Erlang: Concurrencia y tolerancia a fallos

75.59 - Técnicas de programación concurrente I

Facundo Olano - facundo.olano@gmail.com

2021-07-06

Erlang: Concurrencia y tolerancia a fallos

Erlang

- Diseñado por Joe Armstrong en Ericsson (1986)
- Usado originalmente para switches de telefonía
- Implementa sistemas tolerantes a fallas
  - funcional
  - concurrente
  - observable
  - distribuido

Funcional

Erlang es un lenguaje funcional...

- Las variables son inmutables
- No hay sentencias, solo expresiones
- Se usa recursividad para controlar el flujo

...pero es pragmático

- Las funciones pueden tener efectos secundarios
- Hay estructuras que habilitan estado mutable global

Ejemplo (recursivo)

```
-module(example1).

-export([list_increase/1]).

%% Incrementar en 1 todos los elementos de la lista.
list_increase(List) ->
    list_increase(List, []).

list_increase([N | Rest], Result) ->
    list_increase(Rest, [N + 1 | Result]);

list_increase([], Result) ->
    lists:reverse(Result).
```

Ejemplo (funciones de alto orden)

```
-module(example2).

-export([list_increase/1]).

%% Incrementar en 1 todos los elementos de la lista.
list_increase(List) ->
    lists:map(fun(N) -> N + 1 end, List).
```

Ejemplo (comprensión de listas)

```
-module(example3).

-export([list_increase/1]).

%% Incrementar en 1 todos los elementos de la lista.
list_increase(List) ->
    [N + 1 || N <- List].
```

Concurrente

- Erlang implementa el modelo de actores en su Virtual Machine
  - Procesos livianos y aislados
  - La comunicación es por pasaje de mensajes (valores copiados)
- El runtime no ejecuta programas sino aplicaciones
  - Iniciamos el runtime con determinadas aplicaciones
  - Cada aplicación consiste en un conjunto de procesos

Concurrencia: primitivas

```
%% Obtener el id del proceso actual (el de la shell)
> ParentPid = self().
<0.84.0>

%% Iniciar un nuevo proceso con spawn
> spawn(fun() ->
          %% Enviar un mensaje al proceso de la shell
          ParentPid ! {self(), "hello world!"}
        end).
<0.88.0>

%% Recibir (bloqueando) un mensaje con el patrón {From, Message}
> receive
    {From, Message} ->
      %% Imprimir el valor recibido por stdout
      io:format("Received: ~p from ~p \n", [Message, From])
  end.
Received: hello world! from <0.88.0>
ok
```

Demo: calc_server

Procesos como elemento de diseño

- Como CSP/go-rutinas, pero no exactamente
  - En ambos casos la comunicación es por pasaje de mensajes
  - En erlang se modela el actor (proceso) y no el canal (mailbox)
- Como objetos, pero no exactamente
  - Cercano a la idea original de POO (Smalltalk)
  - Los procesos son baratos, pero no es práctico crear uno por cada entidad del dominio
- Siempre se trata de tolerancia a fallos!
  - Separar componentes para aislar y controlar sus modos de error
  - "Encapsulate what may crash"

VM: Modelo de memoria

- Cada proceso tiene su propio espacio de memoria
  - realiza su propio garbage collection
  - la mayoría guarda poco estado propio
  - en muchos casos mueren antes de necesitar recolectar basura
- Los valores son copiados al enviar mensajes
  - No hay que lidiar con estado compartido
  - se eliminan los *data races*
- Erlang provee mecanismos externos para evitar la copia de grandes datos
  - pasaje por referencia de binarios
  - Almacenamiento clave-valor (ETS)
  - Términos globales (persistent_term)

VM: Scheduler

- Erlang hace preemptive scheduling
  - se parece más a un S.O. que a otros lenguajes
- Ejecuta un scheduler por cada núcleo de la computadora
  - El scheduler asigna un numero de "reducciones" a cada proceso
  - Todas las operaciones consumen reducciones
  - La ejecución del proceso se interrumpe cuando terminan las reducciones y se pasa al siguiente proceso en la cola
- Erlang prioriza latencia sobre throughput
  - La tarea del scheduler implica un costo extra
  - Se garantiza un reparto equitativo de los recursos
  - Un proceso lento/trabajoso no puede afectar a los demás
  - Los sistemas suelen degradar "graciosamente" ante mayor carga

Concurrencia robusta

Los procesos son terminados en la presencia de errores.
Además del manejo tradicional (try/catch), Erlang da herramientas para propagar o delegar el manejo de errores hacia otros procesos.

