The Actor Model

CSCI 5828: Foundations of Software Engineering
Lecture 16 — 10/16/2014

Goals

- Cover the material presented in Chapter 5, of our concurrency textbook
    - In particular, the material presented in Day 1
- Elixir
- Actors
    - Asynchronous message passing, message patterns

Elixir

- The Actor model of concurrency was developed alongside a functional programming language called Erlang
  - Erlang is 20-years old (!) and has been used heavily in the telecom industry
  - In Erlang, Actors are implemented by a framework called OTP
    - OTP has the ability to create thousands of actors, distribute them across multiple nodes in a cluster, and have them all run in parallel
- Our book describes the actor model via the use of Elixir, a new programming language that runs on top of the Erlang virtual machine
  - This is analogous to how Clojure and Scala run on top of the Java VM
    - Note: The Java VM is also 20 years old!

Installing Elixir

- Installing Elixir is straightforward
    - On Mac with Homebrew installed:
        - brew install elixir (this also installs Erlang and OTP)
    - On Windows and Linux
        - Follow the instructions here
            - <http://elixir-lang.org/install.html>
- To test if successful, try typing: `iex` ; you should see a banner and then a REPL prompt: `iex(1)>`
- I present a brief tutorial on Elixir; for more information
    - <https://pragprog.com/book/elixir/programming-elixir>

Hello World

- A simple Hello World program in Elixir
    - `IO.puts “Hello, CSCI 5828!”`
- This can be put in a file such as hello.exs
    - either .ex or .exs as file extensions; the latter is meant for scripts
- You can then run it from the command line with “elixir hello.exs”
    - or by invoking iex and typing c “hello.exs”
        - The c command is short for compile. iex compiles the file and executes any top level code

Assignment == Pattern Matching

- The first thing to learn about Elixir is that assignment statements are not the same as found in other programming languages
    - assignment statements invoke pattern matching
    - <symbol> = <value> in Elixir means “can I make the symbol on the left match the value on the right?”
        - a = 1
            - If a is unbound, then bind it to have a value 1; otherwise rebind it
        - 1 = a
            - This will work, since a currently equals 1 but if you tried
                - 2 = a, this will fail. There’s no way to rebind “2” to “1”

Pattern Matching is Powerful

- `list = [1, 2, [3, 4, 5]]`
    - Lists in Elixir are denoted with square brackets
        - values are separated by commas
- `[a, b, c] = list`
    - This performs a pattern match. Elixir tries to make the left side equal the right side; Therefore
        - `a => 1`
        - `b => 2`
        - `c => [3, 4, 5]`
- `[a, b, c, d] = list => MatchError`

Pattern Matching, explained

- A pattern on the left hand side of an assignment statement is matched if the values on the right hand side have the same structure as the pattern and each term in the pattern can be matched to the corresponding term in the values
  - literal values in the pattern must match the same value on the right hand side
  - variables (symbols) in the pattern match by taking on the corresponding value from the right hand side (as we saw with [a, b, c])
    - variables bind only once in a pattern; `[a, a] = [1, 2]` would fail
    - if you don't want a variable to be rebound, prefix it with a caret
    - `a = 2; [^a, 2] = [1, 2] => MatchError`
  - An underscore in the pattern matches any corresponding value from the right hand side
    - `[_, b, _] = [2, "ken", 3] => b == “ken”`

Immutability

- In Elixir, all values are immutable
    - large, deeply nested lists are treated the same as the integer 42
        - neither can be changed
    - Instead, as we saw with Clojure, if you want to transform a list, you take an existing list and then transform it in some way creating a new list
        - and, just as in Clojure, Elixir's collection classes are persistent (meaning they share as much structure as they can with previous versions of the list before they will create a new copy)

Types

- Elixir provides a wide range of types
    - Value Types: integers, floats, atoms (like symbols in Ruby; keywords in Clojure); ranges (5..15), and regular expressions
    - System Types:
        - pids: a “process id”; not a Unix process, an Elixir process
            - the function `self` will return the pid of the current process
        - ports: a reference to an external resource for reading/writing
        - refs: a globally unique id
    - Boolean values: true, false, nil
        - In boolean contexts, only false and nil evaluate to false; everything else evaluates to true

Collection Types

- Elixir has the following collection types
  - Tuples: an ordered collection of values
    - { 1, :ok, “hello” } — you can use tuples in pattern matching
      - We will use tuples to pass messages between actors
  - Lists — a linked data structure with a head and a tail
    - the head contains a value; the tail is another list; a list can be empty
  - Maps — a collection of key-value pairs
    - %{ key => value, key => value }

