

Shibuya.ex #1

Elixir ご紹介

Naoya Ito

Kaizen Platform, Inc.

8/25 2015



KAIZEN PLATFORM

こんにちは、残念な 日本のWeb技術者です



アジェンダ

- 話すこと

- Elixir のざっくり特徴
- 言語的な機能
- OTP 周りの概要

- 話さないこと

- 細かいシンタックス … 参考資料をどうぞ



Elixir、ざっくり特徴



Elixir

- Erlang VM の上で動く
- 動的型付けな関数型言語
- こなれたパッケージ管理システム Mix
- 軽量プロセス、アクターによる並行処理
- OTP



```
defmodule WeatherClient do
  def get do
    HTTPoison.start
    HTTPoison.get!(
      "http://api.openweathermap.org/data/2.5/weather?q=Tokyo,jp"
    ) |> process_response
  end

  def process_response(%{status_code: 200, body: body}) do
    body
    |> Poison.decode!
    |> extract_weather
  end

  def extract_weather(%{"weather" => weather}), do: weather
end

Enum.each WeatherClient.get, fn(w) ->
  IO.puts w["main"]
end
```



Ruby?

- 一見すると Ruby っぽい
- 実際それほど Ruby っぽくない
- むしろ Erlang/OTP
 - シンタックスが馴染みやすい Erlang/OTP と捉えたほうが良い



なぜ Elixir?

- ランタイムが強力
 - アクターモデルによる並行処理基盤を言語 + ランタイムで標準搭載

というわけで、重要なのは
Ruby みたい~ではなく、
Erlang/OTP の上に乗って
るってとこ



関数型言語としての Elixir



動的型付けの関数型言語

```
defmodule Fib do
  def of(0), do: 0
  def of(1), do: 1
  def of(n), do: Fib.of(n-1) + Fib.of(n-2)
end
```

```
IO.inspect Fib.of(20)
```

```
fibs = Stream.unfold({0,1}, fn {f1, f2} ->
  {f1, {f2, f1 + f2}}
end)
```

```
fibs |> Enum.take(10) |> IO.inspect
```



foldl(list, acc, function)

Specs:

```
foldl([elem], acc, (elem, acc -> acc)) :: acc when elem: var, acc: var
```

Folds (reduces) the given list to the left with a function. Requires an accumulator.

Examples

```
iex> List.foldl([5, 5], 10, fn (x, acc) -> x + acc end)
```

```
20
```

```
iex> List.foldl([1, 2, 3, 4], 0, fn (x, acc) -> x - acc end)
```

```
2
```

[Source](#)

foldr(list, acc, function)

Specs:

```
foldr([elem], acc, (elem, acc -> acc)) :: acc when elem: var, acc: var
```

Folds (reduces) the given list to the right with a function. Requires an accumulator.

Examples

```
iex> List.foldr([1, 2, 3, 4], 0, fn (x, acc) -> x - acc end)
```

```
-2
```

[Source](#)

関数型言語としての Elixir

- 第一級関数、高階関数
- パターンマッチ
- 不変なデータ型
- 再帰 (末尾呼び出し最適化あり)



ないもの

- (では)ない
 - 静的型付け
 - 純粋関数型 (副作用あり)
 - 部分適用 / カーリー化
 - Option 型 (Maybe)
- あるよ
 - リスト内包表記
 - 遅延評価、無限リスト (※言語全体ではない)



パターンマッチ

- Elixir の文法の基礎になってる機能
 - 関数型言語では結構みる・・・ Haskell とか
 - if 文などを使わない宣言的な記述に貢献
- 値のパターンを記述し、値と照合する
 - パターンと値がマッチしたら、何がしかが行われる
 - = は実はパターンマッチ演算子



パターンマッチ

```
result = case File.read("/etc/hosts" ) do
  {:ok, res} -> res
  {:error, :enoent} -> "oh it isn't here"
  {:error, :eaccess} -> "you can't read it"
  _ -> "?"
end
```

http://www.slideshare.net/Joe_noh/elixir-01



パターンマッチ + 再帰

リストが空のとき
(停止条件)

```
defmodule MyMath do
  def count(list), do: count(list, 0)
  defp count([], acc), do: acc
  defp count([_head|tail], acc) do
    count(tail, acc + 1)
  end
end
```

リストの先頭要素が `_head`
を束縛、残りが `tail` を束縛



パターンマッチのユースケース

- 変数束縛
- データ構造の分解
- リスト処理 (w/ 再帰)
- case 文
- パターン毎の関数定義
 - 値による関数の選択



|> … パイプライン演算子

```
1..10  
|> Enum.map(fn(x) -> x * x end)  
|> Enum.filter(fn(x) -> rem(x, 2) == 0 end)  
|> Enum.sum  
|> to_string  
|> IO.puts
```

関数を繋げて書いて
気持ちいい

関数の戻り値を、次の第一引数に渡す。

(F#インスパイアらしい)



```
import Ecto.Query
```

```
alias KaizenShoten.Repo
```

```
alias KaizenShoten.Book
```

