progetto elixir.

```
$> mix docs
```

Doctest multilinea

Puoi fare un doctest su più righe usando '...>' per le linee che seguono il primo

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

Leggi doctests online: https://riptutorial.com/it/elixir/topic/2708/doctests

Capitolo 9: Ecto

Examples

Aggiunta di un Ecto.Repo in un programma di elisir

Questo può essere fatto in 3 passaggi:

1. Devi definire un modulo elixir che usi Ecto.Repo e registra la tua app come otp_app.

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

2. È inoltre necessario definire alcune configurazioni per il Repo che consentiranno di connettersi al database. Ecco un esempio con postgres.

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

3. Prima di utilizzare Ecto nell'applicazione, è necessario assicurarsi che Ecto sia avviato prima dell'avvio dell'app. Può essere fatto con la registrazione di Ecto in lib / custom_app.ex come supervisore.

```
    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
```

"e" clausola in un Repo.get_by / 3

Se si dispone di un Ecto.Queryable, denominato Post, che ha un titolo e una descrizione.

Puoi recuperare il post con il titolo: "ciao" e descrizione: "mondo" eseguendo:

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

Tutto ciò è possibile perché Repo.get_by si aspetta in secondo argomento un elenco di parole chiave.

Interrogazione con campi dinamici

Per interrogare un campo il cui nome è contenuto in una variabile, utilizzare la funzione campo .

```
some_field = :id
some_value = 10

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

Aggiungi tipi di dati personalizzati alla migrazione e allo schema

(Da questa risposta)

L'esempio seguente aggiunge un tipo enumerato a un database postgres.

Innanzitutto, modifica il **file di migrazione** (creato con `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
```

In secondo luogo, nel **file** del **modello** aggiungi un campo con un tipo di Elisir:

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

o implementare il comportamento di `Ecto.Type` .

Un buon esempio per quest'ultimo è il pacchetto `ecto_enum` e può essere usato come modello. Il suo utilizzo è ben documentato sulla sua **pagina github** .

Questo commit mostra un esempio di utilizzo in un progetto Phoenix aggiungendo enum_ecto al progetto e utilizzando il tipo enumerato nelle viste e nei modelli.

Leggi Ecto online: https://riptutorial.com/it/elixir/topic/6524/ecto

Capitolo 10: elenchi

Sintassi

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

Examples

Elenchi di parole chiave

Gli elenchi di parole chiave sono elenchi in cui ogni elemento dell'elenco è una tupla di un atomo seguita da un valore.

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

Una notazione abbreviata per la scrittura di elenchi di parole chiave è la seguente:

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

Gli elenchi di parole chiave sono utili per la creazione di strutture dati di coppie valore-chiave ordinate, in cui possono esistere più articoli per una determinata chiave.

Il primo elemento in un elenco di parole chiave per una determinata chiave può essere ottenuto in questo modo:

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

Un caso d'uso per gli elenchi di parole chiave potrebbe essere una sequenza di attività denominate da eseguire:

```
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
```

Questo codice può essere chiamato con un elenco di parole chiave come questo:

```
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 Lists

Le stringhe in elisir sono "binari". Tuttavia, nel codice Erlang, le stringhe sono tradizionalmente "elenchi di caratteri", quindi quando si chiamano le funzioni di Erlang, potrebbe essere necessario utilizzare elenchi di caratteri anziché normali stringhe di elisir.

Mentre le stringhe regolari sono scritte usando virgolette doppie `"` , le liste dei caratteri vengono scritte usando le virgolette singole `'` :

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

Gli elenchi di caratteri sono semplicemente elenchi di numeri interi che rappresentano i punti di codice di ciascun carattere.

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

Una stringa può essere convertita in una lista char con `to_charlist/1` :

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

E il contrario può essere fatto con `to_string/1` :

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

Chiamando una funzione di Erlang e convertendo l'output in una normale stringa di Elisir:

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

Contro le cellule

Gli elenchi in elisir sono elenchi concatenati. Ciò significa che ogni elemento in un elenco è costituito da un valore, seguito da un puntatore all'elemento successivo nell'elenco. Questo è implementato in Elisir usando cellule cons.

Le celle Contro sono semplici strutture di dati con un valore "sinistro" e un "giusto", o "testa" e una "coda".

A `|` il simbolo può essere aggiunto prima dell'ultimo elemento in una lista per notificare un elenco (improprio) con una data testa e coda. Quella che segue è una cella singola con `1` come la testa e `2` come la coda:

```
[1 | 2]
```

La sintassi standard di Elixir per un elenco è in realtà equivalente alla scrittura di una catena di celle di controllo nidificate:

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

La lista vuota `[]` viene usata come coda di una cella di controllo per rappresentare la fine di una lista.

Tutti gli elenchi in elisir sono equivalenti alla forma `[head | tail]` , dove la `head` è il primo elemento della lista e la `tail` è il resto della lista, meno la testa.

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

Usando il `[head | tail]` notation è utile per la corrispondenza dei pattern nelle funzioni ricorsive:

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

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

Liste di mappatura

`map` è una funzione nella programmazione funzionale che, data una lista e una funzione, restituisce una nuova lista con la funzione applicata a ciascuna voce in quella lista. In Elixir, la funzione `map/2` è nel modulo Enum .

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

Utilizzando la sintassi di acquisizione alternativa per le funzioni anonime:

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

Riferendosi ad una funzione con sintassi di cattura:

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

Operazioni di elenco concatenate utilizzando l'operatore di condotte:

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

Elenco delle comprensioni

L'elisir non ha loop. Invece di quelli per gli elenchi ci sono dei grandi moduli `Enum` e `List` , ma ci sono anche delle List comprehension.

Le comprensioni delle liste possono essere utili per:

- crea nuove liste

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

- filtraggio liste, utilizzando `guard` espressioni ma usarli senza `when` parola chiave.

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

- crea una mappa personalizzata, usando `into` parola chiave:

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

Esempio combinato

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

Sommario

Comprensioni delle liste:

- usa per `for..do` sintassi con guardie aggiuntive dopo le virgole e `into` parola chiave quando si restituisce una struttura diversa da quella degli elenchi, ad es. carta geografica.
- in altri casi restituire nuove liste
- non supporta gli accumulatori
- non può interrompere l'elaborazione quando viene soddisfatta una determinata condizione
- `guard` dichiarazioni di `guard` devono essere prima in ordine dopo `for` e prima di `do` o `into` simboli. L'ordine dei simboli non ha importanza

In base a questi vincoli, le Comprensioni delle liste sono limitate solo per un utilizzo semplice. Nei casi più avanzati l'utilizzo delle funzioni dei moduli `Enum` e `List` sarebbe l'idea migliore.

Lista differenza

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

`--` rimuove la prima occorrenza di un elemento nell'elenco a sinistra per ciascun elemento a destra.

Elenco membri

Utilizzare `in` operatore per verificare se un elemento è un membro di un elenco.

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

Convertire le liste in una mappa

Utilizzare `Enum.chunk/2` per raggruppare elementi in sotto-elenchi e `Map.new/2` per convertirlo in una mappa:

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

Darebbe:

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

Leggi elenchi online: https://riptutorial.com/it/elixir/topic/1279/elenchi

Capitolo 11: Erlang

Examples

Utilizzando Erlang

I moduli di Erlang sono disponibili come atomi. Ad esempio, il modulo di matematica di Erlang è disponibile come `:math` :

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

Ispeziona un modulo di Erlang

Usa `module_info` sui moduli di Erlang che desideri esaminare:

```
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>>]
```

Leggi Erlang online: https://riptutorial.com/it/elixir/topic/2716/erlang

Capitolo 12: ExDoc

Examples

introduzione

Per generare documentazione in formato `HTML` dagli attributi `@doc` e `@moduledoc` nel codice sorgente, aggiungi `ex_doc` e un processore markdown, in questo momento ExDoc supporta Earmark , Pandoc , Hoedown e Cmark , come dipendenze nel tuo file `mix.exs` :

```
# config/mix.exs

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

Se si desidera utilizzare un altro processore Markdown, è possibile trovare ulteriori informazioni nella sezione Modifica il Markdown .

Puoi utilizzare Markdown negli `@doc` Elixir `@doc` e `@moduledoc` .

Quindi, avvia i `mix docs` .

Una cosa da tenere a mente è che ExDoc consente i parametri di configurazione, come ad esempio:

```
  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
```

Puoi visualizzare maggiori informazioni su queste opzioni di configurazione con i `mix help docs`

Leggi ExDoc online: https://riptutorial.com/it/elixir/topic/3582/exdoc

Capitolo 13: ExUnit

Examples

Asserire eccezioni

Utilizzare `assert_raise` per verificare se è stata sollevata un'eccezione. `assert_raise` accetta `assert_raise` e una funzione da eseguire.

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

Avvolgi tutto il codice che desideri testare in una funzione anonima e `assert_raise` a `assert_raise` .

Leggi ExUnit online: https://riptutorial.com/it/elixir/topic/3583/exunit

Capitolo 14: FASCIO

Examples

introduzione

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

`:observer.start` apre l'interfaccia dell'osservatore GUI, che mostra la ripartizione della CPU, l'utilizzo della memoria e altre informazioni fondamentali per comprendere i modelli di utilizzo delle applicazioni.

Leggi FASCIO online: https://riptutorial.com/it/elixir/topic/3587/fascio

Capitolo 15: funzioni

Examples

Funzioni anonime

In Elixir, una pratica comune è usare le funzioni anonime. La creazione di una funzione anonima è semplice:

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

La sintassi generale è:

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

Per migliorare la leggibilità, puoi mettere le parentesi attorno agli argomenti:

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

Per richiamare una funzione anonima, chiamala con il nome assegnato e aggiungi `.` tra il nome e gli argomenti.

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

È possibile dichiarare funzioni anonime senza argomenti:

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

Utilizzando l'operatore di cattura

Per rendere più concise le funzioni anonime è possibile utilizzare l' **operatore di cattura** `&` . Ad esempio, invece di:

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

Tu puoi scrivere:

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

Con più parametri, utilizza il numero corrispondente a ciascun argomento, contando da `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
```

Corpi multipli

Una funzione anonima può anche avere più corpi (come risultato della corrispondenza del modello):

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

Quando si chiama una funzione con più corpi, l'elisir tenta di far corrispondere i parametri forniti con il corpo della funzione appropriata.

Elenchi di parole chiave come parametri di funzione

Utilizza gli elenchi di parole chiave per i parametri di stile "opzioni" che contengono più coppie chiave-valore:

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

Possiamo chiamare la funzione sopra in questo modo:

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

che è equivalente a:

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

I valori degli argomenti sono disponibili rispettivamente come `opts.pizza` e `opts.soda` .
In alternativa, potresti usare gli atomi: `opts[:pizza]` e `opts[:soda]` .

Funzioni nominate e funzioni private

Funzioni nominate

```
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
```

Funzioni private

```
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
```

Pattern Matching

L'elisir corrisponde una chiamata di funzione al suo corpo in base al valore dei suoi argomenti.

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

Qui, fattoriale di numeri positivi corrisponde alla seconda clausola, mentre `factorial(0)` corrisponde alla prima. (ignorando i numeri negativi per motivi di semplicità). L'elisir cerca di far corrispondere le funzioni dall'alto verso il basso. Se la seconda funzione è scritta sopra la prima, avremo un risultato inaspettato mentre si procede verso una ricorsione senza fine. Perché `factorial(0)` corrisponde a `factorial(n)`

Clausole di guardia

Le clausole di guardia ci consentono di controllare gli argomenti prima di eseguire la funzione. Le clausole di guardia sono solitamente preferite a `if` e `cond` causa della loro leggibilità e per rendere più semplice una determinata tecnica di ottimizzazione per il compilatore. La prima definizione di

funzione in cui tutte le protezioni corrispondono viene eseguita.

Ecco un esempio di implementazione della funzione fattoriale usando le protezioni e la corrispondenza del modello.

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

Il primo schema corrisponde se (e solo se) l'argomento è `0` . Se l'argomento non è `0` , la corrispondenza del modello fallisce e viene controllata la funzione successiva.

Quella seconda definizione di funzione ha una clausola di guardia: `when n > 0` . Ciò significa che questa funzione corrisponde solo se l'argomento `n` è maggiore di `0` . Dopo tutto, la funzione fattoriale matematica non è definita per gli interi negativi.

Se nessuna delle definizioni di funzione (incluse la corrispondenza del modello e le clausole di protezione) corrispondono, verrà generato un `FunctionClauseError` . Questo accade per questa funzione quando passiamo un numero negativo come argomento, poiché non è definito per i numeri negativi.

Notare che questo `FunctionClauseError` stesso non è un errore. Restituire `-1` o `0` o qualche altro "valore di errore" come è comune in alcune altre lingue nasconderebbe il fatto che hai chiamato una funzione indefinita, nascondendo la fonte dell'errore, creando probabilmente un enorme bug doloroso per un futuro sviluppatore.

Parametri predefiniti

È possibile passare i parametri predefiniti a qualsiasi funzione denominata utilizzando la sintassi: `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]
```

Cattura le funzioni

Usa `&` per acquisire funzioni da altri moduli. È possibile utilizzare le funzioni acquisite direttamente come parametri di funzione o all'interno di funzioni anonime.

