BUILD YOUR OWN SMALL SCALE OTP

Feedback and questions

Claudio Ortolina

Head of Elixir @ Erlang Solutions Ltd.

- Email: claudio.ortolina@erlang-solutions.com
- Twitter: @cloud8421

Welcome

Welcome! In this tutorial we'll take a look at how to implement a simplified version of `GenServer` and `Supervisor` behaviours. By doing this from scratch, we'll get some insights around the following topics:

- Message passing between processes
- Selective receiving of messages
- Timeouts
- Synchronous replies
- Behaviours
- Links
- Failure handling (e.g. trapping exits)

Why is this useful?

A good rule of thumb when writing Elixir or Erlang is to start with OTP constructs. The main reason is that it helps removing a whole class of concurrency-related bugs which could be introduced by writing low-level code by hand.

OTP is often seen as a blackbox, preventing people from understanding it properly. By rebuilding some of its core functionality from scratch, we'll gain a better understanding on how to use it.

Message passing

1. Setup

Let's start by typing:

```
$ mix new otp
```

This will scaffold a barebones elixir project, ready for us to start working.

2. Basic message passing

We'll start with a simple echo process, which replies with whatever message we send to it.

Let's write a test first in `test/otp/echo_test.exs`:

```
defmodule OTP.EchoTest do
  use ExUnit.Case

  alias OTP.Echo
```

```
  test "echo" do
    {:ok, pid} = Echo.start_link()
    Echo.send(pid, :hello)
    assert_receive :hello
    Echo.send(pid, :hi)
    assert_receive :hi
  end
end
```

For the implementation, let's create a `lib/otp/echo.ex` file:

```
defmodule OTP.Echo do
  def start_link do
    pid = spawn_link(__MODULE__, :loop, [])
    {:ok, pid}
  end

  def send(pid, msg) do
    Kernel.send(pid, {msg, self()})
  end

  def loop do
    receive do
      {msg, caller} when is_pid(caller) ->
        Kernel.send(caller, msg)
        loop()
    end
  end
end
```

The core of the implementation is the `loop/0` method, which blocks on any message with a tuple shape and a pid as a second argument, replying back to the caller with the same term.

EXERCISE:

- Drop non-compliant messages

3. Timeout support

The `OTP.Echo` process will block forever. What if we want to support a timeout? If no messages are received in X milliseconds, the process dies.

Let's test this:

```
test "timeout" do
  {:ok, pid} = Echo.start_link()
  Process.sleep(50)
  assert false == Process.alive?(pid)
end
```

For the implementation:

```
def loop do
  receive do
    {msg, caller} ->
      Kernel.send(caller, msg)
      loop()
  after
    10 ->
      exit(:normal)
  end
end
```

We explicit exit with a `:normal` reason, which means that this is an expected outcome in the process lifetime.

4. Receive replies

So far we relied on "inspecting" the caller process mailbox to prove that the `Echo` process replies back to the caller. More often than not we need synchronous replies, so let's implement that.

Let's start by revising our tests:

```
defmodule OTP.EchoTest do
  use ExUnit.Case
```

```
  alias OTP.Echo

  test "timeout" do
    {:ok, pid} = Echo.start_link()
    Process.sleep(50)
    assert false == Process.alive?(pid)
  end

  test "async echo" do
    {:ok, pid} = Echo.start_link()
    Echo.async_send(pid, :hello)
    assert_receive :hello
    Echo.async_send(pid, :hi)
    assert_receive :hi
  end

  test "sync echo" do
    {:ok, pid} = Echo.start_link()
    assert :hello == Echo.sync_send(pid, :hello)
  end
end
```

Note that we now want to have `async_send/2` and `sync_send/2`.

```
def async_send(pid, msg) do
  Kernel.send(pid, {msg, self()})
end

def sync_send(pid, msg) do
  async_send(pid, msg)
  receive do
    msg -> msg
  end
end
```

This is a very naive implementation, as it suffers from two major issues, which we'll tackle straight away.