Book

```
|> where([book], like(book.title, "%Ruby%"))  
|> order_by([book], desc: book.id)  
|> Repo.all  
|> Repo.preload(:author)  
|> Enum.map(fn(book) -> book.author.name end)  
|> IO.inspect
```

小さな関数をパイプライン
で繋げていくのが Elixir 流

(うーむ、Maybe 欲しい…)



不変なデータ型

- データ型は不変
 - String, List, Tuple, Map, HashDict, HashSet ...
- 要するに破壊的操作が不可能ってだけ
- `dict2 = Dict.put(dict, :foo, 3)`
 - `dict` は不変、更新された `dict2`



for, while がない

- 再帰を使うか (末尾呼び出し最適化)
- さもなくばルーク、Enum を使え!

```
iex> [1,2,3] |> Enum.each fn(x) -> IO.puts(x) end  
1  
2  
3  
:ok
```

```
iex> [1,2,3] |> Enum.reduce(0, fn(x, acc) -> x + acc end)  
6
```



リスト内包表記

- Haskell 等でおなじみの
 - `[x | x <- xs, x < p]`
- Elixir では
 - `for x <- xs, x < p, do: x`

正直これに関しては
Erlang そのままで良かったのでは… (個人の感想)

ピタゴラス数を探す

$A^2 + B^2 = C^2$
を満たす A, B, C の整数の
組を見つける

```
defmodule Pythag do
  def pythag(n) do
    for a <- 1..n,
        b <- 1..n,
        c <- 1..n,
        a + b + c <= n, a*a + b*b === c*c, do: {a, b, c}
  end
end
```

```
IO.inspect Pythag.pythag(16)
# [{3, 4, 5}, {4, 3, 5}]
```

```
IO.inspect Pythag.pythag(30)
# [{3, 4, 5}, {4, 3, 5}, {5, 12, 13}, {6, 8, 10}, {8, 6, 10}, {12, 5, 13}]
```


Stream

- 合成可能で遅延評価な Enumerables
 - Enum は Greedy
 - Stream は Lazy
- 無限リストや遅延評価が欲しいときはこれ



Range を Stream に変換、
パイプライン演算子で
filter と map を合成

```
require Integer
1..1000000
|> Stream.filter(fn(x) -> Integer.is_even(x) end)
|> Stream.map(fn(x) -> "This is number #{x}" end)
|> Enum.take(2)
|> IO.inspect
```

take(2) したところで初めて
評価 → 2件のみ計算
(遅延評価)



フィボナッチストリーム

Stream.unfold はアキュム
レータに計算結果を積んで再
帰的に関数を適用

```
fibs = Stream.unfold({1, 1}, fn {a, b} ->
  {a, {b, a + b}}
end)
fibs |> Enum.take(10) |> IO.inspect
# [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Enum.take(10) したところ
で初めて計算



ポリモーフィズム

- ポリモーフィズムの実現
 - Java … インタフェースなどの上位の型で
 - Ruby … ダックタピングで
 - Haskell … 多相型で
 - Elixir … プロトコルで



プロトコル

```
defprotocol Blank do
  def blank?(data)
end
```

```
defimpl Blank, for: Integer do
  def blank?(_), do: false
end
```

```
defimpl Blank, for: List do
  def blank?([]), do: true
  def blank?(_), do: false
end
```

異なるデータ型に同じ名前の関数を適用。型に応じて振る舞いが変わる

```
Blank.blank?(0) |> IO.inspect # false
Blank.blank?([]) |> IO.inspect # true
```

プロトコルの良い利用例

