蛇とビーム

ヽ（´・肉・｀）ノ

- にくです
- 農業とITをつなぐベンチャーで働いています
- コンサドーレ札幌が好きです

発表の目的

必要なときに ErlangVM を思い出して
選択肢として含められるようにすること

サッポロビームの方から

http://sapporo-beam.github.io/

ErlangVMに載っている言語に関する話題やそうでない話題でわいわいやる集まりです.
だいたい毎週木曜日にやっています.

このまえ 80 回目でした

ビーム(Beam)

```
http://www.ymotongpoo.com/works/lyse-ja/ja/04_modules.html#id3
```

.beam はBogdan/Björnさんの Erlang抽象マシン（Bogdan/ Björn's Erlang Abstract Machine）を表しています。この名前はVM自身の名前です。

ErlangVM

そもそもは Erlang という言語のために作られ，育てられた VM

Erlang

そもそもは ERICSSON という会社で作られた言語

ERICSSON

スウェーデンにある通信機器メーカー

通信機器ソフトに求められる

- 処理を並行してたくさん行う
- 障害をのりきって動く
- バージョンアップ中にも動く
- それなりに速く動作する

ErlangVM もまた

- 処理を並行してたくさん行う
- 障害をのりきって動く
- バージョンアップ中にも動く
- それなりに速く動作する

日本だと

http://www.slideshare.net/takahiro_yachi/ss-44828680

ドワンゴさんのニコニコ生放送は
Erlang で運用されているみたい

（｀・肉・）くよさそう

フフ，そうでしょうとも．

(´・肉・）く知らないなあ

アッ……ハイ

すごくよく話題に登るわけではない

出自が異なる

http://archive.oreilly.com/pub/a/oreilly//news/languageposter_0504.html

- Python は C 由来(たぶん？)
- Erlang は Prolog 由来

C で見慣れた構文とは少し違う

Erlang コードの例

運転免許の例

```
-module(license).
-export([right_age/1]).

right_age(X) when X >= 16, X =< 104 ->
    true;
right_age(_) ->
    false.
```

- 慣れると特に困らない(みたい

Elixir コードの例

```
defmodule License do
  def right_age(x) when x >= 16 and x <= 104, do: true
  def right_age(_), do: false
end
```

- ErlangVM で使える言語 Elixir
- 速度ペナルティはない
- 多少みたことある感じ

今回は Elixir で説明します

- 僕が慣れているため
- 当然 Erlang でも同じことができます

処理を並行してたくさん行う

重くてどのくらいかかるかわからない処理を 6 個行う.
結果がわかった順に 2 つ 1 組のペアにして表示する.
ただし 5 秒待って何も結果が出なければタイムアウトする.

好きな言語で実装してみよう

処理を並行してたくさん行う

```elixir
defmodule Producer do
  def produce(dest_pid) do
    Stream.repeatedly(&:random.uniform/0) # ランダムな値
    |> Stream.with_index                  # n 番目
    |> Stream.each(fn {random, index} ->  # {ランダムな値, n 番目}
      spawn fn ->                         # 別プロセスを作る
        sleeptime = trunc(random * 2 * 1000)
        IO.puts "Producer #{index} 計算時間: #{sleeptime}ms"
        :timer.sleep(sleeptime)  # 0-2 秒待って(重い処理のつもり)
        IO.puts "Producer #{index} 送りまーす"
        send(dest_pid, index)    # dest_pid に結果を送る
      end
    end)
  end
end
```

処理を並行してたくさん行う

```
defmodule Consumer do
  def consume(list) do
    IO.puts "Consumer まちまーす(list の長さ #{length list})"
    receive do
      x ->       # x を受けとったら
        IO.puts "Consumer #{x} 受けましたー"
        if length(list) == 1 do
          IO.puts "Consumer ペアは #{hd(list)} と #{x} です"
          consume([])
        else
          consume([x])
        end
    after
      5000 ->    # 5 秒待ったら打ち切る
        IO.puts "Consumer 時間切れ"
    end
  end
end
```

