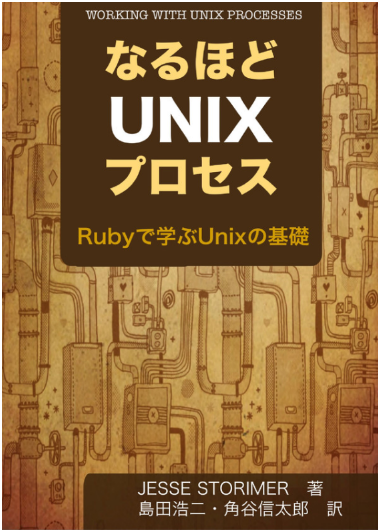
なるほど Erlang プロセス

About me

- ヽ（´・肉・｀）ノ / @niku_name
- コンサドーレ札幌が好きです
- サッポロビームによくいます

なるほど UNIX プロセス



なるほど UNIX プロセス

Erlang

- 元々は電話交換機の制御用
- 耐障害性が高い
- 並行処理が得意

なるほど Erlang プロセス

```
Erlang プロセスには ID がある
Erlang プロセスは子プロセスを作れる
Erlang プロセスはプロセス毎に値を保持できる
Erlang プロセスはメッセージを送受信できる
Erlang プロセスは別々に動ける
Erlang プロセスは名前を束縛できる
Erlang プロセスはプロセス数を制限できる
Erlang プロセスは軽量
Erlang プロセスは exit シグナルを送る/受けとる
Erlang プロセスは繋げる
Erlang プロセスは見ておくことができる
Erlang プロセスを扱うライブラリ Erlang/OTP
```

Erlang プロセスには ID がある

```
/Users/niku% iex
iex(1)> self()
#PID<0.80.0>
iex(2)>
```

Erlang プロセスは子プロセスを作れる

```
iex(1)> self()
#PID<0.80.0>
iex(2)> spawn(fn ->
...(2)>   IO.inspect self()
...(2)> end)
#PID<0.85.0>
#PID<0.85.0>
iex(3)>
```

Erlang プロセスはプロセス毎に値を保持できる

```
iex(1)> Process.put(:a, "foo")
nil
iex(2)> Process.get(:a)
"foo"
iex(3)> spawn(fn ->
...(3)>   IO.inspect Process.get(:a)
...(3)> end)
nil
#PID<0.86.0>
iex(4)> Process.get(:a)
"foo"
iex(5)>
```

Erlang プロセスはメッセージを送受信できる

プロセスは，メッセージを send と receive してプロセス間のやりとりを行う.

- 送る方 send は非同期
- 受ける方 receive は同期（メッセージがくるまでブロックする）
  タイムアウトできる

Erlang プロセスはメッセージを送受信できる

```
iex(1)> child_pid = spawn(fn ->
...(1)>   IO.puts "spawned"
...(1)>   receive do
...(1)>     x -> IO.puts "#{inspect self()} #{x}"
...(1)>   end
...(1)>   IO.puts "received"
...(1)> end)
spawned
#PID<0.88.0>
iex(2)> send(child_pid, "hello~")
#PID<0.88.0> hello~
received
"hello~"
iex(3)>
```

Erlang プロセスはメッセージを送受信できる

```
child_pid = spawn(fn ->
...(1)>   IO.puts "spawned"
...(1)>   receive do
...(1)>     x -> IO.puts "#{inspect self()} #{x}"
...(1)>   after 10_000 ->
...(1)>     IO.puts "Timeout"
...(1)>   end
...(1)>   IO.puts "received"
...(1)> end)
spawned
#PID<0.91.0>
iex(2)>
nil
iex(3)> # 10秒待つ
nil
iex(4)> Timeout
iex(4)> received
iex(4)>
```