Functions

- Anonymous functions in Elixir are created using the `fn` keyword
  - sum = fn (x, y) -> x + y end
- Or generically
  - fn
    - parameter-list -> body
    - parameter-list -> body
  - end
- To invoke this function, you use the syntax
  - sum.(10, 15) => 25
- The . is needed when the function is anonymous
  - The parens are needed when calling, including functions with zero args
    - life = fn -> 42 end; life.() => 42

Pattern Matching Occurs on Function Calls

- `swap_me = fn ({a, b}) -> { b, a } end`
  - here the argument to the anonymous function is a 2-tuple
  - when you pass a tuple into the function, its parts are bound to a & b
- `swap_me.({23, 42}) => {42, 23}`

Shorthand Anonymous Functions

- `fn (n) -> n + 1 end`
  - is the same as
- `&(&1 + 1)`
- You can use this syntax when you need to pass a short function into some other function, such as map
  - `Enum.map [1, 2, 3, 4], &(&1 + 1)`
    - returns [2, 3, 4, 5]

What about named functions?

- You can have named functions in Elixir (such as Enum.map)
    - BUT
        - named functions have to be created inside of modules

- `defmodule Times do`
    - `def double(n) do`
        - `n * 2`
    - `end`
    - `def quadruple(n) do`
        - `double(n) * 2`
    - `end`
- `end`

- Use it like this: `Times.double(4); Times.double 21` <= parens are optional

Pattern Matching and Named Functions

- Pattern matching is used to determine which instance of a function should be invoked based on its arguments
  - To set this up, you write a function definition multiple times each with a different set of parameters that can be matched at run-time
- `defmodule Factorial do`
  - `def fact(0) do`
    - `1`
  - `end`
  - `do fact(n) do`
    - `n * fact(n-1)`
  - `end`
- `end`
- `Factorial.fact(1000) => 4023872600770937735…0000`

Actors

- Elixir makes use of a novel approach to concurrency, pioneered by Erlang, called the Actor model
  - In this model, actors are independent entities that run in parallel
  - Actors encapsulate state that can change over time
    - but that state is not shared with any other actor
    - As a result, there can be no race conditions
  - Actors communicate by sending messages to one another
    - An actor will process its messages sequentially
      - While many actors can run at once, each actor is itself a single-threaded sequential program
        - that's an abstraction which which developers are comfortable

Processes

- Actors are also called “processes”
    - In most programming languages / operating systems, processes are heavy weight entities
        - In Elixir, a process is very lightweight in terms of resource consumption and start-up costs; lighter weight even than threads
- Elixir programs might launch thousands of processes all running concurrently
    - and without having to create thread pools or manage the concurrency
- Instead, Elixir programs make sure the right processes get started and then get work done by passing messages to them

Messages and Mailboxes

- Messages in Elixir are asynchronous
    - When you send a message to an actor, the message is placed instantly in the actor's mailbox; the calling actor does not block
- Mailboxes in Elixir are queues
    - Actors perform work in response to messages
    - When an actor is ready, it pulls a message from its mailbox
        - and responds to it, possibly sending other messages in response
    - It then processes the next message and the next, until the mailbox is empty
        - at that point, it blocks waiting for a new message to arrive

First Example

```
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(“HB #{name}. #{age} years old!")
    end
    loop
  end
end
```

```
pid = spawn(&Talker.loop/0)
send(pid, {:greet, "Ken"})
send(pid, {:praise, "Lilja"})
send(pid, {:celebrate, "Miles", 42})
:timer.sleep(2000)       <= Note: change from book’s code
```

Discussion (I)

- The actor specifies what messages it can process with `receive`
  - Each message uses pattern matching
    - specifying a literal atom (:praise) and a variable that then matched whatever was sent with the rest of the message
      - {:praise, name} matches all 2-tuples that start with the :praise atom and then binds name to the second value
        - that binding can then be used in the message handler
          - `IO.puts("#{name}, you're amazing!”)`
  - The call to `receive` blocks the actor until there is a message to process
- The actor defines a single function: `loop`; `loop` is seemingly implemented as an infinite loop because after it calls receive, it calls loop and starts again
  - however, tail call elimination implements this with a goto
    - not a recursive call

Discussion (II)

- The rest of the code is used to actually create an actor (process) and send messages to it
    - since the message sends are asynchronous, this code ends with a call to :timer.sleep (actually an Erlang module/function) to allow time for the messages to be received
- The call to `spawn`, returns a process id that allows us to send messages to the actor with the function `send`. `send` takes a pid and a tuple, adds the tuple to the actor’s mailbox and returns immediately
    - The syntax &Talker.loop/0 is the same as &(Talker.loop/0) and essentially creates an anonymous function that refers to the specified function
        - the /0 refers to the arity of the function, loop takes zero parameters
        - The & shortcut is smart enough to skip creating an anonymous function, it just passes the original function to spawn.

