Extracted from:

Seven Concurrency Models in Seven Weeks

When Threads Unravel

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Seven Concurrency Models in Seven Weeks

When Threads Unravel

Paul Butcher

Series editor: *Bruce Tate*

Development editor: *Jacquelyn Carter*

Seven Concurrency Models in Seven Weeks

When Threads Unravel

Paul Butcher

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Bruce A. Tate (series editor)
Jacquelyn Carter (editor)
Potomac Indexing, LLC (indexer)
Molly McBeath (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-937785-65-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2014

CHAPTER 5

Actors

An actor is like a rental car—quick and easy to get a hold of when you want one, and if it breaks down you don't bother trying to fix it; you just call the rental company and another one is delivered to you.

The actor model is a general-purpose concurrent programming model with particularly wide applicability. It targets both shared- and distributed-memory architectures, facilitates geographical distribution, and provides especially strong support for fault tolerance and resilience.

More Object-Oriented than Objects

Functional programming avoids the problems associated with shared mutable state by avoiding mutable state. Actor programming, by contrast, retains mutable state but avoids sharing it.

An actor is like an object in an object-oriented (OO) program—it encapsulates state and communicates with other actors by exchanging messages. The difference is that actors run concurrently with each other and, unlike OO-style message passing (which is really just calling a method), actors *really* communicate by sending messages to each other.

Although the actor model is a general approach to concurrency that can be used with almost any language, it's most commonly associated with Erlang.[1] We're going to cover actors in Elixir,[2] a relatively new language that runs on the Erlang virtual machine (BEAM).

Like Clojure (and Erlang), Elixir is an impure, dynamically typed functional language. If you're familiar with Java or Ruby, you should find it easy enough

1. http://www.erlang.org/
2. http://elixir-lang.org/

to read. This isn't going to be an Elixir tutorial (this is a book about concurrency, after all, not programming languages), but I'll introduce the important language features we're using as we go along. There may be things you just have to take on faith if you're not already familiar with the language—I recommend *Programming Elixir [Tho14]* if you want to go deeper.

In day 1 we'll see the basics of the actor model—creating actors and sending and receiving messages. In day 2 we'll see how failure detection, coupled with the "let it crash" philosophy, allows actor programs to be fault-tolerant. Finally, in day 3 we'll see how actors' support for distributed programming allows us to both scale beyond a single machine and recover from failure of one or more of those machines.

Day 1: Messages and Mailboxes

Today we'll see how to create and stop processes, send and receive messages, and detect when a process has terminated.

Joe asks:

Actor or Process?

In Erlang, and therefore Elixir, an actor is called a *process*. In most environments a process is a heavyweight entity that consumes lots of resources and is expensive to create. An Elixir process, by contrast, is very lightweight—lighter weight even than most systems' threads, both in terms of resource consumption and startup cost. Elixir programs typically create thousands of processes without problems and don't normally need to resort to the equivalent of thread pools (see *Thread-Creation Redux,* on page ?).

Our First Actor

Let's dive straight in with an example of creating a simple actor and sending it some messages. We're going to construct a "talker" actor that knows how to say a few simple phrases in response to messages.

The messages we'll be sending are *tuples*—sequences of values. In Elixir, a tuple is written using curly brackets, like this:

```
{:foo, "this", 42}
```

This is a 3-tuple (or *triple*), where the first element is a keyword (Elixir's keywords are very similar to Clojure's, even down to the initial colon syntax), the second a string, and the third an integer.

Here’s the code for our actor:

Actors/hello_actors/hello_actors.exs

```
defmodule Talker do
  def loop do
    receive do
      {:greet, name} -> IO.puts("Hello #{name}")
      {:praise, name} -> IO.puts("#{name}, you're amazing")
      {:celebrate, name, age} -> IO.puts("Here's to another #{age} years, #{name}")
    end
    loop
  end
end
```

We’ll pick through this code in more detail soon, but we’re defining an actor that knows how to receive three different kinds of messages and prints an appropriate string when it receives each of them.

Here’s code that creates an instance of our actor and sends it a few messages:

Actors/hello_actors/hello_actors.exs

```
pid = spawn(&Talker.loop/0)
send(pid, {:greet, "Huey"})
send(pid, {:praise, "Dewey"})
send(pid, {:celebrate, "Louie", 16})
sleep(1000)
```

First, we *spawn* an instance of our actor, receiving a *process identifier* that we bind to the variable pid. A process simply executes a function, in this case the loop() function within the Talker module, which takes zero arguments.

Next, we send three messages to our newly created actor and finally sleep for a while to give it time to process those messages (using sleep() isn’t the best approach—we’ll see how to do this better soon).

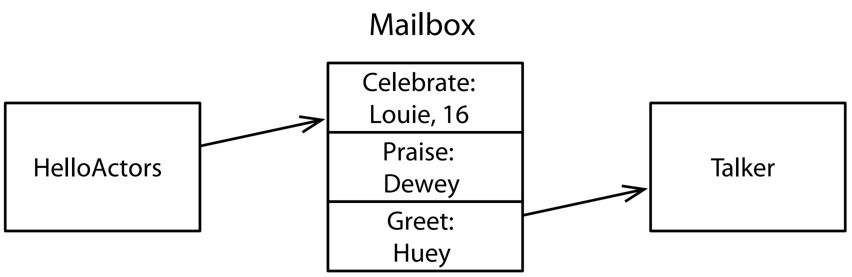
Here’s what you should see when you run it:

```
Hello Huey
Dewey, you're amazing
Here's to another 16 years, Louie
```

Now that we’ve seen how to create an actor and send messages to it, let’s see what’s going on under the hood.