Erlang プロセスはメッセージを送受信できる

送る方は send で勝手におくりつけてくる
受ける方は receive を明示的に呼ばないと，メッセージを受ける体制に入らない

x = x + 1 をマルチスレッドで動かすとおかしくなるような問題が起きない

Erlang プロセスは別々に動ける

```
# fib.exs
defmodule Fib do
  def calc(n) when n == 0, do: 0
  def calc(n) when n == 1, do: 1
  def calc(n) when 2 <= n do
    calc(n - 1) + calc(n - 2)
  end

  def calc_three_processes_in_parallel(n) do
    me = self()
    # 3並列に計算し,完了したら答えを送ってもらう
    spawn(fn -> send(me, calc(n)) end)
    spawn(fn -> send(me, calc(n)) end)
    spawn(fn -> send(me, calc(n)) end)
    do_receive([])
  end

  defp do_receive(result) when length(result) == 3 do
    result
  end
  defp do_receive(result) when length(result) < 3 do
    receive do
      x ->
        do_receive([x | result])
    end
  end
end
```

Erlang プロセスは別々に動ける

```
iex(1)> import_file("fib.exs")
{:module, Fib,
 <<70, 79, 82, 49, 0, 0, 9, 152, 66, 69, 65, 77, 69, 120, 68, 99, 0, 0, 0, 219,
   131, 104, 2, 100, 0, 14, 101, 108, 105, 120, 105, 114, 95, 100, 111, 99, 115,
   95, 118, 49, 108, 0, 0, 0, 4, 104, 2, ...>>, {:do_receive, 1}}
iex(2)> {single_execution_time, answer} =
          :timer.tc(Fib, :calc, [40])
{7219356, 102334155}
iex(3)> {multi_execution_time, answer} =
          :timer.tc(Fib, :calc_three_processes_in_parallel, [40])
{10781720, [102334155, 102334155, 102334155]}
iex(4)> multi_execution_time / single_execution_time
1.4934462298299183
iex(5)>
```

Erlang プロセスは名前を束縛できる

```
iex(1)> child_pid = spawn(fn ->
...(1)>   receive do
...(1)>     x -> IO.puts "#{inspect self()}: #{x}"
...(1)>   end
...(1)> end)
#PID<0.109.0>
iex(2)> Process.register(child_pid, :echo)
true
iex(3)> send(:echo, "hi")
#PID<0.109.0>: hi
"hi"
iex(4)>
```

Erlang プロセスはプロセス数を制限できる

```
iex(1)> :erlang.system_info(:process_limit)
262144
iex(2)>
```

Erlang プロセスはプロセス数を制限できる

```
/Users/niku% ELIXIR_ERL_OPTIONS="+P 1024" iex
iex(1)> :erlang.system_info(:process_limit)
1024
iex(2)> Enum.map(0..1024, fn n ->
...(2)>   IO.puts Enum.count(Process.list())
...(2)>   spawn(fn ->
...(2)>     receive do
...(2)>       x -> IO.inspect x
...(2)>     end
...(2)>   end)
...(2)> end)
43
44
(...snip...)
1022
1023
1024
** (SystemLimitError) a system limit has been reached

23:12:15.338 [error] Too many processes
```

Erlang プロセスは軽量

```
iex(1)> f = fn ->
...(1)>   receive do
...(1)>   after :infinity -> :ok
...(1)>   end
...(1)> end
#Function<20.52032458/0 in :erl_eval.expr/5>
iex(2)> {_, total_bytes} = Process.info(spawn(f), :memory)
{:memory, 2720}
iex(3)> wordsize = :erlang.system_info(:wordsize)
8
iex(4)> total_bytes / wordsize
340.0
iex(5)> {_, heap_size} = Process.info(spawn(f), :heap_size)
{:heap_size, 233}
iex(6)>
```

Erlang プロセスは軽量

- 1プロセス作るのにヒープ領域込みで最小 309 words(**2472バイト**)
- この環境では Hipe という拡張が有効になっているため 340 words になっている