```
defimpl Poison.Encoder, for: MyApp.Page do
  def encode(page, _options) do
    %{
      title: page.title,
      body: page.body
    } |> Poison.Encoder.encode([])
  end
end
```

<http://blog.drewolson.org/building-an-elixir-web-app/>



Elixir らしいコード

- パターンマッチを積極的に使う
- 副作用は可能なら避ける / 分離
- Enum 等の関数を組み合わせて宣言的に書く
- パイプライン演算子 `|>` 使い単一の役割の小さな関数を繋げる



Erlang/OTP と Elixir



Erlang VM の上で動く

Elixir

OTP

BEAM (Erlang VM)



軽量プロセス

- Erlang VM 内の実行コンテキストの単位
 - OS のプロセスではない
- 1プロセス 300ワード程度。超軽量
 - 1ノード内で数百～数千プロセスとか平気で使いまくる
 - 4,00,000 プロセスで～とかそういう話も聞く



軽量プロセスと並行

- 軽量プロセスは VM でスケジューリングされて実行される
- VM はスレッドプール (確か) で実装されている
 - 軽量プロセスでの並行処理はマルチコアで使える

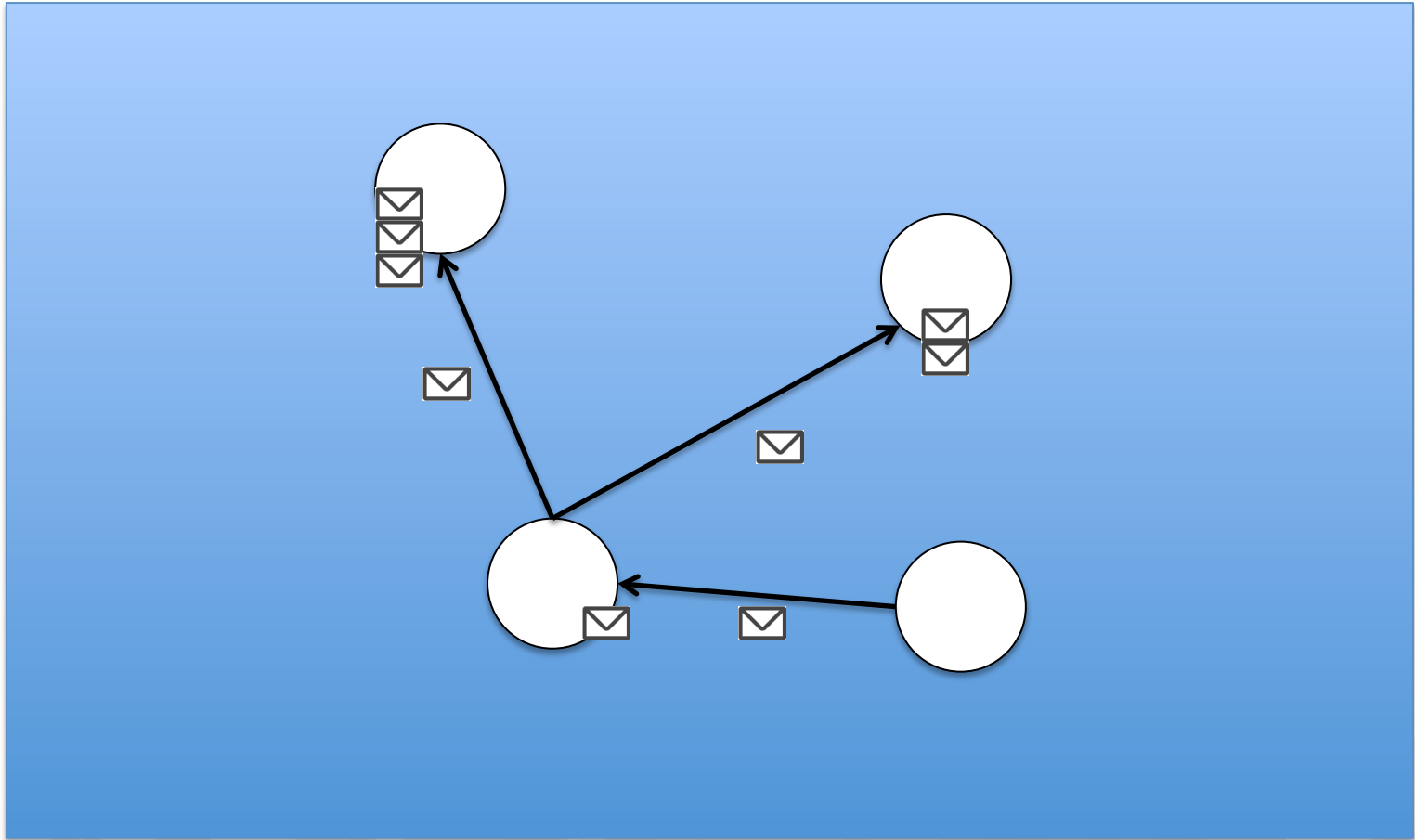


アクターモデル

- プロセス間通信はメッセージパッシング
 - プロセスの中に「メールボックス」
 - そこにメッセージを送る (send)
 - メッセージには任意の値を添付可
 - プロセスはそれを受信待ち (receive)
 - 値は必ず**コピーされる** (共有されない)
 - データを共有しない ⇒ ロックがいらぬい



アクターモデル



```
defmodule SumProcess do
  def loop do
    # receive/0 でメッセージを受信
    # 本体は受け取ったメッセージのパターンマッチ
    receive do
      {:sum, list} ->
        IO.puts Enum.sum(list)
        loop()
    end
  end
end
```

```
# spawn/3 でモジュールを軽量プロセスで実行
# 戻り値でメッセージの宛先 pid を取得
pid = spawn SumProcess, :loop, []
```

```
# プロセスにリストをメッセージで送る
send(pid, {:sum, [1, 2, 3, 4, 5]})
```

OTP

- プロセス周りの標準ライブラリ/フレームワーク群
- Erlang の最大の資産
 - アクターモデル + Erlang/OTP は など他言語へ大きな影響を与えている
 - 例: Scala の Akka



OTPビヘイビア (GenServer)

```
defmodule SumProcess do
  use GenServer
  def start_link do
    GenServer.start_link(__MODULE__, nil)
  end

  def sum(pid, list) do
    GenServer.call(pid, {:sum, list})
  end

  def handle_call({:sum, list}, _, _) do
    {:reply, Enum.sum(list), nil}
  end
end