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

Può essere reso più conciso usando `&` :

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

Catturare le funzioni senza passare alcun argomento richiede di specificare esplicitamente la propria appartenenza, ad esempio `&String.capitalize/1` :

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

Leggi funzioni online: https://riptutorial.com/it/elixir/topic/2442/funzioni

Capitolo 16: Gestione dello stato in elisir

Examples

Gestire un pezzo di stato con un agente

Il modo più semplice per avvolgere e accedere a un pezzo di stato è `Agent` . Il modulo consente di generare un processo che mantiene una struttura di dati arbitraria e consente di inviare messaggi per leggere e aggiornare quella struttura. Grazie a ciò l'accesso alla struttura viene serializzato automaticamente, in quanto il processo gestisce solo un messaggio alla volta.

```
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}
```

Leggi Gestione dello stato in elisir online: https://riptutorial.com/it/elixir/topic/6596/gestione-dello-stato-in-elisir

Capitolo 17: Installazione

Examples

Installazione di Fedora

```
dnf install erlang elixir
```

Installazione OSX

Su OS X e MacOS, Elixir può essere installato tramite i gestori di pacchetti comuni:

homebrew

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

macports

```
 $ sudo port install elixir
```

Installazione Debian / Ubuntu

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

# Update package index
sudo apt-get update

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

Installazione di Gentoo / Funtoo

Elixir è disponibile nel repository dei pacchetti principali.
Aggiorna l'elenco dei pacchetti prima di installare qualsiasi pacchetto:

```
emerge --sync
```

Questa è una fase di installazione:

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

Leggi Installazione online: https://riptutorial.com/it/elixir/topic/4208/installazione

Capitolo 18: Mappe e elenchi di parole chiave

Sintassi

- map =% {} // crea una mappa vuota
- map =% {: a => 1,: b => 2} // crea una mappa non vuota
- list = [] // crea una lista vuota
- list = [{: a, 1}, {: b, 2}] // crea un elenco di parole chiave non vuote

Osservazioni

Elixir fornisce due strutture dati associative: *mappe* e *elenchi di parole chiave* .

Le mappe sono il tipo di chiave-valore Elixir (chiamato anche dizionario o hash in altre lingue).

Gli elenchi di parole chiave sono tuple di chiave / valore che associano un valore a un determinato tasto. Vengono generalmente utilizzati come opzioni per una chiamata di funzione.

Examples

Creazione di una mappa

Le mappe sono il tipo di chiave-valore Elixir (chiamato anche dizionario o hash in altre lingue). Crei una mappa usando la sintassi `%w{}` :

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

Chiavi e valori possono essere di qualsiasi tipo:

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

Inoltre, puoi avere mappe con tipi misti per chiavi e valori ":

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

Quando tutte le chiavi di una mappa sono atomi, puoi utilizzare la sintassi delle parole chiave per comodità:

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

Creazione di un elenco di parole chiave

Gli elenchi di parole chiave sono tuple di chiave / valore, generalmente utilizzate come opzioni per una chiamata di funzione.

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

Gli elenchi di parole chiave possono avere la stessa chiave ripetuta più di una volta.

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

Chiavi e valori possono essere di qualsiasi tipo:

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

Differenza tra mappe e elenchi di parole chiave

Le mappe e gli elenchi di parole chiave hanno un'applicazione diversa. Ad esempio, una mappa non può avere due chiavi con lo stesso valore e non è ordinata. Viceversa, un elenco di parole chiave può essere un po 'difficile da utilizzare nella corrispondenza dei modelli in alcuni casi.

Ecco alcuni casi d'uso per mappe vs elenchi di parole chiave.

Utilizza gli elenchi di parole chiave quando:

- hai bisogno degli elementi da ordinare
- hai bisogno di più di un elemento con la stessa chiave

Utilizza le mappe quando:

- vuoi modellare la corrispondenza con alcune chiavi / valori
- non hai bisogno di più di un elemento con la stessa chiave
- ogni volta che non hai bisogno esplicitamente di un elenco di parole chiave

Leggi Mappe e elenchi di parole chiave online: https://riptutorial.com/it/elixir/topic/2706/mappe-e-elenchi-di-parole-chiave

Capitolo 19: Mescolare

Examples

Crea un'attività di mix personalizzato

```
# 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
```

Compilare ed eseguire:

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

Attività di missaggio personalizzata con argomenti della riga di comando

In un'implementazione di base, il modulo task deve definire una funzione `run/1` che accetta un elenco di argomenti. Ad esempio `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
```

Compilare ed eseguire:

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

alias

Elixir ti consente di aggiungere alias per i tuoi comandi mix. Bella cosa se vuoi salvarti digitando.

Apri `mix.exs` nel tuo progetto Elixir.

Innanzitutto, aggiungi la funzione `aliases/0` all'elenco di parole chiave restituito dalla funzione di `project` . *L'aggiunta `()` alla fine della funzione alias impedirà al compilatore di lanciare un avviso.*

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

Quindi, definisci la tua funzione `aliases/0` (ad esempio nella parte inferiore del tuo file `mix.exs`).

```
...

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

Ora puoi usare `$ mix go` per avviare il tuo server Phoenix (se stai usando un'applicazione Phoenix). E usa `$ mix trident` per dire mix per recuperare tutte le dipendenze, compilare ed eseguire il server.

Ottieni aiuto sulle missioni disponibili

Per elencare le attività di mix disponibili, utilizzare:

```
mix help
```

Per ottenere aiuto su un compito specifico usa l' `mix help task` ad esempio:

```
mix help cmd
```

Leggi Mescolare online: https://riptutorial.com/it/elixir/topic/3585/mescolare

Capitolo 20: metaprogrammazione

Examples

Genera test in fase di compilazione

```
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
```

Produzione:

```
.

  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
```

Leggi metaprogrammazione online: https://riptutorial.com/it/elixir/topic/4069/metaprogrammazione

Capitolo 21: Migliore debugging con IO.inspect ed etichette

introduzione

`IO.inspect` è molto utile quando si tenta di eseguire il debug delle catene del metodo di chiamata. Può diventare disordinato se lo si usa troppo spesso.

Dal momento che Elixir 1.4.0 l'opzione `label` di `IO.inspect` può aiutare

Osservazioni

Funziona solo con Elixir 1.4+, ma non posso ancora etichettarlo.