処理を並行してたくさん行う

https://gist.github.com/niku/5796caa4aec4f9edd438

実行する

```
:random.seed(:os.timestamp)                # ランダムな値を生成するための処理
                                           # (おまじない)
Enum.take(Producer.produce(self()), 6) # 自分のプロセス self() に 6 個作って送る
Consumer.consume([])                       # メッセージが送られてくるのを待つ
```

処理を並行してたくさん行う

```
Consumer まちまーす(list の長さ 0)
Producer 0 計算時間: 1129ms
Producer 1 計算時間: 776ms
Producer 2 計算時間: 174ms
Producer 3 計算時間: 909ms
Producer 4 計算時間: 1474ms
Producer 5 計算時間: 1328ms
Producer 2 送りまーす
Consumer 2 受けましたー
Consumer まちまーす(list の長さ 1)
Producer 1 送りまーす
Consumer 1 受けましたー
Consumer ペアは 2 と 1 です
Consumer まちまーす(list の長さ 0)
Producer 3 送りまーす
Consumer 3 受けましたー
Consumer まちまーす(list の長さ 1)
Producer 0 送りまーす
Consumer 0 受けましたー
Consumer ペアは 3 と 0 です
Consumer まちまーす(list の長さ 0)
Producer 5 送りまーす
Consumer 5 受けましたー
Consumer まちまーす(list の長さ 1)
Producer 4 送りまーす
Consumer 4 受けましたー
Consumer ペアは 5 と 4 です
Consumer まちまーす(list の長さ 0)
Consumer 時間切れ
```

障害をのりきって動く

「最初は `0` で初期化，値がきたら今までの値と足し合わせる．
エラーになったら合計を表示する．
その後また初期化から処理を再開する」
というモジュールを作る．

好きな言語で実装してみよう

障害をのりきって動く

http://www.ymotongpoo.com/works/lyse-ja/ja/01_introduction.html#erlang

Erlangの一般的なポリシーをお教えします：「クラッシュするならさせておけ」です。(略)
下にネットが敷いてある綱渡りのようなものです。

障害をのりきって動く

Let it crash
(クラッシュするならさせておけ)

（｀・肉・）＜ **信じて作る**

障害をのりきって動く

```elixir
defmodule Calculator do
  use GenServer
  # Client
  def start_link(number), do: GenServer.start_link(__MODULE__, number, name: __MODULE__)
  def plus(x), do: GenServer.cast(__MODULE__, {:plus, x})

  # Server
  def init(number) do
    IO.puts "#{number} から始めまーす"
    {:ok, number}
  end

  def handle_cast({:plus, x}, state) do
    IO.puts "#{state} に #{x} を足しまーす"
    {:noreply, state + x}
  end

  def terminate(_reason, state) do
    IO.puts "#{state} で終わりまーす"
    :ok
  end
end
```

障害をのりきって動く

```
$ iex -r let-it-crash.exs
iex(1)> Calculator.start_link(0)
0 から始めまーす
{:ok, #PID<0.68.0>}
iex(2)> Calculator.plus(1)
0 に 1 を足しまーす
:ok
iex(3)> Calculator.plus(3)
1 に 3 を足しまーす
:ok
iex(4)> Calculator.plus("bom!")
4 に bom! を足しまーす
:ok
4 で終わりまーす
(略)
iex(1)> Calculator.plus(2)
** (ArgumentError) argument error
             :erlang.send(Calculator, {:"$gen_cast", {:plus, 2}})
    (elixir) lib/gen_server.ex:614: GenServer.do_send/2
```

障害をのりきって動く

（´・肉・）＜ 障害のりきれなかた

障害をのりきって動く

「最初は `0` で初期化，値がきたら今までの値と足し合わせる．
エラーになったら合計を表示する．
`-------` ここまでできた `--------`
その後また初期化から処理を再開する」
というモジュールを作る．

障害をのりきって動く

http://www.ymotongpoo.com/works/lyse-ja/ja/01_introduction.html#erlang

Erlangの一般的なポリシーをお教えします：「クラッシュするならさせておけ」です。(略)
下にネットが敷いてある綱渡りのようなものです。

（´・肉・）＜**ネット敷いてなかた**

障害をのりきって動く

ErlangVM においてのネット
それは **Supervisor** (監視者)