{:ok, pid} = SumProcess.start_link
SumProcess.sum(pid, [1,2,3,4,5])
|> IO.inspect
```

軽量プロセスにパターン
(ビヘイビア) に乗っかる
だけでサーバを作れる。
(要はフレームワーク)

OTPビヘイビア (Task)

```
defmodule Fib do
  def of(0), do: 0
  def of(1), do: 1
  def of(n), do: Fib.of(n-1) + Fib.of(n-2)
end
```

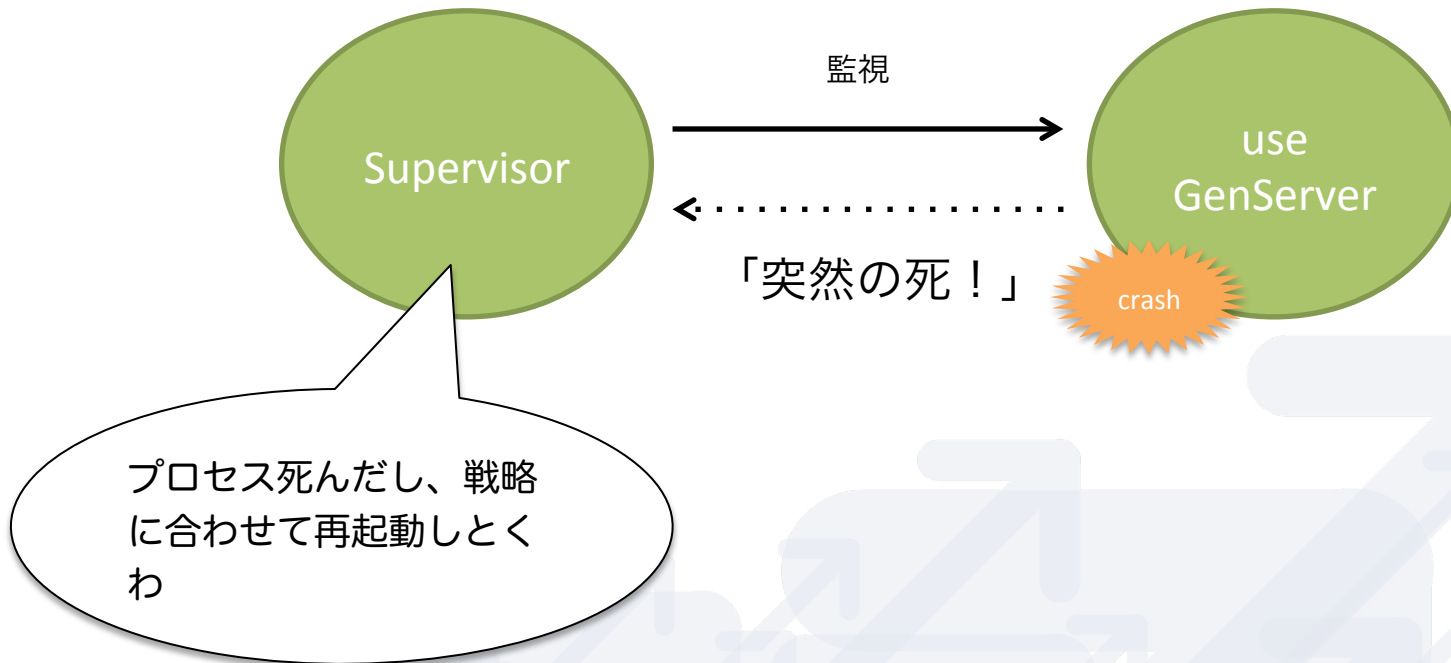
```
IO.inspect Fib.of(20)
```

