

The Power of ~~LOVE~~... Actors

Bernhard Stöcker

Recognizer Group GmbH

bernhard.stoecker@recognizer.de

November 6, 2016

Overview

- 1 What are Actors?
- 2 Actors in the real world
- 3 Actors in Elixir
- 4 Actors in Scala
- 5 Elixir vs Scala

What are Actors?

What are Actors?

What are Actors?

Act-or: to act: **"to do something for a particular purpose or to solve a problem"** (From the Cambridge dictionary)

What are Actors?

Wikipedia:

- A mathematical model of concurrent computation
- Actors can hold and modify private state
- Affect each other through messages only
- In response to a message that it receives, an actor can:
 - Make local decisions
 - Create more actors
 - Send more messages
 - Respond to the incoming message

Actors in the real world

WhatsApp

What are Actors?
Actors in the real world
Actors in Elixir
Actors in Scala
Elixir vs Scala

Actors in the real world

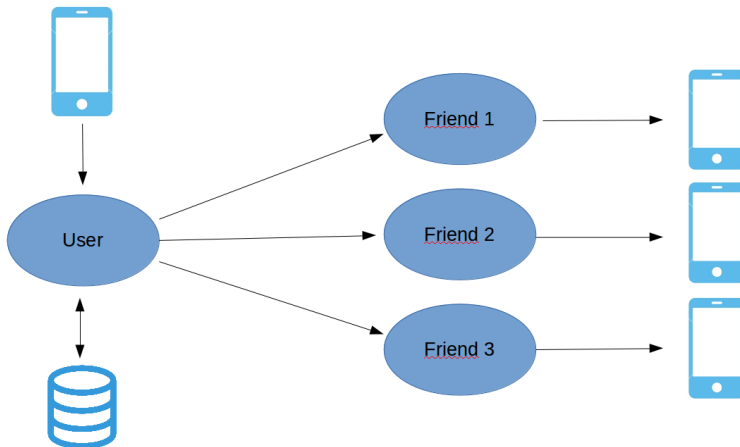


Actors in the real world

WhatsApp

- Every user has an actor representing her
- When sending a message my actor sends a message to all related users
- The users receiving a message ensure the message is delivered.

Actors in the real world



Actors in Elixir

Actors in Elixir

Actors in Elixir

```
bernhard@bernhard's-thinkpad ~ $ iex
Erlang/OTP 19 [erts-8.1] [source-4cc2ce3] [64-bit]

Interactive Elixir (1.3.3) - press Ctrl+C to exit (
iex(1)> spawn fn -> IO.puts("Hello World") end
Hello World
#PID<0.83.0>
iex(2)> █
```

Actors in Elixir

```
bernhard@bernhard's-thinkpad ~ $ iex
Erlang/OTP 19 [erts-8.1] [source-4cc2ce3] [64-bit] [smp:4:4]

Interactive Elixir (1.3.3) - press Ctrl+C to exit (type h() E
iex(1)> func = fn ->
...(1)> receive do
...(1)> name -> IO.puts("Hello #{name}")
...(1)> end
...(1)> end
#Function<20.52032458/0 in :erl_eval.expr/5>
iex(2)> pid = spawn func
#PID<0.88.0>
iex(3)> send(pid, "Kira")
Hello Kira
"Kira"
iex(4)> █
```

Actors in Elixir

```
iex(1)> defmodule Value do
... (1)> def current(x) do
... (1)> receive do
... (1)> add -> IO.puts(x + add); current(x + add)
... (1)> end
... (1)> end
... (1)> end
{:module, Value,
 <<70, 79, 82, 49, 0, 0, 5, 40, 66, 69, 65, 77, 69, 120, 68, 99, 0, 0, 0, 148,
 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, ...>>, {:current, 1}}
iex(2)> pid = spawn fn -> Value.current(0) end
#PID<0.95.0>
iex(3)> send pid, 17
17
iex(4)> send pid, 16
33
iex(5)> send pid, 9
42
iex(6)> 
```

Actors in Elixir

```
defmodule Stack do
  use GenServer

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

  def handle_call(:pop, _from, []) do
    {:reply, nil, []}
  end
  def handle_call(:pop, _from, [h | t]) do
    {:reply, h, t}
  end
  end

  def handle_call(:top, _from, []) do
    {:reply, nil, []}
  end
  def handle_call(:top, _from, [h | t]) do
    | {:reply, h, [h | t]}
  end
  end

  def handle_cast({:push, item}, state) do
    {:noreply, [item | state]}
  end
end
```

Actors in Elixir

```
^Cbernhard@bernhards-thinkpad ~/Dokumente/Officetalk/elixir (master ***) $ iex
Erlang/OTP 19 [erts-8.1] [source-4cc2ce3] [64-bit] [smp:4:4] [async-threads:10]

Interactive Elixir (1.3.3) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> c("stack.ex")
[Stack]
iex(2)> Stack.start_link(["Hello", "World"], [name: MyStack])
{:ok, #PID<0.89.0>}
iex(3)> GenServer.call(MyStack, :pop)
"Hello"
iex(4)> GenServer.call(MyStack, :pop)
"World"
iex(5)> GenServer.cast(MyStack, {:push, "Hello Kira"})
:ok
iex(6)> GenServer.call(MyStack, :top)
"Hello Kira"
iex(7)> █
```

Actors in Elixir

```
defmodule SupervisedStack do
  import Supervisor.Spec

  def start_link do
    children = [
      worker(Stack, [[:hello], [name: MyStack]])
    ]

    Supervisor.start_link(children, strategy: :one_for_one)
  end
end
```