Worker が綱から落ちたら再挑戦
させてくれる

障害をのりきって動く

```
defmodule Calculator.Supervisor do
  use Supervisor

  def start_link, do: Supervisor.start_link(__MODULE__, [])

  def init(_arg) do
    # 監視する対象を指定して
    children = [
      worker(Calculator, [0])
    ]

    # 監視する
    supervise(children, strategy: :one_for_one)
  end
end
```

障害をのりきって動く

```
$ iex -r let-it-crash.exs
iex(1)> Calculator.Supervisor.start_link
0 から始めまーす
{:ok, #PID<0.71.0>}
iex(2)> Calculator.plus(1)
0 に 1 を足しまーす
:ok
iex(3)> Calculator.plus(3)
1 に 3 を足しまーす
:ok
iex(4)> Calculator.plus("bom!")
4 に bom! を足しまーす
4 で終わりまーす
:ok
0 から始めまーす
iex(5)> Calculator.plus(2)
0 に 2 を足しまーす
:ok
```

障害をのりきって動く

「最初は 0 で初期化，値がきたら今までの値と足し合わせる．
エラーになったら合計を表示する．
その後また初期化から処理を再開する」
というモジュールを作る．

（｀・肉・）＜ **障害のりきれた！**

バージョンアップ中にも動く

「ver0では数字が1ずつ増える．ver1では数字が10ずつ増える．
そのときデータの流れは止めない」

好きな言語で実装してみよう

バージョンアップ中にも動く

```elixir
# version_up-0.exs
defmodule Successor do
  use GenServer

  @vsn "0"

  def start_link, do: GenServer.start_link(__MODULE__, 0, name: __MODULE__)
  def next_number, do: GenServer.call(__MODULE__, :next_number)

  def handle_call(:next_number, _from, state) do
    reply = state + 1
    {:reply, reply, reply}
  end
end
```

バージョンアップ中にも動く

```
# version_up-1.exs
defmodule Successor do
  use GenServer

  @vsn "1"

  def start_link, do: GenServer.start_link(__MODULE__, 0, name: __MODULE__)
  def next_number, do: GenServer.call(__MODULE__, :next_number)

  def handle_call(:next_number, _from, state) do
    reply = state + 10
    {:reply, reply, reply}
  end

  def code_change("0", state, extra) do
    IO.puts "バージョン 0 から #{@vsn} にコード書き変えます．今の値は #{state}"
    {:ok, state * 10}
  end
end
```

バージョンアップ中にも動く

```
$ iex -r version_up-0.exs
iex(1)> Successor.start_link
{:ok, #PID<0.68.0>}
iex(2)> Successor.next_number
1
iex(3)> Successor.next_number
2
iex(4)> :sys.suspend Successor
:ok
iex(5)> c("version_up-1.exs")
version_up-1.exs:1: warning: redefining module Successor
version_up-1.exs:14: warning: variable extra is unused
[Successor]
iex(6)> :sys.change_code Successor, Successor, "0", []
バージョン 0 から 1 にコード書き変えます. 今の値は 2
:ok
iex(7)> :sys.resume Successor
:ok
iex(8)> Successor.next_number
30
```

それなりに速く動作する

- Erlang だとプロセスの分離 spawn を多用する
- Python だとだいたい Thread といえるかな……

多用する spawn はどのくらい速いのか

それなりに速く動作する

```
import threading

list(map (lambda x:threading.Thread().start(), range(1000000)))
```

```
$ time python3 thread.py
python3 thread.py  59.94s user 45.48s system 124% cpu 1:24.54 total
```

それなりに速く動作する

```
Enum.map((1..1000000), fn (_x) ->
  spawn(fn ->
  end)
end)
```

```
$ time elixir --erl "+P 1000000" spawns.exs
elixir --erl "+P 1000000" fork.exs  5.18s user 8.35s system 148% cpu 9.125 total
```

それなりに速く動作する

- Python 100 万個のスレッドを作るのに 84.5 秒.
- ErlangVM 100 万個のプロセス (スレッドのようなもの) を作るのに 9.13 秒

10 倍くらい速いみたい

それなりに速く動作する

注意: **ErlangVMは普通のCPU処理は遅いみたいです**