Examples

Senza etichette

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

Ciò si tradurrà in un sacco di output senza contesto:

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

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

Con etichette

utilizzare l'opzione `label` per aggiungere contesto può aiutare molto:

```
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}
```

Leggi Migliore debugging con IO.inspect ed etichette online:
https://riptutorial.com/it/elixir/topic/8725/migliore-debugging-con-io-inspect-ed-etichette

Capitolo 22: moduli

Osservazioni

Nomi dei moduli

In Elixir, i nomi di moduli come `IO` o `String` sono solo atomi sotto il cofano e vengono convertiti nel modulo `:"Elixir.ModuleName"` in fase di compilazione.

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

Examples

Elenca le funzioni o i macro di un modulo

La funzione `__info__/1` accetta uno dei seguenti atomi:

- `:functions` - Restituisce un elenco di parole chiave di funzioni pubbliche insieme alle loro entità
- `:macros` - Restituisce un elenco di parole chiave di macro pubbliche insieme alle loro entità

Per elencare le funzioni del modulo `Kernel` :

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

Sostituisci il `Kernel` con qualsiasi modulo di tua scelta.

Utilizzando i moduli

I moduli hanno quattro parole chiave associate da utilizzare in altri moduli: `alias` , `import` , `use` e `require` .

`alias` registrerà un modulo con un nome diverso (solitamente più breve):

```
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` renderà tutte le funzioni del modulo disponibili senza nome davanti a loro:

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

`use` consente a un modulo di iniettare codice nel modulo corrente - questo in genere viene fatto come parte di un framework che crea le proprie funzioni per fare in modo che il modulo confermi un comportamento.

`require` carichi macro dal modulo in modo che possano essere utilizzati.

Delega di funzioni a un altro modulo

Utilizzare `defdelegate` per definire le funzioni che delegano a funzioni con lo stesso nome definite in un altro modulo:

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


iex> Math.pi
3.141592653589793
```

Leggi moduli online: https://riptutorial.com/it/elixir/topic/2721/moduli

Capitolo 23: nodi

Examples

Elencare tutti i nodi visibili nel sistema

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

Collegamento di nodi sulla stessa macchina

Avvia due nodi denominati in due finestre di terminale:

```
>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)>
```

Connetti due nodi istruendo un nodo per la connessione:

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

I due nodi sono ora connessi e consapevoli l'uno dell'altro:

```
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"]
```

È possibile eseguire il codice su altri nodi:

```
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
```

Collegamento di nodi su macchine diverse

Avvia un processo denominato su un indirizzo IP:

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

Avvia un altro processo denominato su un indirizzo IP diverso:

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

Leggi nodi online: https://riptutorial.com/it/elixir/topic/2065/nodi

Capitolo 24: operatori

Examples

L'operatore del tubo

L'operatore di pipe `|>` prende il risultato di un'espressione a sinistra e lo alimenta come primo parametro di una funzione a destra.

```
expression |> function
```

Usa Pipe Operator per concatenare le espressioni e per documentare visivamente il flusso di una serie di funzioni.

Considera quanto segue:

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

Nell'esempio, `Oven.bake` arriva prima di `Ingredients.mix` , ma viene eseguito per ultimo. Inoltre, potrebbe non essere ovvio che `:temperature` è un parametro di `Oven.bake`

Riscrivi questo esempio usando l'operatore di pipe:

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

dà lo stesso risultato, ma l'ordine di esecuzione è più chiaro. Inoltre, è chiaro che `:temperature` è un parametro della chiamata `Oven.bake` .

Si noti che quando si utilizza Pipe Operator, il primo parametro per ciascuna funzione viene riposizionato prima dell'operatore Pipe e quindi la funzione chiamata sembra avere un parametro in meno. Per esempio:

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

equivale a:

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

Operatore del tubo e parentesi

Le parentesi sono necessarie per evitare l'ambiguità:

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

Dovrebbe essere scritto come:

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

Operatori booleani

Esistono due tipi di operatori booleani in elisir:

- operatori booleani (si aspettano che sia `true` o `false` come primo argomento)

```
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
```

Tutti gli operatori booleani genereranno `ArgumentError` se il primo argomento non sarà strettamente booleano, il che significa solo `true` o `false` (`nil` non è booleano).

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

- operatori booleani rilassati (funzionano con qualsiasi tipo, tutto ciò che né `false` né `nil` è considerato `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
```

Operatore `||` restituirà sempre il primo argomento se è vero (Elixir considera tutto tranne `nil` e `false` per essere vero nei confronti), altrimenti restituirà il secondo.

```
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
```

L'operatore `&&` e restituirà sempre il secondo argomento se è vero. Altrimenti tornerà rispettivamente agli argomenti, `false` o `nil` .

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

Sia `&&` che `||` sono operatori di cortocircuito. Eseguono solo il lato destro se il lato sinistro non è sufficiente per determinare il risultato.

Operatore `!` restituirà il valore booleano della negazione del termine corrente:

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

Un modo semplice per ottenere il valore booleano del termine selezionato è semplicemente raddoppiare questo operatore:

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

Operatori di confronto

Uguaglianza:

- valore uguaglianza `x == y` (`1 == 1.0 # true`)
- disuguaglianza di valore `x == y` (`1 != 1.0 # false`)
- uguaglianza rigorosa `x === y` (`1 === 1.0 # false`)
- disuguaglianza rigorosa `x === y` (`1 !== 1.0 # true`)

Confronto:

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

Se i tipi sono compatibili, il confronto utilizza l'ordinamento naturale. Altrimenti esiste una regola di confronto dei tipi generali:

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

Unisciti agli operatori

Puoi unire (concatenare) i binari (comprese le stringhe) e gli elenchi:

```
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"
```

Operatore "In"

`in` operatore consente di verificare se un elenco o un intervallo include un elemento:

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

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

Leggi operatori online: https://riptutorial.com/it/elixir/topic/1161/operatori

Capitolo 25: Ottenere aiuto nella console IEx

introduzione

IEx fornisce accesso alla documentazione di Elixir. Quando Elixir è installato sul tuo sistema, puoi avviare IEx ad es. Con il comando `iex` in un terminale. Quindi digitare il comando `h` sulla riga di comando IEx seguito dal nome della funzione preceduto dal nome del modulo, ad esempio `h List.foldr`

Examples

Elenco dei moduli e delle funzioni di Elixir

Per ottenere l'elenco dei moduli Elixir basta digitare

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

Premendo [TAB] compila automaticamente i nomi dei moduli e delle funzioni. In questo caso elenca tutti i moduli. Per trovare tutte le funzioni in un modulo, ad es `List` Uso `List`

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

Leggi Ottenere aiuto nella console IEx online: https://riptutorial.com/it/elixir/topic/10780/ottenere-aiuto-nella-console-iex

Capitolo 26: Ottimizzazione

Examples

Misura sempre per primo!

Questi sono consigli generali che in generale migliorano le prestazioni. Se il tuo codice è lento, è sempre importante indicarlo per capire quali parti sono lente. Indovinare **non** è **mai** abbastanza. Migliorare la velocità di esecuzione di qualcosa che richiede solo l'1% del tempo di esecuzione probabilmente non vale la pena. Cerca i lavandini di grande tempo.

Per ottenere numeri un po 'precisi, assicurati che il codice che stai ottimizzando sia eseguito per almeno un secondo durante la profilazione. Se spendi il 10% del tempo di esecuzione in quella funzione, assicurati che l'esecuzione completa del programma richieda almeno 10 secondi e assicurati di poter eseguire gli stessi esatti dati attraverso il codice più volte, per ottenere numeri ripetibili.

ExProf è semplice per iniziare.

Leggi Ottimizzazione online: https://riptutorial.com/it/elixir/topic/6062/ottimizzazione

Capitolo 27: Pattern matching

Examples

Funzioni di corrispondenza del modello