Actors in Elixir

```

bernhard@bernhards-thinkpad ~/Dokumente/Officetalk/elixir (master *) $ iex
Erlang/OTP 19 [erts-8.1] [source-4cc2ce3] [64-bit] [smp:4:4] [async-threads:10] [hipe] [kernel-poll:false]

Interactive Elixir (1.3.3) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> c(["stack.ex", "supervised_stack.ex"])
[Stack, SupervisedStack]
iex(2)> SupervisedStack.start_link(["Hello World"], [name: TheStack])
{:ok, #PID<0.94.0>}
iex(3)> GenServer.call(TheStack, :pop)
"Hello World"
iex(4)> GenServer.cast(TheStack, {:push, "Hello Kira"})
:ok
iex(5)> GenServer.call(TheStack, :top)
"Hello Kira"
iex(6)> GenServer.call(TheStack, :foobar)
** (exit) exited in: GenServer.call(TheStack, :foobar, 5000)
    ** (EXIT) an exception was raised:
        ** (FunctionClauseError) no function clause matching in Stack.handle_call/3
            stack.ex:8: Stack.handle_call(:foobar, {#PID<0.81.0>, #Reference<0.0.2.467>}, ["Hello Kira"])
            (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

22:00:21.799 [error] GenServer TheStack terminating
** (FunctionClauseError) no function clause matching in Stack.handle_call/3
    stack.ex:8: Stack.handle_call(:foobar, {#PID<0.81.0>, #Reference<0.0.2.467>}, ["Hello Kira"])
    (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: :foobar
State: ["Hello Kira"]
    (elixir) lib/gen_server.ex:604: GenServer.call/3
iex(6)> GenServer.call(TheStack, :top)
"Hello World"

```

Actors in Scala

Actors in Scala

Actors in Scala

```
package example

import akka.actor.{ Props, Actor, Terminated }

final case class Hello(var name: String)

object Example {
  def props() :Props = Props(classOf[Example])
}

class Example extends Actor {
  def receive = {
    case Hello(name) => { println("Hello " + name) }
    case _ => println("Example received unknown message")
  }
}
```

Actors in Scala

```
package runtime

import akka.actor._
import example._

object Main extends App {

  val system = ActorSystem("KidsActorSystem")
  val exampleActor = system.actorOf(Example.props())

  exampleActor ! Hello("World")

  system.terminate
}
```

Actors in Scala

```
bernhard@bernhards-thinkpad ~/Dokumente/Officetalk/scala_  
[info] Set current project to Actors (in build file:/home/  
[info] Compiling 1 Scala source to /home/bernhard/Dokumen  
[info] Running runtime.Main  
Hello World  
[success] Total time: 4 s, completed 03.11.2016 16:53:15  
bernhard@bernhards-thinkpad ~/Dokumente/Officetalk/scala_
```

Actors in Scala

```
package family

import akka.actor._

trait BaseParent extends Actor {
  def spawnChild(context: ActorContext) :ActorRef

  var child = respawnChild

  def receive = {
    case MeasureKidSize => child ! TellMeSize
    case FeedKid => child ! Feed
    case KillKid => child ! PoisonPill
    case KidSize(size) => println("The child is " + size + "cm tall!")
    case Terminated(childActor) => {
      println("Child actor died. Respawn!")
      child = respawnChild
    }
    case _ => println("Example received unknown message")
  }

  def respawnChild = {
    val childActor = spawnChild(context)
    context.watch(childActor)
    childActor
  }
}
```

Actors in Scala

```
package family

import akka.actor._

object Parent {
  def props(): Props = Props(classOf[Parent])
}

class Parent extends BaseParent {
  def spawnChild(context: ActorContext) = {
    context.system.actorOf(Child.props())
  }
}
```

Actors in Scala

```
package family

import akka.actor.{ Props, Actor }

object Child {
  def props() :Props = Props(classOf[Child])
}

class Child extends Actor {
  var currentSize = 55
  def receive = {
    case Feed => {
      currentSize += 1
      sender() ! KidSize(currentSize)
    }
    case TellMeSize => sender() ! KidSize(currentSize)
    case _ => println("Example received unknown message")
  }
}
```


Actors in Scala

```
package family

case object FeedKid
case object Feed
case object KillKid
final case class KidSize(val size: Int)
case object TellMeSize
case object MeasureKidSize
```

Actors in Scala

```
package runtime

import akka.actor._
import family._

object Main extends App {
  val system = ActorSystem("KidsActorSystem")
  val parentActor = system.actorOf(Parent.props())

  parentActor ! MeasureKidSize
  parentActor ! FeedKid
  parentActor ! FeedKid
  parentActor ! KillKid
  Thread.sleep(100)
  parentActor ! MeasureKidSize
  Thread.sleep(1000)
  system.terminate
}
```

Elixir vs Scala

Elixir	Scala
<ul style="list-style-type: none">• BEAM• Everything is immutable• Actors are recursive functions• Actors are independent from each other• Errors are handled by supervisor only	<ul style="list-style-type: none">• JVM• Choice between mutable and immutable• Actors are objects• Actors are organized in a tree structure• Errors are handled or escalated to parent actor

Elixir vs Scala

Summary

- Scala is faster regarding pure computation speed
- Scala can use eco system of all JVM languages
- Scala is conceptual easier for beginners due to object orientation

Elixir vs Scala

Summary

- BEAM is optimized for actor handling
- Elixir is more pragmatic (get things done much faster)
- Elixir is much easier to test
- Elixir can use Erlang eco system

The End