The Erlanger Playbook

Loïc Hoguin

Preview, built on June 19, 2015

Contents

Part I

Code

Chapter 1

Special processes

Special processes are processes started using *proc_lib* that implement system messages handling. This includes but is not limited to the standard behaviors.

1.1 Why implement my own?

Behaviors are great. Erlang developers tend to use the *gen_server* behavior… a lot. *gen_fsm* and *gen_event* also have good use cases and all of them cover 99% of your needs. What's the remaining 1% then?

Because they are so generic, they also contain a lot of logic for cases that may not apply to you. This is no problem until this part of your program becomes a bottleneck. Then, and only then, you should consider writing a custom special process.

For example:

- You have a supervisor of worker processes, and a separate *gen_server* that keeps track of these worker processes to limit their number. The work is duplicated.
- You have a *gen_server* that is only used by local Erlang processes, but they make a very large number of calls. The generic call mechanism becomes the bottleneck.

> When the abstraction is inappropriate, you should ditch the gen_server and roll your own.
>
> — Joe Armstrong

1.2 Why not a normal process?

Do not ever write a normal process for any purpose other than performing an asynchronous function call, as covered previously.

Use a special process.

A special process will:

- Tell you which process is its parent.
- Die gracefully when its parent dies.
- Produce logs when it unexpectedly dies.
- Allow inspecting or replacing its state.

All of this is more than worth the extra five minutes to implement them properly.

1.3 Implementing a special process

A special process must be implemented using `proc_lib` and `sys`.

1.3.1 proc_lib

A process started using `proc_lib` always adds two pieces of information to the process dictionary: the initial function call of the process, and its parent and ancestors:

```
1> Pid = proc_lib:spawn_link(fun() -> receive after infinity -> ok end end).
<0.35.0>
2> process_info(Pid).
[{current_function,{prim_eval,'receive',2}},
 {initial_call,{proc_lib,init_p,3}},
 {status,waiting},
 {message_queue_len,0},
 {messages,[]},
 {links,[<0.33.0>]},
 {dictionary,[{'$ancestors',[<0.33.0>]},
              {'$initial_call',{erl_eval,'-expr/5-fun-3-',0}}]},
```

These two values are used by various debugging and inspecting tools.

A process started using `proc_lib` will produce crash report when it crashes, so long as SASL is enabled:

```
$ erl -boot start_sasl
...
1> proc_lib:spawn_link(fun() -> 1 = 2 end).

=CRASH REPORT==== 25-Apr-2015::14:11:51 ===
  crasher:
    initial call: erl_eval:-expr/5-fun-3-/0
    pid: <0.44.0>
    registered_name: []
    exception error: no match of right hand side value 2
    ancestors: [<0.42.0>]
    messages: []
    links: [<0.42.0>]
    dictionary: []
    trap_exit: false
    status: running
    heap_size: 233
    stack_size: 27
    reductions: 95
  neighbours:
    neighbour: [{pid,<0.42.0>},
                  {registered_name,[]},
                  {initial_call,{erlang,apply,2}},
                  {current_function,{io,execute_request,2}},
                  {ancestors,[]},
                  {messages,[]},
                  {links,[<0.27.0>,<0.44.0>]},
                  {dictionary,[]},
                  {trap_exit,false},
                  {status,waiting},
                  {heap_size,610},
                  {stack_size,30},
                  {reductions,1128}]
** exception exit: {badmatch,2}
```

Notice the initial call and ancestors values in the crash report.

Finally, `proc_lib` provides an optional feature: synchronous start of processes using an acknowledgement function.

1.3.2 sys

A process started using `proc_lib` must implement the *sys protocol*.

This provides you with additional debugging and tracing mechanisms. I have little use for these as Erlang has much better built-in tracing nowadays.

You will however be very interested in the inspecting power it provides:

- `sys:get_status/1` provides a complete peek into the process. This is optionally a formatted output if the format callback is exported.

- `sys:get_state/1` returns the state of the process itself. In a *gen_server* for example, this is the `State` variable.
- `sys:replace_state/2` allows you to replace this state.

All of this of course as the process is running. The getter functions are also safe to use in production; be careful when replacing the state, though.

A process that implements the sys protocol can also be paused and resumed at will. This is what enables the hot code upgrade mechanism where modules and their state are updated in a running node.

1.3.3 Asynchronous start implementation

The steps to implement a special process with asynchronous start are as follow:

1. Start the process with `proc_lib:spawn_link/1..4` or `proc_lib:spawn_opt/2..5`.
2. Write a receive loop.
3. Exit when the parent process dies. This means that if you trap exit signals you need to handle the message to exit when the parent does.
4. Handle system messages.
5. Implement the `system_continue/3`, `system_terminate/4` and `system_code_change/4` callbacks.

This results in the following code:

```
start_link() ->
    proc_lib:spawn_link(?MODULE, init, [self()]).

init(Parent) ->
    loop(Parent).

loop(Parent) ->
    receive
        %% Only required when trap_exit is enabled.
        {'EXIT', Parent, Reason} ->
            terminate(State, Reason, NbChildren);
        {system, From, Request} ->
            sys:handle_system_msg(Request, From, Parent, ?MODULE, [],
                {state, Parent});
        Msg ->
```

```
            error_logger:error_msg("Unexpected message ~p~n", [Msg]),
            loop(Parent)
    end.

system_continue(_, _, {state, Parent}) ->
    loop(Parent).

system_terminate(Reason, _, _, _) ->
    exit(Reason).

system_code_change(Misc, _, _, _) ->
    {ok, Misc}.
```

1.3.4 Synchronous start implementation

The steps to implement a special process with synchronous start are slightly different:

1. Start the process with `proc_lib:start_link/1..4`.
2. Call `proc_lib:init_ack/1` from the newly started process.
3. Continue from 2. in the previous section.