```
#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
```

Pattern matching su una mappa

```
%{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"}
```

Pattern matching su una lista

È inoltre possibile eseguire la corrispondenza di modelli su strutture dati di elisir come elenchi.

elenchi

Abbinare su una lista è abbastanza semplice.

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

Funziona facendo corrispondere i primi (o più) elementi nella lista sul lato sinistro di `|` (pipe) e il resto della lista alla variabile sul lato destro del `|` .

Possiamo anche abbinare su valori specifici di una lista:

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

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

Associazione di più valori consecutivi a sinistra di `|` è anche permesso:

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

Ancora più complesso: possiamo abbinare su un valore specifico e confrontarlo con una variabile:

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

Ottieni la somma di un elenco usando la corrispondenza del modello

```
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
```

Funzioni anonime

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

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

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

Le tuple

```
{ 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" }
```

Leggere un file

La corrispondenza del modello è utile per un'operazione come la lettura di file che restituisce una tupla.

Se il file `sample.txt` contiene `This is a sample text` , quindi:

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

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

Altrimenti, se il file non esiste:

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

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

Pattern che corrisponde alle funzioni anonime

```
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
```

Leggi Pattern matching online: https://riptutorial.com/it/elixir/topic/1602/pattern-matching

Capitolo 28: Polimorfismo in elisir

introduzione

Il polimorfismo è la fornitura di una singola interfaccia per entità di diversi tipi. Fondamentalmente, consente a tipi di dati diversi di rispondere alla stessa funzione. Quindi, la stessa funzione modella per diversi tipi di dati per ottenere lo stesso comportamento. Il linguaggio elisir ha `protocols` per implementare il polimorfismo in modo pulito.

Osservazioni

Se si desidera coprire tutti i tipi di dati, è possibile definire un'implementazione per `Any` tipo di dati. Infine, se hai tempo, controlla il codice sorgente di Enum e String.Char , che sono buoni esempi di polimorfismo nel nucleo di elisir.

Examples

Polimorfismo con protocolli

Implementiamo un protocollo di base che converte le temperature Kelvin e Fahrenheit in gradi Celsius.

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

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

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

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

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

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

Ora, abbiamo implementato i nostri convertitori per i tipi Kelvin e Fahrenheit. Facciamo alcune conversioni:

```
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"}
```

Proviamo a convertire qualsiasi altro tipo di dati che non ha implementazione per la funzione `to_celsius` :

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

Leggi Polimorfismo in elisir online: https://riptutorial.com/it/elixir/topic/9519/polimorfismo-in-elisir

Capitolo 29: Processi

Examples

Generare un processo semplice

Nell'esempio seguente, la funzione di `greet` all'interno del modulo `Greeter` viene eseguita in un processo separato:

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

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

Qui `#PID<0.122.0>` è l' *identificatore di processo* per il processo generato.

Invio e ricezione di messaggi

```
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}
```

Ricorsione e ricezione

La ricorsione può essere utilizzata per ricevere più messaggi

```
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 utilizzerà un'ottimizzazione della ricorsione in coda, purché la chiamata alla funzione sia l'ultima cosa che accade nella funzione come nell'esempio.

Leggi Processi online: https://riptutorial.com/it/elixir/topic/3173/processi

Capitolo 30: Programmazione funzionale in elisir

introduzione

Proviamo a implementare le funzioni di base degli ordini superiori come la mappa e riduciamo l'uso di Elisir

Examples

Carta geografica

Map è una funzione che prenderà una matrice e una funzione e restituirà una matrice dopo aver applicato quella funzione a ***ciascun elemento*** in quella lista

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

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

Copia incolla in `iex` ed esegui:

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

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

Ridurre

Riduci è una funzione che richiede un array, una funzione e un accumulatore e utilizza l' ***accumulatore come seme per avviare l'iterazione con il primo elemento per fornire l'accumulatore successivo e l'iterazione continua per tutti gli elementi dell'array*** (vedere sotto l'esempio)

```
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
```

Copia incolla lo snippet sopra riportato in iex:

1. Per aggiungere tutti i numeri in un array: `MyList.reduce [1,2,3,4], fn acc, element -> acc + element end, 0`
2. Per mutilare tutti i numeri in un array: `MyList.reduce [1,2,3,4], fn acc, element -> acc * element end, 1`

Spiegazione per esempio 1:

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

Filtra l'elenco usando riduci

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

Leggi Programmazione funzionale in elisir online:
https://riptutorial.com/it/elixir/topic/10186/programmazione-funzionale-in-elisir

Capitolo 31: protocolli

Osservazioni

Una nota sulle strutture

Invece di condividere l'implementazione del protocollo con le mappe, le strutture richiedono la propria implementazione del protocollo.

Examples

introduzione

I protocolli consentono il polimorfismo in elisir. Definire i protocolli con `defprotocol` :

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

Implementa un protocollo con `defimpl` :

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

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

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

Con le suddette implementazioni, possiamo fare:

```
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
```

I protocolli consentono di inviare qualsiasi tipo di dati, purché implementino il protocollo. Questo include alcuni tipi built-in come `Atom` , `BitString` , `Tuples` e altri.

Leggi protocolli online: https://riptutorial.com/it/elixir/topic/3487/protocolli

Capitolo 32: ruscello

Osservazioni

Gli stream sono componibili, pigri enumerabili.

A causa della loro pigrizia, i flussi sono utili quando si lavora con collezioni grandi (o addirittura infinite). Quando si concatenano molte operazioni con `Enum` , vengono creati elenchi intermedi, mentre `Stream` crea una ricetta di calcoli che vengono eseguiti in un secondo momento.