Mailboxes Are Queues

One of the most important features of actor programming is that messages are sent *asynchronously*. Instead of being sent directly to an actor, they are placed in a *mailbox*:



This means that actors are *decoupled*—actors run at their own speed and don't block when sending messages.

An actor runs concurrently with other actors but handles messages sequentially, in the order they were added to the mailbox, moving on to the next message only when it's finished processing the current message. We only have to worry about concurrency when sending messages.

Receiving Messages

An actor typically sits in an infinite loop, waiting for a message to arrive with receive and then processing it. Here's Talker's loop again:

Actors/hello_actors/hello_actors.exs
```
def loop do
  receive do
    {:greet, name} -> IO.puts("Hello #{name}")
    {:praise, name} -> IO.puts("#{name}, you're amazing")
    {:celebrate, name, age} -> IO.puts("Here's to another #{age} years, #{name}")
  end
  loop
end
```

This function implements an infinite loop by calling itself recursively. The receive block waits for a message and then uses pattern matching to work out how to handle it. Incoming messages are compared against each pattern in turn—if a message matches, the variables in the pattern (name and age) are bound to the values in the message and the code to the right of the arrow (->) is executed. That code prints a message constructed using *string interpolation*—the code within each #{…} is evaluated and the resulting value inserted into the string.

The code on page 7 sleeps for a second to allow messages to be processed before exiting. This is an unsatisfactory solution—we can do better.

Joe asks:
Won’t Infinite Recursion Blow Up the Stack?

You might be worried that a function like Talker’s loop(), which recurses infinitely, would result in the stack growing forever. Happily, there’s no need to worry—in common with many functional languages (Clojure being a prominent exception—see *What Is Loop/Recur?*, on page ?), Elixir implements *tail-call elimination*. Tail-call elimination, as its name suggests, replaces a recursive call with a simple jump if the last thing the function does is call itself.

Linking Processes

We need two things to be able to shut down cleanly. First we need a way to tell our actor to stop when it’s finished processing all the messages in its queue. And second, we need some way to know that it has done so.

We can achieve the first of these by having our actor handle an explicit shutdown message (similar to the poison pill we saw in the code on page ?):

Actors/hello_actors/hello_actors2.exs
```
defmodule Talker do
  def loop do
    receive do
      {:greet, name} -> IO.puts("Hello #{name}")
      {:praise, name} -> IO.puts("#{name}, you're amazing")
      {:celebrate, name, age} -> IO.puts("Here's to another #{age} years, #{name}")
➤     {:shutdown} -> exit(:normal)
    end
    loop
  end
end
```

And second, we need a way to tell that it has exited, which we can do by setting :trap_exit to true and *linking* to it by using spawn_link() instead of spawn():

Actors/hello_actors/hello_actors2.exs
```
Process.flag(:trap_exit, true)
pid = spawn_link(&Talker.loop/0)
```

This means that we’ll be notified (with a system-generated message) when the spawned process terminates. The message that’s sent is a triple of this form:

```
{:EXIT, pid, reason}
```

All that remains is to send the shutdown message and listen for the exit message:

Actors/hello_actors/hello_actors2.exs

```
send(pid, {:greet, "Huey"})
send(pid, {:praise, "Dewey"})
send(pid, {:celebrate, "Louie", 16})
send(pid, {:shutdown})

receive do
  {:EXIT, ^pid, reason} -> IO.puts("Talker has exited (#{reason})")
end
```

The ^ (caret) in the receive pattern indicates that instead of binding the second element of the tuple to pid, we want to match a message where the second element has the value that's already bound to pid.

Here's what you should see if you run this new version:

```
Hello Huey
Dewey, you're amazing
Here's to another 16 years, Louie
Talker has exited (normal)
```

We'll talk about linking in much more detail tomorrow.

Stateful Actors

Our Talker actor is stateless. It's tempting to think that you would need mutable variables to create a stateful actor, but in fact all we need is recursion. Here, for example, is an actor that maintains a count that increments each time it receives a message:

Actors/counter/counter.ex

```
defmodule Counter do
  def loop(count) do
    receive do
      {:next} ->
        IO.puts("Current count: #{count}")
        loop(count + 1)
    end
  end
end
```

Let's see this in action in Interactive Elixir, iex (the Elixir REPL):

```
iex(1)> counter = spawn(Counter, :loop, [1])
#PID<0.47.0>
iex(2)> send(counter, {:next})
Current count: 1
{:next}
iex(3)> send(counter, {:next})
{:next}
Current count: 2
```

```
iex(4)> send(counter, {:next})
{:next}
Current count: 3
```

We start by using the three-argument form of `spawn()`, which takes a module name, the name of a function within that module, and a list of arguments, so that we can pass an initial `count` to `Counter.loop()`. Then, as we expect, it prints a different number each time we send it a `{:next}` message—a stateful actor with not a mutable variable in sight. And furthermore, this is an actor that can safely access that state without any concurrency bugs, because messages are handled sequentially.