Erlang プロセスは軽量

100万プロセス動かす

Erlang プロセスは軽量

```elixir
# chain.exs
defmodule Chain do
  def create_processes(n) do
    # n個目の(最後の)プロセスは親プロセスにメッセージを送る
    last_pid = Enum.reduce(1..n, self(), fn (_, send_to) ->
      spawn(fn ->
        # メッセージを受けたら1足して次のメッセージに送る
        receive do
          x -> send(send_to, x + 1)
        end
      end)
    end)
    IO.puts "count: #{Enum.count(Process.list())}, memory: :#{:erlang.memory(:total)}"
    send(last_pid, 0) # n個の子プロセスの最初の1個を動かす

    receive do
      final_answer -> "Result is #{final_answer}"
    end
  end

  def run(n) do
    {execution_micro_secs, final_answer} =
      :timer.tc(Chain, :create_processes, [n])
    seconds = execution_micro_secs / (1000 * 1000)
    IO.puts "#{seconds} seconds, #{final_answer}"
  end
end
```

Erlang プロセスは軽量

```
/Users/niku% system_profiler SPHardwareDataType
Hardware:

    Hardware Overview:

      Model Name: MacBook Pro
      Model Identifier: MacBookPro12,1
      Processor Name: Intel Core i5
      Processor Speed: 2.7 GHz
      Number of Processors: 1
      Total Number of Cores: 2
      L2 Cache (per Core): 256 KB
      L3 Cache: 3 MB
      Memory: 16 GB
(...snip...)

/Users/niku% ELIXIR_ERL_OPTIONS="+P 10000000" elixir -r chain.exs -e "Chain.run(1_000_000)"
count: 1000037, memory: :2970141432
8.74099 seconds, Result is 1000000
```

Erlang プロセスは軽量

軽量2つの利点

- かかった時間8.75秒 -> プロセス作成の速さ
- 使用メモリ3GB弱 -> 省メモリ

カジュアルに殺したり作ったりしやすい

Erlang プロセスは exit シグナルを送る/受けとる

2種類の exit シグナル

1. `exit(:normal)` :: 受けとった方は何もしない
2. `exit(任意の値)` :: 受けとった方は死ぬ

Erlang プロセスは exit シグナルを送る/受けとる

```
iex(1)> other_pid = spawn(fn ->
...(1)>   receive do
...(1)>     x -> IO.puts x
...(1)>   end
...(1)> end)
#PID<0.88.0>
iex(2)> Process.alive?(other_pid)
true
iex(3)> Process.exit(other_pid, :normal)
true
iex(4)> Process.alive?(other_pid)
true
iex(5)> Process.exit(other_pid, :omg)
true
iex(6)> Process.alive?(other_pid)
false
iex(7)>
```

Erlang プロセスは exit シグナルを送る/受けとる

- exit シグナルも特別なメッセージにすぎない

Erlang プロセスは exit シグナルを送る/受けとる

```
iex(1)> other_pid = spawn(fn ->
...(1)>   Process.flag(:trap_exit, true)
...(1)>   receive do
...(1)>     x -> IO.inspect x
...(1)>   end
...(1)> end)
#PID<0.88.0>
iex(2)>
nil
iex(3)> Process.exit(other_pid, :omg)
{:EXIT, #PID<0.81.0>, :omg}
true
iex(4)>
```

Erlang プロセスはシグナルを送る/受けとる

- exit シグナルを受けとって死ぬことに意味はあるのか？
- 次に挙げる特徴と組合せると便利

Erlang プロセスは繋げる

- プロセスを繋げる(linkする)ことができる
- **繋がっていないプロセス** で何があっても **影響を受けない**
- **繋がっているプロセス** は **exitシグナルを受けとる**