```
task = Task.async(fn -> Fib.of(20) end)
result = Task.await(task)
```

```
IO.inspect(result)
```

async/await の裏ではア
クターが処理を並列化す
る

Supervisor

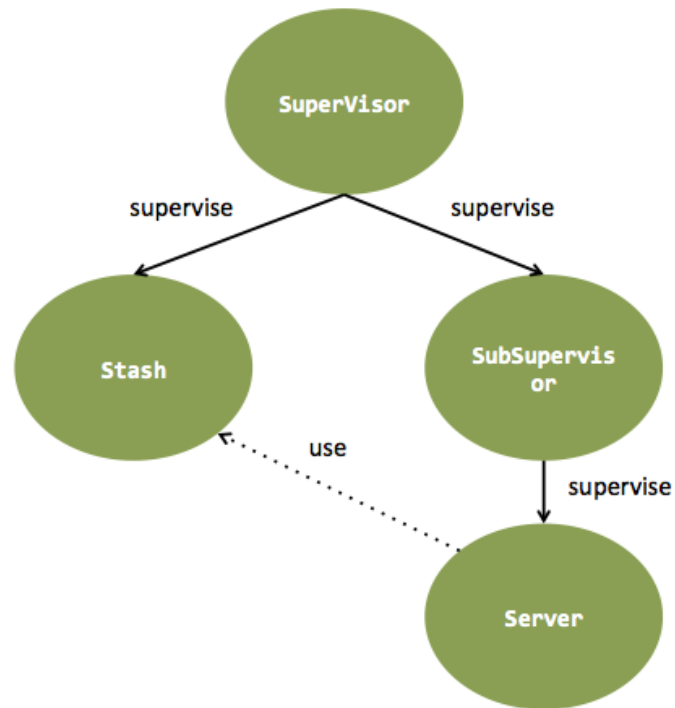


Let it crash

- 失敗に備えない。例外を捕まえない
- Supervisor で監視しておいて、そのまま起こすなりパラメータ変えてリトライさせるなり…



Supervision Tree



<http://qiita.com/naoya@github/items/ad18b49e9ed56a72cab6>

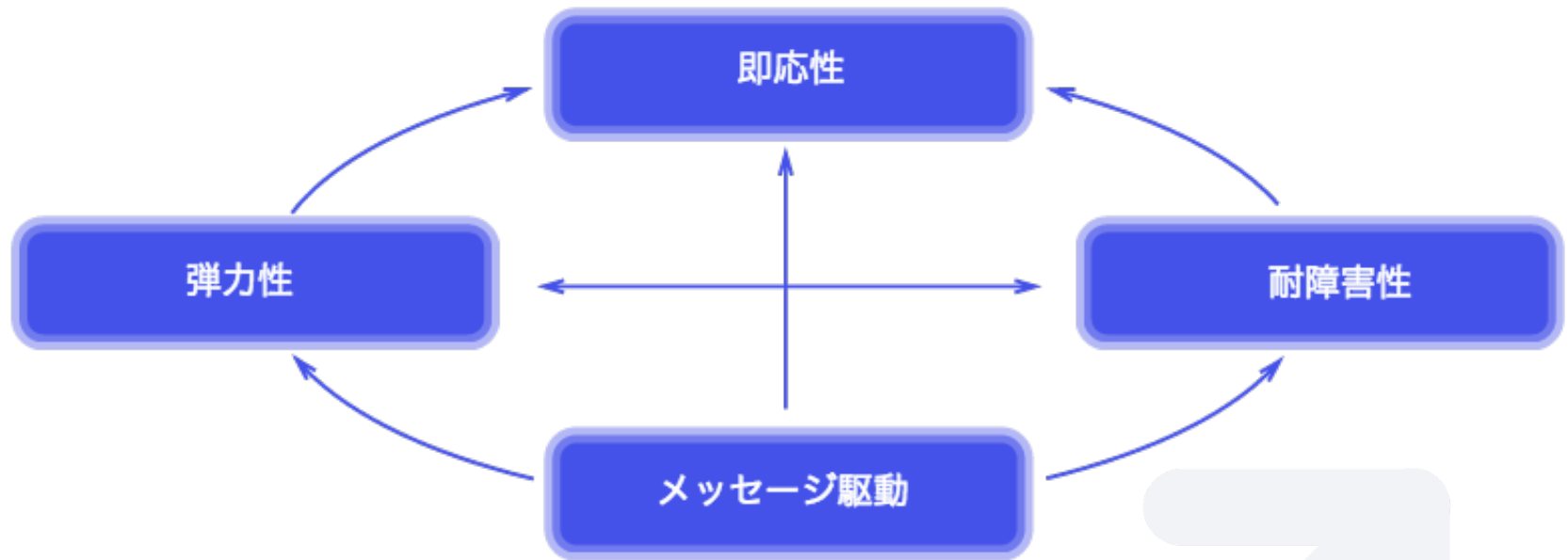


語りたかったが多分時間がない！

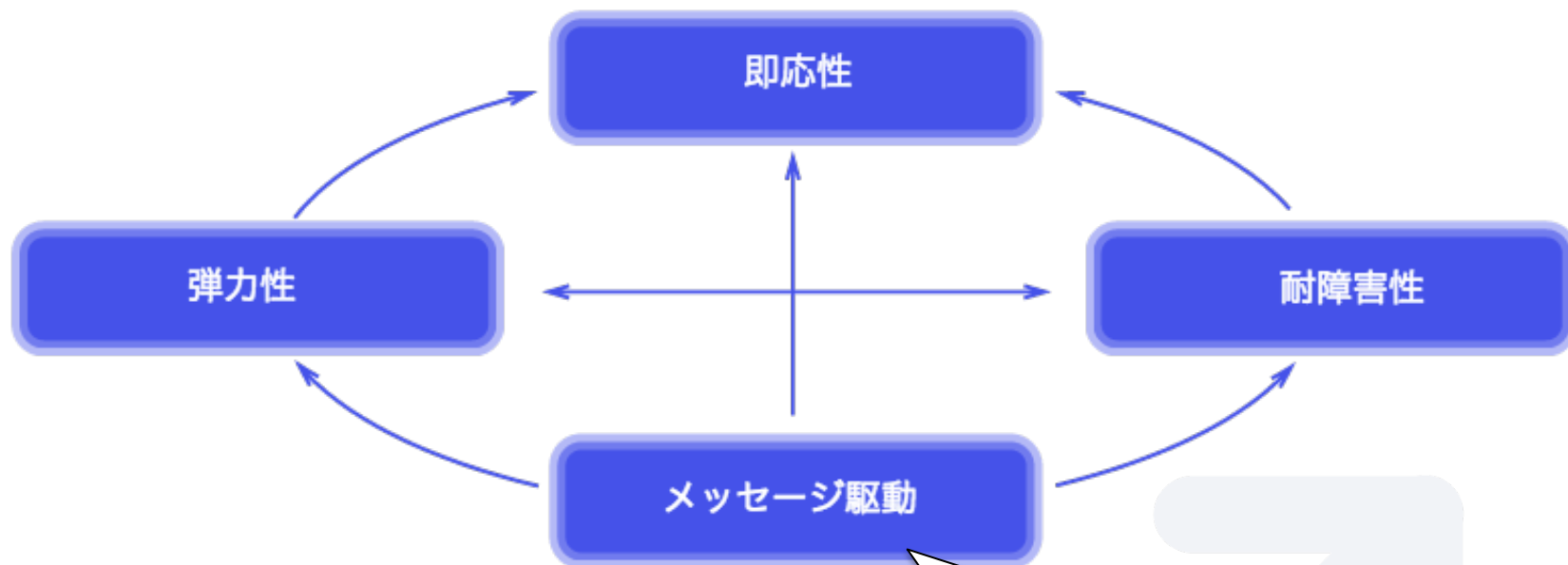
- 軽量プロセスはシステムの色々なデザインと相乗効果を働かせているよ
 - パターンマッチで宣言的に関数を選択することと Let it crash は相性が良いよ
 - プロセスのアドレスは透過性があって、他のノードへのメッセージパッシングも一緒だよ
 - 簡単にスケールするよ
 - 軽量プロセス単位で GC するからメモリいらなくなったらプロセスごと捨てれば良いよ!



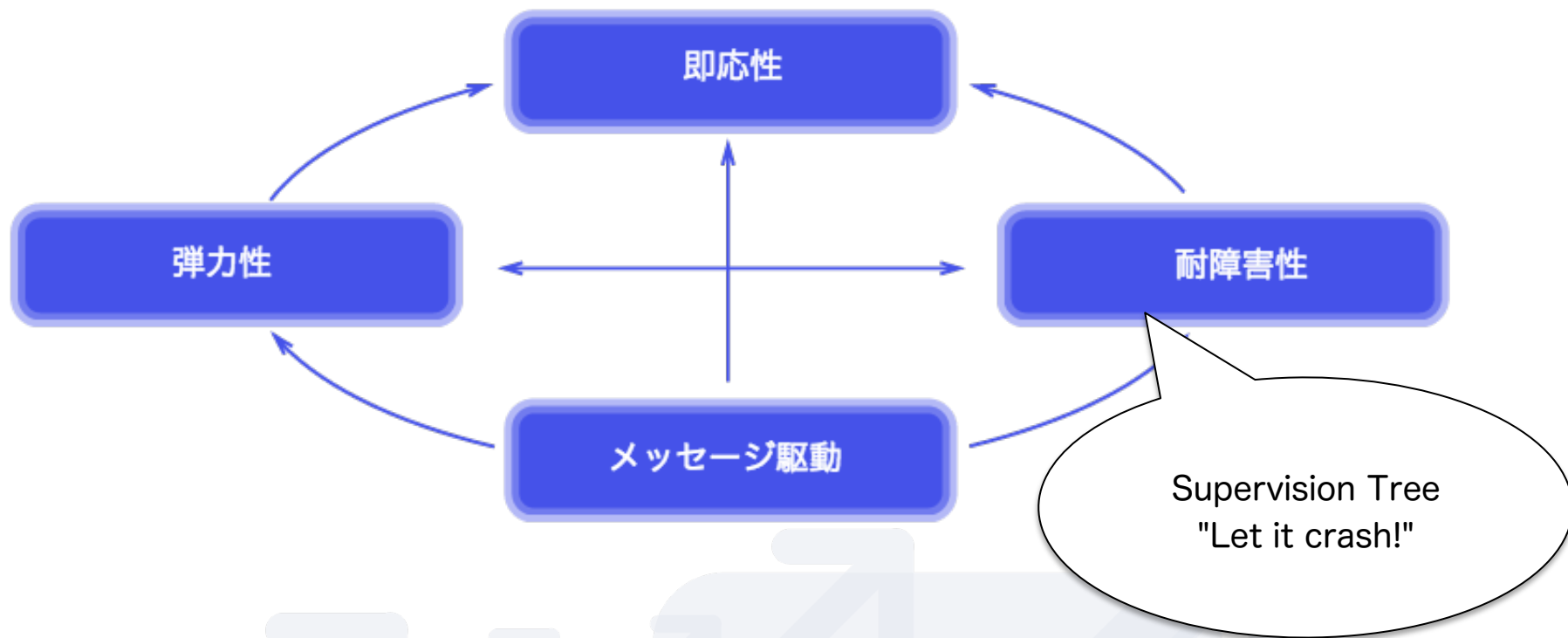
Reactive

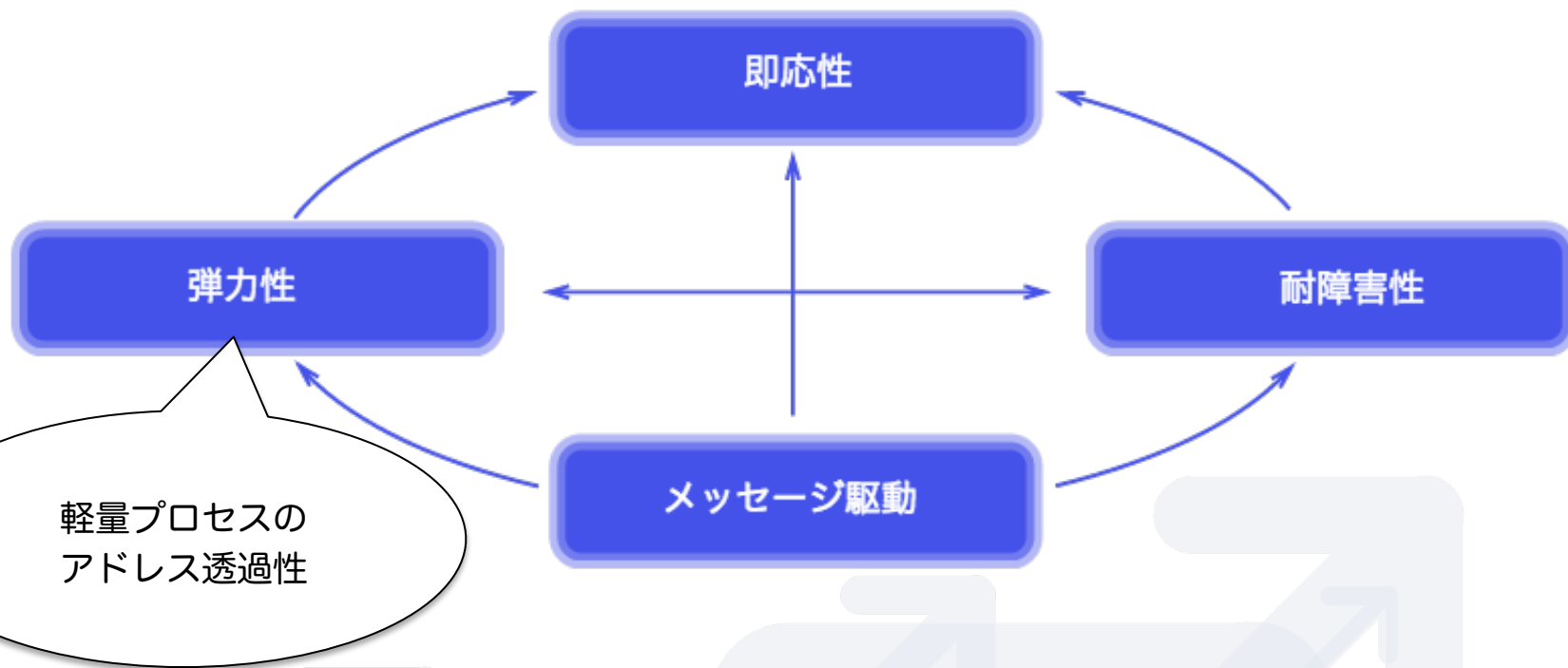


Reactive Manifesto (リアクティブ宣言)
<http://www.reactivemanifesto.org/ja>



アクターによる
メッセージパッシング





Erlang との相互互換性

- Erlang と Elixir は相互互換
 - Erlang のライブラリを Elixir で使える
 - Elixir のを Erlang でも使える

