The Actor Model, Part Two

CSCI 5828: Foundations of Software Engineering
Lecture 18 — 10/23/2014
Goals

• Cover the material presented in Chapter 5, of our concurrency textbook
  • In particular, the material presented in Days 2 and 3
Fibonacci Calculator (I)

• Let’s jump back into Elixir and the Actor model

  • This example is taken from the excellent Programming Elixir book from Pragmatic Programmers

• We’ll take a look at using Actors to calculate Fibonacci numbers

  • 0, 1, 1, 2, 3, 5, 8, 13, …

• Our example will calculate a set of Fibonacci numbers using a different number of actors; starting with one actor and proceeding up to ten actors running at once
Elixir Function Composition

• In order to understand the source code of the example, we must review Elixir’s function composition operator, also known as the “pipe operator”

• If you had a series of statement like this
  • a = f(x); b = g(a); c = h(b)

• You could also write it like this
  • c = h(g(f(x)))

• In Elixir, you would write it like
  • c = x |> f |> g |> h
    • x is piped into f, the result is piped into g, the result is piped into h

• The functions on the right hand side can have parameters
  • x |> f(y, z) is equivalent to calling f(x, y, z) —the thing being piped becomes the first argument of the function on the right hand side
To start our Fibonacci example, we first design two actors

• A solver: is able to calculate the nth Fibonacci number
• A scheduler: distributes calculation requests to a set of 1 or more solvers

A solver will sit in loop and do the following

• It sends {:ready, pid} to the scheduler
• It will then receive a :fib message asking it to calculate a number
• When it is done, it will send an :answer message to the scheduler

The solver will perform these actions until it receives a :shutdown message

• The scheduler will receive an array of integers that represent the Fibonacci numbers to calculate
  • it will send out :fib messages to :ready solvers until all requests are done
Fibonacci Calculator (III)

- The solver

```elixir
defmodule FibSolver do
  def fib(scheduler) do
    send(scheduler, {:ready, self})
    receive do
      {:fib, n, client} ->
        send(client, {:answer, n, fib_calc(n), self})
        fib(scheduler)
      {:shutdown} -> exit(:normal)
    end
  end
end

defp fib_calc(0) do 0 end
defp fib_calc(1) do 1 end
defp fib_calc(n) do fib_calc(n-1) + fib_calc(n-2) end
end
```

© Kenneth M. Anderson, 2014
defmodule Scheduler do
  def run(num_processes, module, func, to_calculate) do
    (1..num_processes) |> Enum.map(fn(_) -> spawn(module, func, [self]) end) |> schedule_processes(to_calculate, [])
  end
  
defp schedule_processes(processes, queue, results) do
    receive do
      {:ready, pid} when length(queue) > 0 ->
        [next | tail] = queue
        send(pid, {:fib, next, self})
        schedule_processes(processes, tail, results)
      {:ready, pid} ->
        send(pid, {:shutdown})
        if length(processes) > 1 do
          schedule_processes(List.delete(processes, pid), queue, results)
        else
          Enum.sort(results, fn({n1, _}, {n2, _}) -> n1 <= n2 end)
        end
      {:answer, number, result, _pid} ->
        schedule_processes(processes, queue, [ {number, result} | results])
    end
  end
end
Fibonacci Calculator (V): Main Program

to_process = [ 37, 37, 37, 37, 37, 37 ]

Enum.each(1..10, fn (num_processes) ->
  {time, result} =
    timer.tc(Scheduler, :run,
          [num_processes, FibSolver, :fib, to_process])

if num_processes == 1 do
  IO.puts inspect result
  IO.puts "\n # time (s)"
end
end

:io.format "~2B ~.2f~n", [num_processes, time/1000000.0]
Fibonacci Calculator (VI): Results

• On my 8-core machine, the results are:

  • #   time (s)
  • 1   6.22
  • 2   3.07
  • 3   2.10
  • 4   2.14
  • 5   2.43
  • 6   1.65 <= almost 4 times as fast
  • 7   1.72
  • 8   1.77
  • 9   1.78
  • 10  1.89 <= roughly 3.3 times as fast on average
Discussion

• Striking how simple the implementation of the FibSolver Actor is

  • small piece of code with a defined “message API”

  • program can then spin up as many of these actors as they want

• The scheduler is more complex BUT

  • it implemented scheduling in a very generic way

    • the function being calculated was completely abstracted away

    • the logic simply took care of doling out work to all ready actors

      • shutting down actors when there was no more work to be done

• With 11 active actors (10 solvers + 1 scheduler): Elixir has flexibility as to how those actors are distributed across the cores of the machine
Error Handling and Resilience

• Actors provide the ability to write fault-tolerant code

• We can assign a supervisor to a set of actors that detects when an actor has crashed and can do something about it

  • such as restart the actor

• They way they do this is by linking the actors together (as we saw in Lecture 16)

  • First: Process.flag(:trap_exit, true)

  • Second: pid = spawn_link(…)

  • Third: receive do {:EXIT, pid, reason}

• We’re going to build up an example that demonstrates these concepts
An Actor to Test Links: LinkTest

```elixir
defmodule LinkTest do
  def loop do
    receive do
      {:exit_because, reason} -> exit(reason)
      {:link_to, pid} -> Process.link(pid)
      {:EXIT, pid, reason} -> IO.puts("#{inspect(pid)} exited because #{reason}"), loop
    end
  end
  def loop_system do
    Process.flag(:trap_exit, true), loop
  end
end
```

An actor that can link to other actors via :link_to; otherwise it can be told to die by sending it a :exit_because message.

If we want to receive :EXIT messages, we need to invoke this actor with the loop_system call. Otherwise, we can just call loop to see what happens when an actor exits for a non :normal reason.
Example: Linked Actors; Non-Normal Exit

• Create two instances of the actor
  • pid1 = spawn(&LinkTest.loop/0)
  • pid2 = spawn(&LinkTest.loop/0)

• Link them (links are bidirectional)
  • send(pid1, {:link_to, pid2})

• Tell one to quit for a non-normal reason (it doesn’t matter which actor)
  • send(pid2, {:exit_because, :bad_thing})

• The result?
  • BOTH actors die; no :EXIT message received
Example: Linked Actors; Normal Exit

- Create two instances of the actor
  
  • \( \text{pid1} = \text{spawn}(\&\text{LinkTest.loop/0}) \)
  
  • \( \text{pid2} = \text{spawn}(\&\text{LinkTest.loop/0}) \)

- Link them (links are bidirectional)
  
  • \( \text{send(pid1, \{:link_to, pid2\})} \)

- Tell one to quit for a normal reason (it doesn’t matter which actor)
  
  • \( \text{send(pid2, \{:exit_because, :normal\})} \)

- The result?
  
  • Actor 2 dies; Actor 1 lives; still no :EXIT message received
Example: Linked System Actors; Non-Normal Exit

• Create two instances of the actor
  
  • `pid1 = spawn(&LinkTest.loop_system/0)`
  
  • `pid2 = spawn(&LinkTest.loop/0)`

• Link them (links are bidirectional)
  
  • `send(pid1, {:link_to, pid2})`

• Tell one to quit for a normal reason (it doesn’t matter which actor)
  
  • `send(pid2, {:exit_because, :bad_thing})`

• The result?
  