Erlang プロセスは繋げる

```
iex(1)> spawn(fn ->
...(1)>   Process.flag(:trap_exit, true)
...(1)>
...(1)>   spawn(fn ->
...(1)>     exit(:omg)
...(1)>   end)
...(1)>
...(1)>   receive do
...(1)>     message ->
...(1)>       IO.puts "An exit signal received: #{inspect message}"
...(1)>   after 1000 ->
...(1)>       IO.puts "timeout"
...(1)>   end
...(1)> end)
#PID<0.96.0>
iex(2)>
nil
iex(3)> timeout
iex(3)>
```

Erlang プロセスは繋げる

```
iex(1)> spawn(fn ->
...(1)>   Process.flag(:trap_exit, true)
...(1)>
...(1)>   spawn_link(fn ->
...(1)>     exit(:omg)
...(1)>   end)
...(1)>
...(1)>   receive do
...(1)>     message ->
...(1)>       IO.puts "An exit signal received: #{inspect message}"
...(1)>   after 1000 ->
...(1)>       IO.puts "timeout"
...(1)>   end
...(1)> end)
#PID<0.96.0>
An exit signal received: {:EXIT, #PID<0.97.0>, :omg}
iex(2)>
nil
iex(3)>
```

Erlang プロセスは繋げる

`exit` シグナルはハンドリングしないエラーがプロセスに到達した場合にも発生する

Erlang プロセスは繋げる

```
iex(1)> spawn(fn ->
...(1)>   Process.flag(:trap_exit, true)
...(1)>
...(1)>   spawn_link(fn ->
...(1)>     1 / 0
...(1)>   end)
...(1)>
...(1)>   receive do
...(1)>     message ->
...(1)>       IO.puts "An exit signal received: #{inspect message}"
...(1)>     end
...(1)> end)
#PID<0.110.0>
iex(2)>
23:45:15.070 [error] Process #PID<0.111.0> raised an exception
** (ArithmeticError) bad argument in arithmetic expression
    :erlang./(1, 0)
An exit signal received: {:EXIT, #PID<0.111.0>, {:badarith, [{:erlang, :/, [1, 0], []}]}}
iex(2)>
```

Erlang プロセスは繋げる

- [再掲] exit シグナルを受けとって死ぬことに意味はあるのか？

Erlang プロセスは繋げる

もしエラーのあるプロセスがクラッシュしたけれど、それに依存しているプロセスが動き続けているとしたら、それら依存プロセスすべては依存先がなくなったことに対処しなければならなくなります。

すごいErlangゆかいに学ぼう！第12章 - エラーとプロセス

Erlang プロセスは繋げる

- 意味のあるグループで複数のプロセスを繋げる :: **プロセス管理対象数の抑制**
- 1つのプロセスが死んだらグループ全てのプロセスが死ぬ :: **グループの中途半端な状態の抑制**
- 死んだことがすぐ検知できる :: **すぐ新しいプロセスを作ることができる**

Erlang プロセスは繋げる

- 静的型付き言語はランタイムエラーを起こさないことを目指すことで(100%の)安定動作を目指せる
- ErlangVM はエラーになったときに，検知/再開を素早く行うことを目指すことで(99.9…%の)安定動作を目指せる

ソースなし（私見）

Erlang プロセスは見ておくことができる

- プロセスを見ておく(monitor)できる
- 見るプロセス，見られるプロセスという立場が存在する
- 2 つのプロセス間で複数のモニターが持てる(スタックでき，それぞれに識別できる)

Erlang プロセスは見ておくことができる

- 見る方が死んでも，見られる方には関係ない
- モニターを複数作れる

ライブラリのような，相互に強固に結びついていないプロセスが，他のプロセスを監視するのに向いている

Erlang プロセスは見ておくことができる

```
iex(1)> spawn(fn ->
...(1)>   spawn_monitor(fn ->
...(1)>     exit(:omg)
...(1)>   end)
...(1)>
...(1)>   receive do
...(1)>     message ->
...(1)>       IO.puts "An exit signal received: #{inspect message}"
...(1)>   end
...(1)> end)
#PID<0.92.0>
An exit signal received: {:DOWN, #Reference<0.0.4.260>, :process, #PID<0.93.0>, :omg}
iex(2)>
nil
iex(3)>
```

