Advanced Programming

Erlang OTP

Mikkel Kragh Mathiesen
mkm@di.ku.dk

Department of Computer Science
University of Copenhagen

October, 2021

Adapted from slides by Ken Friis Larsen

Part I

Pre-lecture – Separation of Concerns

Recap – Stateful Server

- Organise your code in modules
- Functions are pure (stateless).
- Processes can be used as the guardians of state.
- We organise our code as micro-servers that manage some data (a.k.a. state) that can be manipulated via a client API (a.k.a. concurrent objects).
- Functions starts processes, processes runs functions, functions are defined in modules.

Separation of Concerns – API

```
-export([start/0, incr/1, decr_with/2, get_value/1]).

start() -> spawn(fun () -> loop(0) end).
incr(Cid) -> request(Cid, incr).
decr_with(Cid, N) -> request_reply(Cid, {decr_with, N}).
get_value(Cid) -> request_reply(Cid, get_value).
```

Separation of Concerns – Data Manipulation

```
init() -> [].

handle_request_reply({decr_with, N}, Count) ->
    {Count - N, ok};
handle_request_reply(get_value, Count) ->
    {Count, {ok, Count}}.

handle_request(Count, incr) ->
    Count + 1.
```

Separation of Concerns – Communication

```
request_reply(Pid, Request) ->
  Pid ! {self(), Request},
  receive
    {Pid, Response} -> Response
  end.

request(Pid, Request) ->
    Pid ! {request, Request},
    ok.
```

Separation of Concerns – Communication

```
loop(State) ->
    receive
        {request, Req} ->
            NewState = handle_request(Req, State),
            loop(NewState);
        {request_reply, From, Ref, Req} ->
            {NewState, Res} = handle_request_reply(Req, State),
            From ! {Ref, Res},
            loop(NewState)
    end.
```

Today's Menu

- Library code for making generic servers
- Open Telecom Platform (OTP)

Part II

Generic Servers

Generic Servers

- Goal: Abstract out the difficult handling of concurrency to a generic library
- The difficult parts:
  - The `start-request_reply(/request)-loop` pattern
  - Hot-swapping of code

Simple Server Library

```
start(Module) -> spawn(fun() -> State = Module:init(), loop(Module, State) end).

request(Pid, Request) -> Pid ! {request, Request}, ok.

request_reply(Pid, Request) ->
    Ref = make_ref(),
    Pid ! {request_reply, self(), Ref, Request},
    receive
        {Ref, Response} -> Response
    end.

loop(Module, State) -> receive
        {request, Req} ->
            NewState = Module:handle_request(Req, State),
            loop(Module, NewState);
        {request_reply, From, Ref, Req} ->
            {NewState, Res} = Module:handle_request_reply(Req, State),
            From ! {Ref, Res},
            loop(Module, NewState)
    end.
```

Behaviour

```
-callback init()
    -> State :: term().
-callback handle_request(Req :: term(), State :: term())
    -> State :: term().
-callback handle_request_reply(Req :: term(), State :: term())
    -> { Response :: term(), State :: term() }.
```

Example: Counter Callback Module, 1

```
-module(server_counter).

% Public API
-export([start/0, incr/1, decr_with/2, get_value/1]).

% For server module
-export([init/0, handle_request/2, handle_request_reply/2]).
-behaviour(server).

%% Interface
start() -> server:start(?MODULE).
incr(C) -> server:request(C, incr).
decr_with(C, N) -> server:request_reply(C, {decr_with, N}).
get_value(C) -> server:request_reply(C, get_value).
```

Example: Counter Callback Module, 2

```
%% Callback functions
init() ->
    0.

handle_request_reply({decr_with, N}, Count) ->
    {Count - N, ok};
handle_request_reply(get_value, Count) ->
    {Count, {ok, Count}}.

handle_request(incr, Count) ->
    Count + 1.
```

Counting is important

Suppose that we really most have a counter server running at all times.
Using rigorous testing we have a library without any bugs(?!?!).
However, we fear that we'll discover that we really, really, really want some new functionality. What to do?

Hot Code Swapping

```
swap_code(Name, Mod) -> request_reply(Name, {swap_code, Mod}).

loop(Module, State) ->
    receive
        {request, Req} ->
            NewState = Module:handle_request(Req, State),
            loop(Module, NewState);
        {request_reply, From, Ref, {swap_code, NewModule}} ->
            From ! {Ref, ok},
            loop(NewModule, State);
        {request_reply, From, Ref, Req} ->
            {NewState, Res} = Module:handle_request_reply(Req, State),
            From ! {Ref, Res},
            loop(Module, NewState)
    end.
```

Part III

Open Telecom Platform (OTP)

Open Telecom Platform (OTP)

- Library(/framework/platform) for building large-scale, fault-tolerant, distributed applications.
- A central concept is the OTP *behaviour*
- Some behaviours
  - `supervisor`
  - `gen_server`
  - `gen_statem` (or `gen_fsm`)
  - `gen_event`
- See `proc_lib` and `sys` modules for basic building blocks.

Using gen_server

- Step 1: Decide module name
- Step 2: Write client interface functions
- Step 3: Write the six server callback functions:
  - `init/1`
  - `handle_call/3`
  - `handle_cast/2`
  - `handle_info/2`
  - `terminate/2`
  - `code_change/3`

  (you can implement the callback functions by need.)

Part IV

Summary

Modularity

- Erlang offer different tools for modularity:
  - Functions
  - Modules
  - Processes
  - (Nodes, network, …)
- Be careful when crossing *trust boundaries*.
- Identify and document *assumptions*.

Summary

- Structure your code into the infrastructure parts and the functional parts.
- Use `gen_server` for building robust servers.