Examples

Concatenare più operazioni

`Stream` è particolarmente utile quando si desidera eseguire più operazioni su una raccolta. Questo perché `Stream` è pigro e fa solo un'iterazione (mentre `Enum` farebbe più iterazioni, per esempio).

```
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]
```

Qui, abbiamo incatenato 3 operazioni (`map` , `filter` e `take_every`), ma l'iterazione finale è stata eseguita solo dopo che `Enum.to_list` stato chiamato.

Quello che `Stream` fa internamente, è che attende fino a quando è richiesta una valutazione effettiva. Prima di ciò crea una lista di tutte le funzioni, ma una volta che la valutazione è necessaria, passa attraverso la raccolta una volta, eseguendo tutte le funzioni su ogni elemento. Questo lo rende più efficiente di `Enum` , che in questo caso farebbe 3 iterazioni, per esempio.

Leggi ruscello online: https://riptutorial.com/it/elixir/topic/2553/ruscello

Capitolo 33: sigilli

Examples

Costruisci un elenco di stringhe

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

Costruisci una lista di atomi

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

Sigilli personalizzati

È possibile creare sigilli personalizzati creando un metodo `sigil_X` dove X è la lettera che si desidera utilizzare (questa può essere solo una singola lettera).

```
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
```

L'argomento delle `options` è un binario degli argomenti forniti alla fine del sigillo, ad esempio:

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

Leggi sigilli online: https://riptutorial.com/it/elixir/topic/2204/sigilli

Capitolo 34: stringhe

Osservazioni

Una `String` in elisir è un binario codificato in `UTF-8` .

Examples

Converti in stringa

Usa Kernel.inspect per convertire qualsiasi cosa in stringa.

```
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\"}"
```

Ottieni una sottostringa

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

Dividere una stringa

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

Interpolazione a stringa

```
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."
```

Controlla se String contiene la sottostringa

```
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
```

Unisci le stringhe

Puoi concatenare stringhe in elisir usando l'operatore `<>` :

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

Per un `List` di stringhe, è possibile utilizzare `Enum.join/2` :

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

Leggi stringhe online: https://riptutorial.com/it/elixir/topic/2618/stringhe

Capitolo 35: Strutture dati

Sintassi

- [testa | tail] = [1, 2, 3, true] # uno può utilizzare la corrispondenza del modello per suddividere le celle. Assegna head a 1 e tail a [2, 3, true]
- % {d: val} =% {d: 1, e: true} # assegna val a 1; nessuna variabile d viene creata perché la d su lhs è in realtà solo un simbolo che viene utilizzato per creare il pattern% {: d => _} (nota che la notazione a razzo hash consente di avere simboli non come chiavi per le mappe come in rubino)

Osservazioni

Per quanto riguarda la struttura dei dati per noi qui ci sono alcune brevi osservazioni.

Se hai bisogno di una struttura dati dell'array se stai scrivendo molte liste di uso. Se invece farai molta lettura dovresti usare le tuple.

Per quanto riguarda le mappe, sono semplicemente come si fanno i negozi di valore chiave.

Examples

elenchi

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

Si noti che questi sono memorizzati in memoria come elenchi collegati. Questa è una serie di celle contro cui la testa (List.hd / 1) è il valore del primo elemento dell'elenco e la coda (List.tail / 1) è il valore del resto dell'elenco.

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

Le tuple

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

Le tuple sono l'equivalente di matrici in altre lingue. Sono memorizzati in modo contiguo nella memoria.

Leggi Strutture dati online: https://riptutorial.com/it/elixir/topic/1607/strutture-dati

Capitolo 36: Suggerimenti e trucchi

introduzione

Elixir Consigli e trucchi avanzati che fanno risparmiare tempo durante la codifica.

Examples

Creazione di sigilli personalizzati e documentazione

Ogni x sigil chiama la rispettiva sigil_x definizione

Definizione di sigilli personalizzati

```
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
```

Multiplo [OR]

Questo è solo l'altro modo di scrivere più condizioni OR. Questo non è l'approccio raccomandato perché in un approccio regolare quando la condizione è vera, interrompe l'esecuzione delle restanti condizioni che risparmiano il tempo di valutazione, diversamente da questo approccio che valuta tutte le condizioni prima nell'elenco. Questo è solo cattivo, ma buono per le scoperte.

```
# 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
```

Configurazione personalizzata iex - Decorazione iex

Copia il contenuto in un file e salva il file come .iex.exs nella tua ~ home directory e vedi la magia. Puoi anche scaricare il file QUI

```
# 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

)
```

Leggi Suggerimenti e trucchi online: https://riptutorial.com/it/elixir/topic/10623/suggerimenti-e-trucchi

Capitolo 37: Suggerimenti e trucchi della console IEx

Examples

Ricompilare il progetto con `ricompilare`

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

Vedi la documentazione con `h`

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

                                def last(list)

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

Examples

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

Ottieni valore dall'ultimo comando con `v`

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

Vedi anche: Ottieni il valore di una riga con `v`

Ottieni il valore di un comando precedente con `v`

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

Puoi ottenere una riga specifica passando l'indice della riga:

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

Puoi anche specificare un indice relativo alla riga corrente:

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

Il valore può essere riutilizzato in altri calcoli:

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

Se si specifica una riga non esistente, `IEx` genererà un errore:

```
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
```

Esci dalla console IEx

1. Usa Ctrl + C, Ctrl + C per uscire

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

2. Usa `Ctrl+ \` per uscire immediatamente

Vedi le informazioni con `i`

```
iex(1)> i :ok
Term
  :ok
Data type
  Atom
Reference modules
  Atom
iex(2)> x = "mystring"
"mystring"
iex(3)> i x
Term
  "mystring"
Data type
  BitString
Byte size
  8
```

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

Creazione di PID

Ciò è utile quando non hai memorizzato il PID da un comando precedente

```
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>
```

Prepara i tuoi alias quando avvii IEx

Se inserisci i tuoi alias più comuni in un file `.iex.exs` nella radice della tua app, IEx li caricherà automaticamente all'avvio.

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

Storia persistente

Per impostazione predefinita, la cronologia degli input dell'utente in `IEx` non viene mantenuta tra sessioni diverse.

`erlang-history` aggiunge il supporto della cronologia sia alla shell di Erlang che a `IEx` :

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

Ora puoi accedere ai tuoi precedenti input usando i tasti freccia su e giù, anche tra diverse sessioni `IEx` .

Quando la console Elixir è bloccata ...

A volte potresti accidentalmente eseguire qualcosa nella shell che finisce per aspettare per sempre, e quindi bloccare la shell:

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

In tal caso, premere Ctrl-g. Vedrai:

```
User switch command
```

Inserisci questi comandi nell'ordine:

- `k` (per uccidere il processo shell)
- `s` (per iniziare un nuovo processo di shell)
- `c` (per connettersi al nuovo processo shell)

Finirai con una nuova shell di Erlang:

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

Per avviare una shell Elixir, digitare:

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

(non dimenticare il punto finale!)