  • Actor 2 dies; Actor 1 lives; :EXIT message received and logged
Creating a Supervisor

- We now have enough knowledge to create an actor and its supervisor
  - The textbook implements a simple “cache” actor and a supervisor that can detect when the cache goes down

- The cache actor can
  - receive a request to store something in the cache
  - receive a request to retrieve something in the cache
  - receive a request to return the size of the cache (in bytes)

- The supervisor will create a cache actor and monitor its status
  - If it goes down, it will restart the cache
Cache

We can cause this actor to crash by sending nil for page in a :put message

```elixir
defmodule Cache do
  def loop(pages, size) do
    receive do
      {:put, url, page} ->
        new_pages = Dict.put(pages, url, page)
        new_size = size + byte_size(page)
        loop(new_pages, new_size)
      {:get, sender, ref, url} ->
        send(sender, {:ok, ref, pages[url]})
      loop(pages, size)
      {:size, sender, ref} ->
        send(sender, {:ok, ref, size})
      loop(pages, size)
      {:terminate} -> # Terminate request - don't recurse
    end
  end
end
```
Cache Helper Routines

These functions provide an “API” to the Cache. We can call them and not worry about starting actors and sending messages.
defmodule CacheSupervisor do
  def start do
    spawn(__MODULE__, :loop_system, [])
  end

  def loop do
    pid = Cache.start_link
    receive do
      {:EXIT, ^pid, :normal} ->
        IO.puts("Cache exited normally")
        :ok
      {:EXIT, ^pid, reason} ->
        IO.puts("Cache failed with reason \
          #{inspect reason} - restarting it")
      end
    loop
    end
  end

  def loop_system do
    Process.flag(:trap_exit, true)
    loop
  end
end

Start up a Cache. If it crashes, restart it; otherwise quit

Make sure we call :trap_exit to receive :EXIT messages
Discussion (I)

• This example illustrates a generic approach to concurrent actor systems
  • Keep the supervisors as small and as simple as possible
    • So simple that they are easy to debug and get correct
  • Have the actors that they supervise crash when things go wrong
    • Let the supervisors detect those crashes and decide what to do
  • This approach maximizes simplicity
    • rather than adding lots of error checking code in the workers
      • implement the success case and let all error cases cause a crash that gets handled by the supervisor => a nice separation of concerns
Discussion (II)

• This example is so generic that most of the work that we did manually has been implemented in a library called OTP

  • Let’s take a look at an OTP version of the Cache and CacheSupervisor

• A worker will make use of a library known as GenServer

  • It can handle “calls” and “casts”

    • the former return a result; the latter do not

• A supervisor will make use of a library known as Supervisor

  • A supervisor has an init method that specifies

    • a list of workers and a restart strategy

    • We use the :one_for_one strategy to specify that crashed workers should simply be restarted
defmodule CacheSupervisor do
  use Supervisor
  def start_link do
    supervisor.start_link(__MODULE__, [])
  end
  def init(_args) do
    workers = [worker(Cache, [])]
    supervise(workers, strategy: :one_for_one)
  end
end
defmodule Cache do
  use GenServer

  def handle_cast({:put, url, page}, {pages, size}) do
    new_pages = Dict.put(pages, url, page)
    new_size = size + byte_size(page)
    {:noreply, {new_pages, new_size}}
  end

  def handle_call({:get, url}, _from, {pages, size}) do
    {:reply, pages[url], {pages, size}}
  end

  def handle_call({:size}, _from, {pages, size}) do
    {:reply, size, {pages, size}}
  end
end
Helper Functions for Cache

```
def start_link do
  :gen_server.start_link({:local, :cache}, __MODULE__, {HashDict.new, 0}, [])
end

def put(url, page) do
  :gen_server.cast(:cache, {:put, url, page})
end

def get(url) do
  :gen_server.call(:cache, {:get, url})
end

def size do
  :gen_server.call(:cache, {:size})
end
```
Using the new version

• Start by creating the supervisor (which creates the Cache, its worker)

  • CacheSupervisor.start_link

• Then just use the Cache

  • Cache.size => 0
  • Cache.put “foo”, “bar” => :ok
  • Cache.size => 3
  • Cache.put “ohnoes”, nil => error message; auto restart
  • Cache.size => 0