This results in the following code:

```
start_link() ->
    proc_lib:start_link(?MODULE, init, [self()]).

init(Parent) ->
    ok = proc_lib:init_ack(Parent, {ok, self()}),
    loop(Parent).

loop(Parent) ->
    receive
        %% Only required when trap_exit is enabled.
        {'EXIT', Parent, Reason} ->
            terminate(State, Reason, NbChildren);
        {system, From, Request} ->
            sys:handle_system_msg(Request, From, Parent, ?MODULE, [],
                {state, Parent});
        Msg ->
            error_logger:error_msg("Unexpected message ~p~n", [Msg]),
            loop(Parent)
    end.

system_continue(_, _, {state, Parent}) ->
    loop(Parent).

system_terminate(Reason, _, _, _) ->
    exit(Reason).
```

```
system_code_change(Misc, _, _, _) ->
    {ok, Misc}.
```

You should call *init_ack* when the initialization required for the process to run is complete. This doesn't necessarily mean all initialization is done, just that you know it will run properly.

1.4 Call

The call mechanism is one that is explained in details when you start learning Erlang. You only see the tip of the iceberg though, as the one built into OTP is much more complex because it needs to handle all edge cases.

This is the main chunk of the code for performing calls in OTP:

```
do_call(Process, Label, Request, Timeout) ->
  try erlang:monitor(process, Process) of
    Mref ->
      %% If the monitor/2 call failed to set up a connection to a
      %% remote node, we don't want the '!' operator to attempt
      %% to set up the connection again. (If the monitor/2 call
      %% failed due to an expired timeout, '!' too would probably
      %% have to wait for the timeout to expire.) Therefore,
      %% use erlang:send/3 with the 'noconnect' option so that it
      %% will fail immediately if there is no connection to the
      %% remote node.
      catch erlang:send(Process, {Label, {self(), Mref}, Request}, [ ↩
           noconnect]),
      receive
        {Mref, Reply} ->
          erlang:demonitor(Mref, [flush]),
          {ok, Reply};
        {'DOWN', Mref, _, _, noconnection} ->
          Node = get_node(Process),
          exit({nodedown, Node});
        {'DOWN', Mref, _, _, Reason} ->
          exit(Reason)
      after Timeout ->
          erlang:demonitor(Mref, [flush]),
          exit(timeout)
      end
    catch
    error:_ ->
      %% Node (C/Java?) is not supporting the monitor.
      %% The other possible case -- this node is not distributed
      %% -- should have been handled earlier.
      %% Do the best possible with monitor_node/2.
      %% This code may hang indefinitely if the Process
      %% does not exist. It is only used for featureweak remote nodes.
      Node = get_node(Process),
      monitor_node(Node, true),
      receive
        {nodedown, Node} ->
          monitor_node(Node, false),
          exit({nodedown, Node})
      after 0 ->
          Tag = make_ref(),
          Process ! {Label, {self(), Tag}, Request},
          wait_resp(Node, Tag, Timeout)
      end
  end.
```

You don't always need to handle these edge cases.

If the process that performs the call never uses a registered name, you don't need to resolve its pid.

If the process target of the call is not a C or a Java node, you can remove the code for handling their edge cases.

If the process target of the call is always a local pid, then the send cannot fail and you don't need to handle exceptions.

If your supervisor strategy is to restart the caller when the callee crashes, then you don't need to monitor.

All these conditions can end up removing a fair chunk of code that now never needs to be executed.

This advice bears repeating: don't do this unless you really need it!

1.5 Case study: custom supervisor

A few years ago a bottleneck was detected in the Ranch application. Ranch is an acceptor pool; an acceptor is a process dedicated to accepting connections.

In order to limit the number of concurrent connections, a new process was added that keeps track of this number and prevent acceptors from accepting too fast if it is too high.

Keeping track of running processes is what a supervisor already does with links. The new process was doing it with monitors. So every time a connection supervisor died, two processes would be notified about its death. This is not a big problem unless you need to accept connections at a very high rate, which we did.

The solution was to implement a custom supervisor that would also maintain a count of connections.

This enabled us to gain a few more things.

When the connection is accepted, the acceptor process sends it to the supervisor using only this code:

```
start_protocol(SupPid, Socket) ->
    SupPid ! {?MODULE, start_protocol, self(), Socket},
    receive SupPid -> ok end.
```

This is safe because the supervisor is always local, and because the Ranch supervision strategy ensures that the caller crashes when the callee crashes.

More savings included not having to pass around parameters all the time, not needing child specs or strategies inside the connection supervisor and more.

1.5.1 And more

As I wrote this I realized that we could save even more by merging the acceptor and the supervisor, in turn getting rid of the unique supervisor bottleneck. Indeed, if we have ten acceptors that are also supervisor, then we don't have the bottleneck that we had when there was ten acceptors for one supervisor.

Food for thoughts!