Quindi vedrai un nuovo processo di shell Elixir in arrivo:

```
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)>
```

Per uscire dalla modalità "in attesa di più input" (a causa di virgolette non **`#iex:break`** , parentesi ecc.) Digitare **`#iex:break`** , seguito da carriage return (▯):

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

iex(1)>
```

quanto sopra è particolarmente utile quando copia-incolla uno snippet relativamente grande trasforma la console in modalità "in attesa di più input".

rompere di espressione incompleta

Quando hai inserito qualcosa in IEx che si aspetta un completamento, come una stringa multilinea, IEx cambierà il prompt per indicare che è in attesa che tu finisca cambiando il prompt per avere un `iex` di sospensione (`...`) piuttosto che `iex` .

Se scopri che IEx ti aspetta per terminare un'espressione ma non sei sicuro di cosa debba terminare l'espressione, o semplicemente vuoi interrompere questa riga di input, digita `#iex:break` come input della console. Ciò farà sì che IEx lanci un `TokenMissingError` e annulli l'attesa di altri input, riportandoti a un input di console "di primo livello" standard.

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

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

Ulteriori informazioni sono disponibili nella documentazione IEx .

Carica un modulo o uno script nella sessione IEx

Se hai un file elisir; uno script o un modulo e vuoi caricarlo nella sessione IEx corrente, puoi utilizzare il metodo `c/1` :

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

Questo compilerà e caricherà il modulo in IEx e sarai in grado di chiamare tutti i suoi metodi pubblici.

Per gli script, eseguirà immediatamente il contenuto dello script:

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

Leggi Suggerimenti e trucchi della console IEx online:
https://riptutorial.com/it/elixir/topic/1283/suggerimenti-e-trucchi-della-console-iex

Capitolo 38: Suggerimenti per il debug

Examples

Debugging con IEX.pry / 0

Il debug con `IEx.pry/0` è abbastanza semplice.

1. `require IEx` nel tuo modulo
2. Trova la riga di codice che vuoi controllare
3. Aggiungi `IEx.pry` dopo la riga

Ora inizia il tuo progetto (es. `iex -S mix`).

Quando viene raggiunta la linea con `IEx.pry/0` il programma si interrompe e si ha la possibilità di ispezionare. È come un punto di interruzione in un debugger tradizionale.

Quando hai finito basta digitare `respawn` nella console.

```
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
```

Debugging con IO.inspect / 1

È possibile utilizzare IO.inspect / 1 come strumento per eseguire il debug di un programma di elisir.

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

Stamperà argomenti_1 e argomenti_2 alla console. Poiché `IO.inspect/1` restituisce il suo argomento, è molto facile includerlo nelle chiamate di funzione o nelle pipeline senza interrompere il flusso:

```
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))
```

Debug in pipe

```
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]
```

Fare leva nel tubo

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

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

```
iex(1)> Demo.foo
Request to pry #PID<0.117.0> at lib/demo.ex:11
```

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

Allow? [Yn] Y

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

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

Leggi Suggerimenti per il debug online: https://riptutorial.com/it/elixir/topic/2719/suggerimenti-per-il-debug

Capitolo 39: Tipi incorporati

Examples

Numeri

L'elisir viene fornito con **numeri interi** e **numeri in virgola mobile** . Un **intero letterale** può essere scritto nei formati decimale, binario, ottale ed esadecimale.

```
iex> x = 291
291

iex> x = 0b100100011
291

iex> x = 0o443
291

iex> x = 0x123
291
```

Poiché Elixir utilizza l'aritmetica bignum, **l'intervallo dell'intero è limitato solo dalla memoria disponibile sul sistema** .

I numeri in virgola mobile sono a doppia precisione e seguono le specifiche IEEE-754.

```
iex> x = 6.8
6.8

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

Nota che Elixir supporta anche il modulo esponenziale per i float.

```
iex> 1 + 1
2

iex> 1.0 + 1.0
2.0
```

Per prima cosa abbiamo aggiunto due numeri interi e il risultato è un numero intero. Successivamente abbiamo aggiunto due numeri in virgola mobile e il risultato è un numero in virgola mobile.

Dividere in Elixir restituisce sempre un numero in virgola mobile:

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

Allo stesso modo, se aggiungi, sottragga o moltiplichi un numero intero con un numero in virgola

mobile, il risultato sarà in virgola mobile:

```
iex> 40.0 + 2
42.0

iex> 10 - 5.0
5.0

iex> 3 * 3.0
9.0
```

Per la divisione in interi, si può usare la funzione `div/2` :

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

atomi

Gli atomi sono costanti che rappresentano un nome di qualche cosa. Il valore di un atomo è il suo nome. Il nome di un atomo inizia con i due punti.

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

Il nome di un atomo è unico. Due atomi con gli stessi nomi sono sempre uguali.

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

iex(2)> b = :atom
:atom

iex(3)> a == b
true

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

Booleans `true` e `false` , in realtà sono atomi.

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

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

Gli atomi sono memorizzati in una tabella speciale di atomi. È molto importante sapere che questa tabella non è raccolta dalla spazzatura. Quindi, se vuoi (o accidentalmente è un fatto) creare costantemente atomi, è una cattiva idea.

Binari e Bitstring

I binari in elixir sono creati usando il costrutto Kernel.SpecialForms << >> .

Sono uno strumento potente che rende Elixir molto utile per lavorare con i protocolli e le codifiche binari.

I binari e le stringhe di bit vengono specificati utilizzando un elenco delimitato da virgole di numeri interi o valori variabili, con la dicitura "<<" e ">>". Sono composti da 'unità', un raggruppamento di bit o un raggruppamento di byte. Il raggruppamento predefinito è un singolo byte (8 bit), specificato utilizzando un numero intero:

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

Le stringhe di elisir convertono anche direttamente in binari:

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

Puoi aggiungere "specificatori" a ciascun "segmento" di un binario, consentendoti di codificare:

- Tipo di dati
- Taglia
- endianness

Questi specificatori sono codificati seguendo ogni valore o variabile con l'operatore "::":

```
<<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
```

I tipi di dati disponibili che puoi utilizzare sono:

- numero intero
- galleggiante
- bit (alias per bitstring)
- bitstring
- binario
- byte (alias per binario)
- utf8
- UTF16
- UTF-32

Tieni presente che quando si specifica la 'dimensione' del segmento binario, varia in base al 'tipo' scelto nello specificatore di segmento:

- intero (predefinito) 1 bit
- galleggiante 1 bit
- binario 8 bit

Leggi Tipi incorporati online: https://riptutorial.com/it/elixir/topic/1774/tipi-incorporati

Capitolo 40: Unisci le stringhe

Examples

Utilizzo dell'interpolazione stringa

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

Uso dell'elenco IO

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

Ciò aumenterà le prestazioni come stringhe non duplicate in memoria.
Metodo alternativo:

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

Utilizzando Enum.join

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

Leggi Unisci le stringhe online: https://riptutorial.com/it/elixir/topic/9202/unisci-le-stringhe

Capitolo 41: uso di base delle clausole di guardia

Examples

usi di base delle clausole di guardia

In Elixir, è possibile creare più implementazioni di una funzione con lo stesso nome e specificare regole che verranno applicate ai parametri della funzione *prima di chiamare la funzione* per determinare quale implementazione eseguire.