Erlang プロセスを扱うライブラリ Erlang/OTP

- 本体に同梱されている
- プロセス間のインタラクションをいい感じに

Erlang プロセスを扱うライブラリ Erlang/OTP

```elixir
# stack.exs
defmodule Stack do
  use GenServer # OTPに含まれるライブラリ

  def start_link(state, opts \\ []) do
    GenServer.start_link(__MODULE__, state, opts)
  end

  # 初期化
  def init(state) do
    {:ok, state}
  end

  # 同期呼び出し
  def handle_call(:pop, _from, [h | t]) do
    {:reply, h, t}
  end

  # 非同期呼び出し
  def handle_cast({:push, item}, state) do
    {:noreply, [item | state]}
  end
end
```

Erlang プロセスを扱うライブラリ Erlang/OTP

```
iex(1)> import_file("stack.exs")
{:module, Stack,
 <<70, 79, 82, 49, 0, 0, 12, 104, 66, 69, 65, 77, 69, 120, 68, 99, 0, 0, 2, 144,
   131, 104, 2, 100, 0, 14, 101, 108, 105, 120, 105, 114, 95, 100, 111, 99, 115,
   95, 118, 49, 108, 0, 0, 0, 4, 104, 2, ...>>, {:handle_cast, 2}}
iex(2)> import Supervisor.Spec
Supervisor.Spec
iex(3)> children = [
...(3)>   worker(Stack, [[:hello], [name: MyStack]])
...(3)> ]
[{Stack, {Stack, :start_link, [[:hello], [name: MyStack]]}, :permanent, 5000,
  :worker, [Stack]}]
iex(4)> {:ok, pid} = Supervisor.start_link(children, strategy: :one_for_one)
{:ok, #PID<0.91.0>}
```

Erlang プロセスを扱うライブラリ Erlang/OTP

```
iex(5)> GenServer.call(MyStack, :pop)
:hello
iex(6)> GenServer.cast(MyStack, {:push, :world})
:ok
iex(7)> GenServer.call(MyStack, :pop)
:world
iex(8)> GenServer.call(MyStack, :pop)
(...snip...)
17:08:37.217 [error] GenServer MyStack terminating
** (FunctionClauseError) no function clause matching in Stack.handle_call/3
    iex:15: Stack.handle_call(:pop, {#PID<0.80.0>, #Reference<0.0.4.299>}, [])
    (stdlib) gen_server.erl:615: :gen_server.try_handle_call/4
    (stdlib) gen_server.erl:647: :gen_server.handle_msg/5
    (stdlib) proc_lib.erl:247: :proc_lib.init_p_do_apply/3
Last message: :pop
State: []
    (elixir) lib/gen_server.ex:737: GenServer.call/3
iex(8)> GenServer.call(MyStack, :pop)
:hello
iex(9)>
```

今日話したこと

- ErlangVM のプロセス軽量
- ErlangVM のプロセス間のやりとり，エラー処理
  - メッセージパッシング
  - リンク
  - モニター
- 使いやすくしたライブラリ OTP

まとめ

- プロセス同士のインタラクションで耐障害性を担保
- 土台(プロセス)から上については意識する機会が多い
- 土台(プロセス)とか，土台(プロセス)同士のインタラクションとか意識して知ると便利かもよ

まとめ

だいたい毎週木曜日19:00から ErlangVM やそうでないことについてわいわいやる **オンライン参加もできる** サッポロビーム

今日話したこと

- ErlangVM のプロセス軽量
- ErlangVM のプロセス間のやりとり，エラー処理
  - メッセージパッシング
  - リンク
  - モニター
- 使いやすくしたライブラリ OTP