5. Caller timeouts

The first issue is that the caller will wait forever for a reply. We can fix it by adding an `after` clause to the `sync_send/2` method.

```
test "sync send timeout" do
  {:ok, pid} = Echo.start_link()
  assert {:error, :timeout} == Echo.sync_send(pid, :no_reply)
end
```

Note that to make this pass in this scenario we need to "artificially" create a `receive` clause where we just don't reply back to the caller.

```
@loop_timeout 10
@sync_send_timeout 200

def sync_send(pid, msg) do
  async_send(pid, msg)
  receive do
    msg -> msg
  after
    @sync_send_timeout -> {:error, :timeout}
  end
end

def loop do
  receive do
    {:no_reply, _caller} ->
      loop()
    {msg, caller} ->
      Kernel.send(caller, msg)
      loop()
  after
    @loop_timeout -> :normal
  end
end
```

6. Caller race conditions

The second bug is a bit more subtle: as the caller blocks on *any* incoming message, it's subject to race conditions:

- **p1** (the caller) starts a `sync_send/2` call. Before **p2** (the `Echo`) replies, a third process **p3** sends a message to p1.
- **p1** will receive **p3**'s message and return it as a result of `sync_send/2`.

We can write a test for this as follows:

```
test "sync send timeout race condition" do
  {:ok, pid} = Echo.start_link()
  Kernel.send(self(), :long_computation)
  assert :long_computation == Echo.sync_send(pid, :no_reply)
end
```

We can simulate an expensive message in the loop:

```
def loop do
  receive do
    {:no_reply, _caller} ->
      loop()
    {:long_computation, caller} ->
      Process.sleep (@sync_send_timeout + 1)
      Kernel.send(caller, :long_computation)
    {msg, caller} ->
      Kernel.send(caller, msg)
      loop()
  after
    @loop_timeout -> :normal
  end
end
```

This way, our assertion will pass.

Fixing this requires introducing references, i.e. unique[1] values that can be used to tag messages. Let's update our test:

```
test "sync send timeout race condition" do
  {:ok, pid} = Echo.start_link()
  Kernel.send(self(), :long_computation)
  assert {:error, :timeout} == Echo.sync_send(pid, :no_reply)
```

```
end
```

The test now expects the call to timeout.

The `OTP.Echo` module needs to make use of references in different places. Note that this means that most of the previous unit tests will now fail.

```
defmodule OTP.Echo do
  @loop_timeout 10
  @sync_send_timeout 200

  def start_link do
    pid = spawn_link(__MODULE__, :loop, [])
    {:ok, pid}
  end

  def async_send(pid, msg) do
    ref = make_ref()
    payload = {ref, self(), msg}
    Kernel.send(pid, payload)
    ref
  end

  def sync_send(pid, msg) do
    ref = async_send(pid, msg)
    receive do
      {^ref, msg} -> msg
    after
      @sync_send_timeout -> {:error, :timeout}
    end
  end

  def loop do
    receive do
      {_ref, _caller, :no_reply} ->
        loop()
      {ref, caller, :long_computation} ->
        Process.sleep (@sync_send_timeout + 1)
        Kernel.send(caller, {ref, :long_computation})
      {ref, caller, msg} ->
```

```
        Kernel.send(caller, {ref, msg})
        loop()
    after
      @loop_timeout -> :normal
    end
  end
end
```

EXERCISE:

- fix unit tests

Introducing GenServer

1. A Working example

To an exact idea of what we need to build, we'll start by implementing a stack process on top of the built-in GenServer.

Let's create a file called `lib/stack.ex`:

```
defmodule Stack do
  use GenServer

  ## PUBLIC

  def start_link(initial) do
    GenServer.start_link(__MODULE__, initial)
  end

  def push(pid, element) do
    GenServer.cast(pid, {:push, element})
  end

  def pop(pid) do
    GenServer.call(pid, :pop)
  end

  ## CALLBACKS
```

```
  def handle_call(:pop, _from, []) do
    {:reply, {:error, :empty}, []}
  end
  def handle_call(:pop, _from, [h | t]) do
    {:reply, {:ok, h}, t}
  end

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

We can then add a `test/stack_test.exs` file with a minimal test case:

```
defmodule OTP.StackTest do
  use ExUnit.Case

  test "push and pop" do
    {:ok, pid} = Stack.start_link([])
    assert {:error, :empty} = Stack.pop(pid)
    :ok = Stack.push(pid, 1)
    assert {:ok, 1} = Stack.pop(pid)
  end
end
```

We can now proceed to replace `GenServer` with our own implementation:

```
defmodule Stack do
  alias OTP.GenServer
  use GenServer
  …
end
```

2. Process scaffold

We can reuse most of what we've seen in the `Echo` server to scaffold our `OTP.GenServer` process. Let's start from the server lifecycle (`start_link/1` and `loop/1`):

```
defmodule OTP.GenServer do
    def start_link(initial_state) do
      pid = spawn_link(__MODULE__, :loop, initial_state)
    {:ok, pid}
    end
  …
  def loop(state) do
    receive do
      {:"$call", from = {ref, caller}, msg} ->
        IO.puts "handle sync"
        Kernel.send(caller, {ref, msg})
        loop(state)
      {:"$cast", _msg} ->
        IO.puts "handle async"
        loop(state)
      _other ->
        loop(state)
    end
  end
end
```

In detail:

- We keep recursing over the state in the loop (no mutation happens for now);
- Messages have a specific format which identifies **calls** (synchronous) and **casts** (asynchronous). Out of band messages are simply dropped.

We can provide an api to generate compliant messages:

```
defmodule OTP.GenServer do
  def cast(server, msg) do
    payload = {:"$cast", msg}
    Kernel.send(server, payload)
    :ok
  end

  def call(server, msg) do
    ref = make_ref()
    payload = {:"$call", {ref, self()}, msg}
```

```
    Kernel.send(server, payload)
    receive do
      {^ref, result} -> result
    after
      5000 -> {:error, :timeout}
    end
  end
    …
end
```

In the code above, `cast/2` fires and forgets, while `call/2` uses the caller `pid` and a `ref` to guarantee a correct response in a certain timeout.

4. State mutation

We can now focus on implementing state mutation, i.e. how we want to modify the state depending on the incoming messages.

To do that, our `OTP.GenServer` needs to be started with a callback module that can be used to operate on the state and compute a reply (only for `calls`) and a new state.

```
defmodule OTP.GenServer do
  def start_link(mod, loopdata) do
    pid = spawn_link(__MODULE__, :loop, [mod, loopdata])
    {:ok, pid}
  end

  def cast(server, msg) do
    payload = {:"$cast", msg}
    Kernel.send(server, payload)
    :ok
  end

  def call(server, msg) do
    ref = make_ref()
    payload = {:"$call", {ref, self()}, msg}
    Kernel.send(server, payload)
    receive do
      {^ref, result} -> result
    after
```

```
      5000 -> {:error, :timeout}
    end
  end

  def loop(mod, loopdata) do
    receive do
      {:"$call", from = {ref, caller}, msg} ->
        {:reply, response, newloopdata} = mod.handle_call(msg,
from, loopdata)
        Kernel.send(caller, {ref, response})
        loop(mod, newloopdata)
      {:"$cast", msg} ->
        {:noreply, newloopdata} = mod.handle_cast(msg, loopdata)
        loop(mod, newloopdata)
      _other ->
        loop(mod, loopdata)
    end
  end
end
```

We can transform `loop/1` to `loop/2`, referencing the callback module where appropriate. Note that we need to make the assumption that the target module implements `handle_call/3` and `handle_cast/2`, each one of them with predictable return values. We can formalise this assumption with a behaviour.

3. OTP.GenServer as a behaviour

A behaviour is a compile-time contract that declares the need for a certain module to implement certain functions.

An `OTP.GenServer` compliant callback module, for example, needs to implement `handle_call/3` and `handle_cast/2`. We can express that at the top of the module:

```
defmoudule OTP.GenServer do
  @type from :: {reference, pid}

  @callback handle_call(term, from, term)
      :: {:reply, term, term}
  @callback handle_cast(term, term)
      :: {:noreply, term}
end
```

We can also provide a `__using__/1` macro implementation to support the `use OTP.GenServer` notation:

```
defmacro __using__(_opts) do
  quote do
    @behaviour OTP.GenServer
  end
end
```

EXERCISE:

- Implement failure cases for `handle_cast/2` and `handle_call/3`

4. Naming a process

It's also possible to name a process: we can modify `OTP.GenServer.start_link/2` to accept a 3rd optional argument that contains a name:

```
defmodule OTP.GenServer do
  def start_link(mod, loopdata, opts \\ []) do
    case Dict.get(opts, :name) do
      nil ->
        pid = spawn_link(__MODULE__, :loop, [mod, loopdata])
        {:ok, pid}
      name when is_atom(name) ->
        pid = spawn_link(__MODULE__, :loop, [mod, loopdata])
        Process.register(pid, name)
        {:ok, pid}
      _invalid_name ->
        {:error, :invalid_name}
    end
  end
end
```

This way we can call:

```
OTP.GenServer.start_link(OTP.Stack, [], name: :my_name)
```

```
…
OTP.GenServer.cast(:my_name, {:push, 1})
```

Supervising a process

Process supervision is one of the most important features of OTP and it builds on top of a very simple idea.

When process spawns another one, it can link itself to it. That means that when the newly spawned process terminate, the other one will terminate as well.

In other words, links are **bidirectional**.

A process can link to another by either using `spawn_link/1-3` or `spawn/1-3` and then `Process.link/1`.

A process can also trap exits with `Process.flag(:trap_exit, true)`.

If process **A** links itself to process **B** and traps exits, it won't crash when **B** dies. Instead it will receive a message in the format of `{:EXIT, pid, reason}`. This allows **A** to decide how to react to **B** crashing.

Knowing this, a supervisor can be defined as a process that: - Defines an api to start other "children" processes - When starting a child, a supervisor keeps track of the child pid, together with the arguments used to spawn it - As it's trapping exits, it can re-spawn a child process when needed, using the information it stored when the process was spawned the first time

1. Boilerplate

We can start with a very simple boilerplate, modelled after what we've seen so far:

```
defmodule OTP.Supervisor do
  def start(name) do
    pid = spawn(__MODULE__, :init, [])
    Process.register(pid, name)
    {:ok, pid}
  end
```

```
  def init() do
    Process.flag(:trap_exit, true)
    loop([])
  end

  defp loop(children) do

  end
end
```

We can see that `start/1` accepts a name, so that we can easily reference our supervisor. We need to break our loop function into two stages:

- `init/0` will setup the `trap_exit` flag
- `loop/1` will enter the receive loop, having a list of running children as state.

2. Supervisor actions

In the receive loop we need to be able to handle a few actions:

- Start a new child
- Handle a child exit
- Stop all children

By knowing this, we can model `loop/1` as follows:

```
defp loop(children) do
    receive do
      {:start_child, callerpid, mod, func, args} ->
        pid = spawn_link(mod, func, args)
        send(clientpid, {:ok, pid})
        loop([{pid, mod, func, args}|children])
      {:EXIT, pid, _reason} ->
        newchildren = List.keydelete(children, pid, 0)
        child = List.keyfind(children, pid, 0)
        {pid, mod, func, args} = child
            newpid = spawn_link(mod, func, args)
            loop([{newpid, mod, func, args}|newchildren])
      :stop ->
        kill_children(children)
    end
```

```
end

defp kill_children(children) do
  killer = fn ({pid, _mod, _func, _args}) ->
    Process.exit(pid, :kill)
  end
  Enum.each(children, killer)
end
```

- Starting a child uses the familiar `{m, f, a}` format and sends a message back to the caller to notify it of the successful operation (`start_child/4` is left as an exercise). After a successful start, the pid and its related metadata is added to the loop data.
- On exit, the element in the loop data that corresponds to the dead pid is removed and a new one is started. It then gets added to the loop data
- The message `:stop` tells the supervisor to loop over all its children and exit them with a `:kill` reason.

---

1. References repeat after 2^{82} calls, so they can be considered unique for practical purposes.