Queste regole sono contrassegnate dalla parola chiave `when` , e vanno tra il `def function_name(params)` e il `do` nella definizione della funzione. Un esempio banale:

```
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
```

Supponiamo che `Math.is_even(2)` con questo esempio. Esistono due implementazioni di `is_even` , con diverse clausole di guardia. Il sistema li guarderà in ordine ed eseguirà la prima implementazione dove i parametri soddisfano la clausola di guardia. Il primo specifica che `num === 1` che non è vero, quindi passa a quello successivo. Il secondo specifica che `num === 2` , che è vero, quindi questa è l'implementazione che viene utilizzata e il valore restituito sarà `true` .

Cosa succede se `Math.is_odd(1)` ? Il sistema esamina la prima implementazione e vede che poiché `num` è `1` la clausola di guardia della prima implementazione è soddisfatta. Utilizzerà quindi tale implementazione e restituirà `true` , senza preoccuparsi di esaminare altre implementazioni.

Le guardie sono limitate nei tipi di operazioni che possono eseguire. La documentazione di Elixir elenca tutte le operazioni consentite ; in poche parole consentono confronti, matematica, operazioni binarie, controllo del tipo (ad es. `is_atom`) e una manciata di piccole funzioni di convenienza (ad es. `length`). È possibile definire clausole di protezione personalizzate, ma è necessario creare macro ed è preferibile una guida più avanzata.

Nota che le guardie non lanciano errori; vengono considerati normali fallimenti della clausola di guardia e il sistema passa alla fase successiva. Se si scopre che si sta ottenendo `(FunctionClauseError) no function clause matching` quando si chiama una funzione protetta con params che si prevede di lavorare, potrebbe essere che una clausola di guardia che si prevede funzioni genererà un errore che viene ingerito.

Per vederlo da solo, crea e poi chiama una funzione con una guardia che non ha senso, come questa che prova a dividere per zero:

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

La chiamata a `BadMath.divide("anything")` fornirà l'errore un po 'inutile `(FunctionClauseError) no function clause matching in BadMath.divide/1` - mentre se si fosse tentato di eseguire `"anything" / 0` direttamente, si otterrebbe un risultato più utile errore: `(ArithmeticError) bad argument in arithmetic expression` errato `(ArithmeticError) bad argument in arithmetic expression` .

Leggi uso di base delle clausole di guardia online: https://riptutorial.com/it/elixir/topic/6121/uso-di-base-delle-clausole-di-guardia

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con Elixir Language	alejosocorro, Andrey Chernykh, Ben Bals, Community, cwc, Delameko, Douglas Correa, helcim, I Am Batman, JAlberto, koolkat, leifg, MattW., rap-2-h, Simone Carletti, Stephan Rodemeier, Vinicius Quaiato, Yedhu Krishnan, Zimm i48
2	.Gitignore di base per il programma di elisir	Yos Riady
3	Compito	mario
4	comportamenti	Yos Riady
5	Condizionali	Andrey Chernykh, evuez, javanut13, Musfiqur Rahman, Paweł Obrok
6	costanti	ibgib
7	doctests	aholt, milmazz, Philippe-Arnaud de MANGOU, Yos Riady
8	Ecto	fgutierr, Philippe-Arnaud de MANGOU, toraritte
9	elenchi	Ben Bals, Candy Gumdrop, emoragaf, PatNowak, Sheharyar, Yos Riady
10	Erlang	4444, Yos Riady
11	ExDoc	milmazz, Yos Riady
12	ExUnit	Yos Riady
13	FASCIO	Yos Riady
14	funzioni	Andrey Chernykh, cwc, Dair, Eiji, Filip Haglund, PatNowak, rainteller, Simone Carletti, Stephan Rodemeier, Yedhu Krishnan, Yos Riady
15	Gestione dello stato in elisir	Paweł Obrok
16	Installazione	cwc, Douglas Correa, Eiji, JAlberto, MattW.
17	Mappe e elenchi di parole chiave	Sam Mercier, Simone Carletti, Yos Riady

18	Mescolare	4444, helcim, rainteller, Slava.K, Yos Riady
19	metaprogrammazione	4444, Paweł Obrok
20	Migliore debugging con IO.inspect ed etichette	leifg
21	moduli	Alex G, javanut13, Yos Riady
22	nodi	Yos Riady
23	operatori	alxndr, Andrey Chernykh, Dair, Gazler, Mitkins, nirev, PatNowak
24	Ottenere aiuto nella console IEx	helcim
25	Ottimizzazione	Filip Haglund, legoscia
26	Pattern matching	Alex Anderson, Dair, Danny Rosenblatt, evuez, Gabriel C, gmile, Harrison Lucas, javanut13, Oskar, PatNowak, theIV, Thomas, Yedhu Krishnan
27	Polimorfismo in elisir	mustafaturan
28	Processi	Alex G, Yedhu Krishnan
29	Programmazione funzionale in elisir	Dinesh Balasubramanian
30	protocolli	Yos Riady
31	ruscello	Oskar
32	sigilli	javanut13, Yos Riady
33	stringhe	Alex G, Sheharyar, Yos Riady
34	Strutture dati	Sam Mercier, Simone Carletti, Stephan Rodemeier, Yos Riady
35	Suggerimenti e trucchi	Ankanna
36	Suggerimenti e trucchi della console IEx	alxndr, Cifer, fahrradflucht, legoscia, mudasobwa, muttonlamb, PatNowak, Paweł Obrok, sbs, Sheharyar, Simone Carletti, Stephan Rodemeier, Uniaika, Vincent, Yos Riady
37	Suggerimenti per il debug	javanut13, Paweł Obrok, Pfitz, Philippe-Arnaud de MANGOU, sbs

38	Tipi incorporati	Andrey Chernykh, Arithmeticbird, Oskar, TreyE, Vinicius Quaiato
39	Unisci le stringhe	Agung Santoso
40	uso di base delle clausole di guardia	alxndr