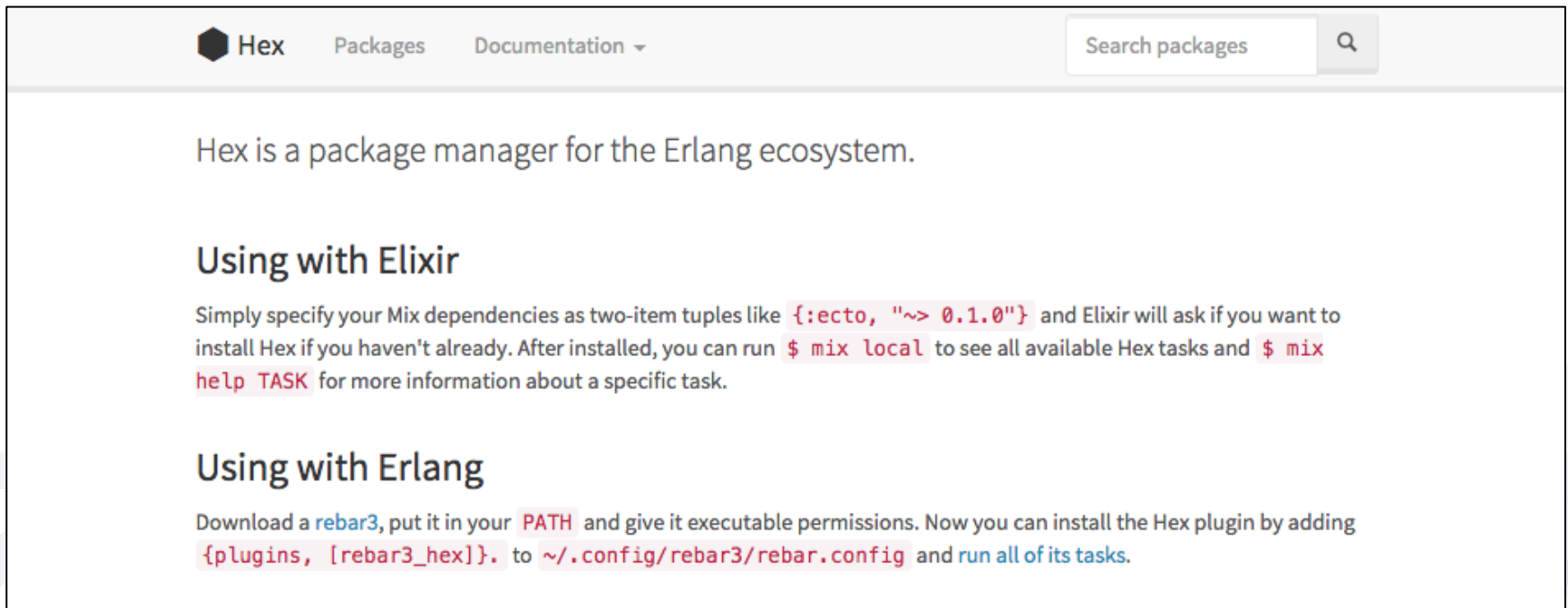


ほか



Mix + Hex

- ビルドツール & パッケージ管理
 - Ruby の Bundler + rubygems
 - Node.js の npm

A screenshot of the Hex website homepage. The header includes the Hex logo, navigation links for 'Packages' and 'Documentation', and a search bar labeled 'Search packages'. The main content area features a heading 'Hex is a package manager for the Erlang ecosystem.', followed by a section titled 'Using with Elixir' which includes a code snippet for specifying dependencies: `{:ecto, "~> 0.1.0"}`. Below that is a section titled 'Using with Erlang' which includes a code snippet for installing the Hex plugin: `{plugins, [rebar3_hex]}`.

Hex is a package manager for the Erlang ecosystem.

Using with Elixir

Simply specify your Mix dependencies as two-item tuples like `{:ecto, "~> 0.1.0"}` and Elixir will ask if you want to install Hex if you haven't already. After installed, you can run `$ mix local` to see all available Hex tasks and `$ mix help TASK` for more information about a specific task.