• Just like that, we’ve reimplemented the previous example
Nodes and Distribution

• The Erlang virtual machine is used to execute Elixir programs

• In an analogous way that Coljure programs compile down to Java bytecodes and are executed by the Java Virtual Machine

• One cool feature of Erlang virtual machines is that they have the capability to act as nodes that can form clusters

• Elixir actors running on one node can easily route messages to actors running on other (possibly) distributed nodes

• To set this up in Elixir, you can launch iex and give it a node name

  • For security reasons, you also give it a “cookie”; only nodes with the same “cookie” can talk to one another

    • `iex --name node2@128.138.72.226 --cookie jiriki ← can be any string`
Connecting Nodes

• Once you have launched a node, you need to tell it about the other nodes
  • iex --name node2@128.138.72.226 --cookie jiriki
  • iex --name node1@128.138.72.226 --cookie jiriki

• Checking status
  • node1> Node.self => :"node1@128.138.72.238"
  • node2> Node.self => :"node2@128.138.72.226"

• Connecting
  • node1> Node.connect(:"node2@128.138.72.226") => true

• Both nodes are now connected to each other
  • node1> Node.list => [:"node2@128.138.72.226"]
  • node2> Node.list => [:"node1@128.138.72.238"]
Sending Code Between Nodes

- Let’s define a function
  ```elixir
  node1> whoami = fn () -> IO.puts(Node.self) end
  ```
- And send it to another node to be executed
  ```elixir
  node1> Node.spawn(:node2@128.138.72.226, whoami)
  ```
  **node1 REPL prints:** node2@128.138.72.226
- Pause to think about what we just did and how easy it was
  - We just
    - defined a function
    - sent it over to another machine as data
    - that machine converted the data back to a function
    - executed it
    - sent back the result
    - and our original machine then displayed the result
Sending Messages Between Nodes: Set-Up

• Let’s launch our Counter actor on node2
  
  • \$node2> pid = spawn(Counter, :loop, [42])

• Now, let’s register that process id and associate it with a global name
  
  • \$node2> :global.register_name(:counter, pid) => :yes

• In this context, “global” means across all connected nodes

• So, now on node1, we can look that name up
  
  • \$node1> pid = :global.whereis_name(:counter) => #PID<9027.73.0>

• Then we can send messages to it

  • (next slide)
Sending Messages Between Nodes

• This version of counter expects a message of the form
  
  • {:next, <caller_pid>, <unique_ref>}

• It then sends back a message of the form
  
  • {:ok, <unique_ref>, count}

• So, to call this Actor from node1, we do
  
  • node1> ref = make_ref
  
  • node1> send(pid, {:next, self, ref})
  
  • node1> receive do {:ok, ^ref, count} -> count end

• Sure enough, we get back the result 42
Just scratched the surface

• With these building blocks, you can move on to create full-fledged distributed, concurrent programs

• Start a bunch of actors on one or more “worker” machines and register their pids via the :global registry

• Start a supervisor on another machine and have it dole out work to the actors using a message pattern similar to the Fibonacci example

• If any of the workers die, have the supervisor restart them automatically

• I highly recommend the Programming Elixir book if you’re curious to see more complicated examples

• It shows an example that starts a server, has it handle requests, then modifies the code of the server, and HOT SWAPS that code into the running server => modifying servers without having to restart them!
Summary

• The Actor model is a powerful model for creating distributed, concurrent systems

• Any individual actor is a single-threaded program with state that changes in well defined ways

• Software design becomes “message design” and system design becomes balancing where actors live and how “message load” is distributed across them

• The one danger in Actor systems is deadlock; “receive” is a blocking call to avoid that, you can have receive timeout and have the Actor do something to recover

• The OTP library can be used to create Client-Server and Cluster-based applications with a minimal amount of code
Coming Up Next

• Lecture 19: Introduction to Software Design
• Lecture 20: The Design of Design, Part One