Linking Actors

- We can establish better interactions with our actors if we link them
  - Linked actors get notified if one of them goes down
    - by either exiting normally or crashing
  - To receive this notification, we have to tell the system to “trap the exit” of an actor and send a message in the form: {:EXIT, pid, reason}
- We can modify our previous example to more cleanly shutdown by implementing another message
  - {:shutdown} -> exit(:normal)
- We then call Process.flag(:trap_exit, true) in our main program, change it to send the shutdown message, and then wait for the system generated notification that the Talker actor shutdown. **DEMO**

Maintaining State

- To maintain state in an actor, we can use pattern matching and recursion
  - `defmodule Counter do`
    - `def loop(count) do`
      - `receive do`
        - `{:next} ->`
          - `IO.puts(“Current count: #{count}”)`
          - `loop(count + 1)`
      - `end`
    - `end`
  - `end`
- `counter = spawn(Counter, :loop, [1])`
- `send(counter, {:next}) => Current count: 1`
- `send(counter, {:next}) => Current count: 2`

Hiding Messages

- You can add functions to your actor to hide the message passing from the calling code
- `def start(count) do`
  - `spawn(__MODULE__, :loop, [count])`
- `end`
- `def next(counter) do`
  - `send(counter, {:next})`
- `end`
- These functions can then be called instead
  - `counter = Counter.start(23)`
  - `Counter.next(counter) => Current count: 24`

Bidirectional Communication

- While asynchronous messages are nice
    - there are times when we will want to ask an actor to do something and then wait for a reply from that actor to receive a value or confirmation that the work has been performed
- To do that, the calling actor (or main program) needs to
    - generate a unique reference
    - call send with a message that includes its pid (obtained via self)
    - wait for a message that includes its ref and includes the response value
- Let's look at a modified version of count that returns the actual count rather than print it out

Receiving the Message in the Actor

- We update actor to expect the pid of the caller and the unique ref
  - `def loop(count) do`
    - `receive do`
      - `{:next, sender, ref} ->`
        - `send(sender, {:ok, ref, count})`
        - `loop(count + 1)`
    - `end`
  - `end`
- We now expect our incoming message to contain the sender’s pid and a unique ref. The :next atom still provides a unique “name” for the message
  - We send the current count back to the caller and pass back its ref too

Receiving the return value in the Caller

- The caller's code has to change as well

- `def next(counter) do`
  - `ref = make_ref()`
  - `send(counter, {:next, self(), ref})`
  - `receive do`
    - `{:ok, ^ref, count} -> count`
  - `end`
- `end`

- In this function, we call make_ref() to get a unique reference. We then send the :next message to the actor. We then block on a call to receive, waiting for the response.
  - The response's ref must match the previous value of ref (i.e. ^ref) and then binds the return value to the count variable which is then returned

Naming Actors

- You can associate names (atoms) with process ids, so you can refer to an actor symbolically
  - Process.register(pid, :counter)
    - this call takes a pid returned by spawn or spawn_link and associates it with the :counter atom
  - Now, when sending messages to that actor, you can use the atom
    - send(:counter, {:next, self(), ref})

Actors run in Parallel

- The book demonstrates that actors run in parallel with a simple implementation of a parallel map operation

```
• defmodule Parallel do
  • def map(collection, fun) do
    • parent = self()
    • processes = Enum.map(collection, fn(e) ->
      • spawn_link(fn()  ->
        • send(parent, {self(), fun.(e)})
      • end)
    • end)
    • Enum.map(processes, fn(pid) ->
      • receive do
        • {^pid, result} -> result
      • end
    • end)
  • end
• end
```

Using Parallel

- `slow_double = fn(x) -> :timer.sleep(1000); x * 2 end`
- `:timer.tc(fn() -> Enum.map([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], slow_double) end)`
- `:timer.tc(fn() -> Parallel.map([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], slow_double) end)`

- On my machine, the first call to :timer.tc returned
    - `{10010165, [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]} <= about 10 seconds`
- The second call returned
    - `{1001096, [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]} <= about 1 second`
- One process got launched per element of the input collection
    - they all waited one second, and then returned their result.
- In the first call to :timer.tc, the delay of one second occurred ten times sequentially; and so the entire call to Enum.map took 10 seconds

Summary

- We have had a brief introduction to the Elixir language
    - pattern matching (used everywhere)
    - value types
    - first-class functions
- We have also been introduced to the actor model
    - multiple actors run in parallel
        - each has its own mailbox and processes messages sequentially
    - if actors want work performed they send asynchronous messages to each other
        - if we need actors to wait for a response, we can do that with refs and calls to receive

Coming Up Next

- Lecture 17: Actors, Part Two
- Lecture 18: **Midterm Review**