- Links
- Traps
- Monitors

Demo: calc_sup

Behaviors

- OTP: Open Telecom Platform
  - Framework para hacer aplicaciones Erlang "estándar"
- Behaviors
  - Mecanismo de reuso de código
  - Permiten separar la parte genérica/reusable de un problema de lo específico
  - Similares a clases abstractas y *template method* en POO
- Algunos behaviors provistos por OTP:
  - gen_server
  - gen_event
  - gen_statem
  - supervisor
  - application

Ejemplo: calc_server como gen_server

Lo genérico: Iniciar un proceso nombrado, procesar recursivamente mensajes ingresantes, responder consultas
Lo particular: mantener un número y exponer operaciones para modificarlo

```
-behavior(gen_server).

%% gen_server callbacks
init([]) -> {ok, 0}.

handle_cast({add, N}, Acc) -> {noreply, Acc + N};
handle_cast({divide, N}, Acc) -> {noreply, Acc / N}.

handle_call(get, _From, Acc) -> {reply, Acc, Acc}.
```

Ejemplo: calc_server como gen_server

Lo genérico: Iniciar un proceso nombrado, procesar recursivamente mensajes ingresantes, responder consultas
Lo particular: mantener un número y exponer operaciones para modificarlo

```
%% API
start_link() ->
    gen_server:start_link({global, calc_server}, ?MODULE, [], []).

add(N) ->
    gen_server:cast({global, calc_server}, {add, N}).

divide(N) ->
    gen_server:cast({global, calc_server}, {divide, N}).

get() ->
    gen_server:call({global, calc_server}, get, _Timeout=1000).
```

Supervisores

Worker: realiza trabajo y puede fallar.
Supervisor: su tarea es reiniciar workers cuando mueren. Pueden supervisar workers o a otros supervisores formando jerarquías o "árboles" de supervisión
Configuración:

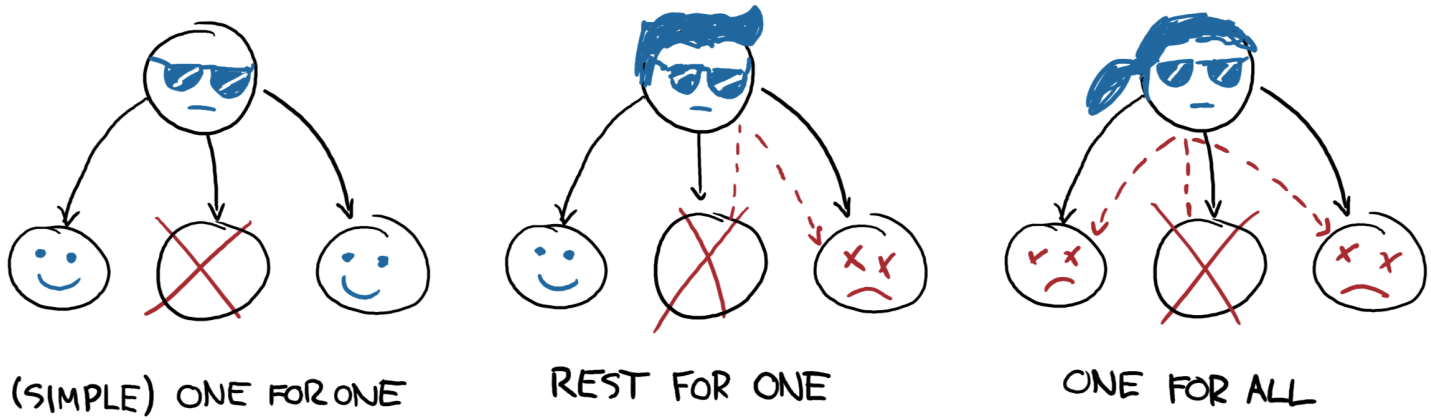
- Qué workers hay que iniciar y con qué parámetros
- La estrategia para propagar errores entre workers
- La frecuencia aceptable de errores