Using with Erlang

Download a `rebar3`, put it in your `PATH` and give it executable permissions. Now you can install the Hex plugin by adding `{plugins, [rebar3_hex]}` to `~/.config/rebar3/rebar.config` and run all of its tasks.

h4cc / awesome-elixir

• ライブラリのリンク集

Actors

Libraries and tools for working with actors and such.

- [exactor](#) - Helpers for easier implementation of actors in Elixir.
- [exos](#) - A Port Wrapper which forwards cast and call to a linked Port.
- [mon_handler](#) - A minimal GenServer that monitors a given GenEvent handler.
- [pool_ring](#) - Create a pool based on a hash ring.
- [poolboy](#) - A hunky Erlang worker pool factory.
- [pooler](#) - An OTP Process Pool Application.
- [sbroker](#) - Sojourn-time based active queue management library.
- [workex](#) - Backpressure and flow control in EVM processes.

Algorithms and Data structures

Libraries and implementations of algorithms and data structures.

Dialyzer

ソースに型情報をアノテーションしとくと
静的型チェックはできる

get_by!/3

Specs:

- `get_by!(Ecto.Queryable.t, Keyword.t, Keyword.t) :: Ecto.Model.t | nil | no_return`

Similar to `get_by/3` but raises `Ecto.NoResultsError` if no record was found.



マクロ



Elixir 入門ソース (日本語)

- Web+DB PRESS vol.88
 - <http://gihyo.jp/magazine/wdpress/archive/2015/vol88>
- Joe_noh さんのスライド集
 - http://www.slideshare.net/Joe_noh/presentations
- Getting Started の翻訳
 - http://elixir-ja.sena-net.works/getting_started/1.html
- Qiita – Elixir
 - <https://qiita.com/tags/elixir>



Elixir 本: おすすめ

The
Pragmatic
Programmers

Programming Elixir

Functional
> Concurrent
> Pragmatic
> Fun

Dave Thomas

Foreword by
José Valim,
Creator of Elixir

edited by Lynn Beighley



KAIZEN PLATFORM

まとめ

- Elixir は Erlang/OTP とズッ友
- 動的型付けの関数型言語
 - と、言っても怖くない
 - パターンマッチ、パイプライン演算子、Enum/Stream
- OTP でアクターな並行処理
- Let it crash