Ejemplo: calc_sup como supervisor

```
-behavior(supevisor).

init([]) ->
    SupervisorFlags = #{
      strategy => one_for_all, %% si falla un worker reiniciar todos
      intensity => 5,          %% hasta 5 restarts
      period => 60             %% cada 60 segundos
    },

    ChildSpec = [#{
      id => calc_server,
      start => {calc_server3, start_link, []},
      restart => permanent
    }, #{
      id => calc_loader,
      start => {calc_loader, start_link, []},
      restart => transient
     }],

    {ok, {SupervisorFlags, ChildSpec}}.
```

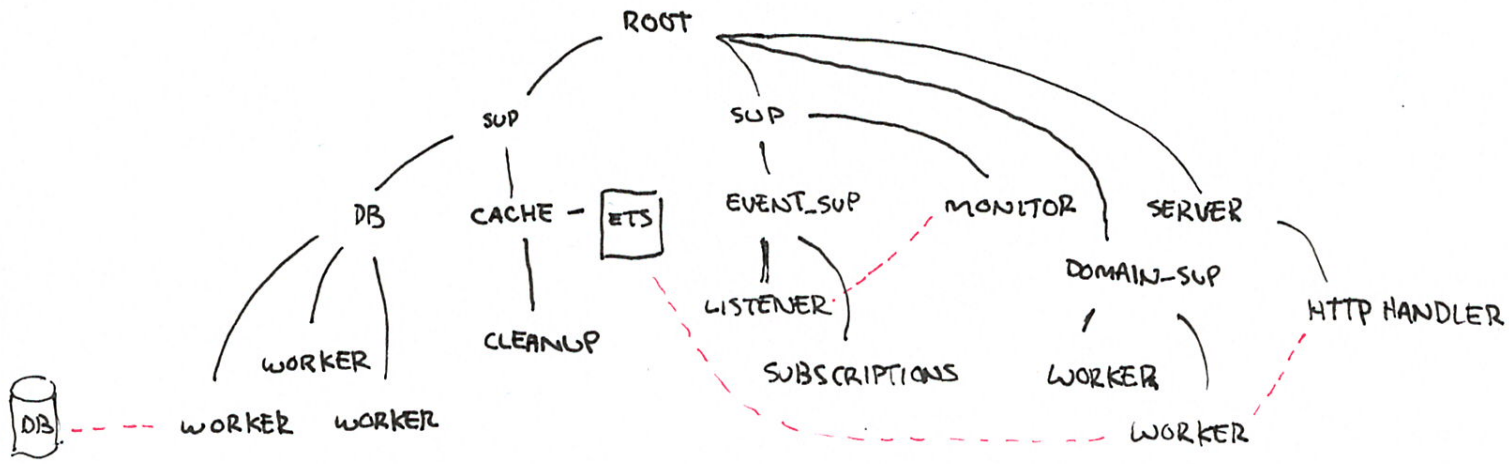
Supervisores: estrategias

Cómo un error en un worker debe afectar a sus pares



Fuente

Supervisores: árboles de supervisión



Fuente

- Los componentes se inician en profundidad, izquierda a derecha
- Los errores se propagan en profundidad, derecha a izquierda
- Cerca de la raíz están las "garantías" del sistema, lo que no puede fallar
- Cerca de las hojas lo más frágil, lo que esperamos que falle
- No se proveen garantías sobre la disponibilidad de sistemas externos (DB)

El Zen de Erlang: let it crash

Los crashes son inevitables: si los controlamos podemos usarlos como herramientas.
Let it crash (dejalo que se rompa)

- La mayoría de los errores son transitorios ("heisenbugs")
- En vez de tratar de predecirlos y manejarlos => Instruir al sistema para recuperarse
- En vez de escribir código defensivo => Dejá que el proceso muera y el supervisor lo reinicie
- El manejo de errores no está en la lógica sino en la estructura de la aplicación

Aún hay más

- Erlang distribuido
- Hot code reloading
- Introspección, observabilidad, tracing
- Elixir

Fuentes

- The Zen of Erlang
- Learn You Some Erlang for Great Good
  - The Hitchhiker's Guide to Concurrency
  - Errors and Processes
  - Who Supervises The Supervisors?
- An Open Letter to the Erlang Beginner (or Onlooker)
- How Erlang does scheduling
- Embrace Copying!
- Adopting Erlang - Supervision trees
- Spawned Shelter!
- Erlang: The Movie

¿Preguntas?