Una introducción a Erlang

Programación Funcional Avanzada

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad Simón Bolívar

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – concurrencia funcional

- Funcional – evaluación ambiciosa, sistema de tipos ligeros.
  - Recursión – explícita y de cola.
  - *Pattern matching* – inspirada en Prolog.
  - Monad implícito.
  - Verificaciones a tiempo de *ejecución*.
- Orientado a procesos – máquina virtual multihilo.
- Escalable – procesos económicos.
- Distribuido – múltiples nodos interconectados.
- Alta disponibilidad – procesos que cuidan procesos ante excepciones.

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – Tipos de datos

Tipos básicos

- Caracteres – $a, $0, ...
- Numéricos
  - Enteros nativos o de precisión arbitraria.
  - Punto flotante de 64 bits.
  - Bases arbitrarias para enteros.
- Atomos – la misma idea de Prolog

```
foo  'Esto es un atomo'
```

- Binarios – una secuencia arbitraria de bytes

```
<<4,8,15,16,23,42,...>>
```

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – Tipos de datos

Tipos compuestos

- Listas
  - Heterogéneas.
  - Construidas y procesadas con sintaxis similar a Prolog.

```
[1,foo,"una cadena",42.0]
[1 | [foo,"una cadena" | [42.0] ] ]
```

- Cadenas – lista de caracteres

```
"foo"    [$f,$o,$o]    [102,111,111]
```

- Tuplas

```
{1,foo,[42.0, 17],"cheezburger"}
```

- Records

```
-record(person,{name,ci}).

#person{name="iamemhn",ci=10041708}
```

Erlang – Tipos de datos

- Los números no son para cálculo numérico – use librerías externas.
- Las cadenas no son para procesamiento de datos – use librerías externas.

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – Tipos de datos

- Los números no son para cálculo numérico – use librerías externas.
- Las cadenas no son para procesamiento de datos – use librerías externas.
- Los binarios son para decodificar al vuelo

```
decode(Segment) ->
  case Segment of
    << SrcPort:16, DstPort:16, SeqNo:32, AckNo:32,
       DataOff:4, _Reserved:4, Flags:8, WinSize:16,
       Cksum:16, Urgent:16,
       Payload/binary >> when DataOffset > 4 ->

  OptSize = (DataOff - 5)*32,
  << Options:OptSize, Message/binary >> = Payload,
  ...
```

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – Tipos de datos

- Variables
  - Cualquier cosa que comienza con mayúscula – `Foo`.
  - Variables *don't care* comienzan con *underscore*.
  - Alcance limitado a la expresión – como en Prolog.
  - Una sola asignación.
- Funciones – *fun*s
  - No currificadas.
  - Definidas por casos – uso de patrones
  - Valores de primera clase.

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – programación secuencial

- Sumar los elementos de una lista.

```
sum(L) -> sum(L,0).

sum([],Acc)         -> Acc;
sum([N|Rest],Acc) -> sum(Rest,N+Acc).
```

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – programación secuencial

- Sumar los elementos de una lista.

```
sum(L) -> sum(L,0).

sum([],Acc)         -> Acc;
sum([N|Rest],Acc) -> sum(Rest,N+Acc).
```

- Construir lista entre `Min` y `Max`.

```
seq(Min,Max) -> seq(Min,Max,[]).

seq(N,M,Acc) when N =< M -> seq(N+1,M,[N|Acc]);
seq(_,_,Acc)             -> lists:reverse(Acc).
```

- Diferentes aridades – `sum/1` y `sum/2`.
- Llamadas completamente calificadas – `modulo:funcion(...)`.

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – Programación secuencial

- Quicksort – listas por comprensión

```
sort([Pivot|T]) ->
  sort([ X || X <- T, X <  Pivot ]) ++
  [Pivot] ++
  sort([ X || X <- T, X >= Pivot ]);
sort([]) -> [].
```

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – Programación secuencial

- Quicksort – listas por comprensión

```
sort([Pivot|T]) ->
  sort([ X || X <- T, X <  Pivot ]) ++
  [Pivot] ++
  sort([ X || X <- T, X >= Pivot ]);
sort([]) -> [].
```

- Clausuras instantáneas

```
> Suma = fun(N) -> fun(X) -> X + N end end.
#Fun
> G = Suma(10).
#Fun
> G(32).
42
```

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – Módulos

Espacio de nombres plano

- Exportar explícitamente funciones públicas indicando aridad.

```
-module(foo).
-export([bar/2,baz/3,qux/0]).
bar(X,Y) -> ...
baz(A,B,C) -> ...
qux() -> ...
```

- Invocar usando módulo y función – `foo:bar(42,X)`, `foo:qux()`.
- Compilan *por demanda* o *explícitamente* – extensión `.beam`.

```
1> c(foo).
{ok,foo}
2> foo:baz(42,"bleh",help)
...
```

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – Procesos

- Muy económicos – hilos dentro de la máquina virtual.
  - Función inicial al ser creados

```
Pid = spawn( funcion(...) ).
```

- Solamente intercambian mensajes
  - No hay estado mutable – no hay *deadlock* ni condiciones de carrera.
  - Envío asíncrono de *cualquier* estructura Erlang.

```
Pid ! {"wtf?",42,[foo,bar,baz]}
```

  - Recepción síncrona.
- Recepción selectiva de mensajes – *pattern matching*.

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – Procesos

Un “servidor” de área

```
-module(area)
-export([start/0,area/2,loop/2]).
start() -> spawn(?MODULE,loop,[0,0]).
loop(Ans,Bad) ->
  receive
    {Pid, {square,X}} -> Pid ! X*X,
                         ?MODULE:loop(Ans + 1,Bad);
    {Pid, {circle,R}} -> Pid ! 3.14*R*R
                         ?MODULE:loop(Ans + 1,Bad);
    {Pid, status}     -> Pid ! {served,Ans,bad,Bad},
                         ?MODULE:loop(Ans,Bad);
    {Pid, _}          -> Pid ! {error,"wtf?"},
                         ?MODULE:loop(Ans,Bad+1)
  end.
```

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – Procesos

Un “cliente” de área

```
area(X,Pid) ->
  Pid ! {self(),X},
  receive
    {served,A,bad,B} -> {status,A,B};
    {error,E}        -> {bad,E};
    R                -> {ok,R}
  after
    10 -> {error,timeout}
  end.
```

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – Procesos

Usando el “servidor” de área

```
1> Server = area:start();
<0.42.0>
2> area:area({square,4},Server).
{ok,16}
3> area:area("lolz",Server).
{bad,{error,"wtf?"}}
4> area:area(status,Server).
{status,1,1}
```

- Flexibilidad en los mensajes basados en estructuras de datos.
- Armar y desarmar respuestas intermedias.
- El servidor mantiene su estado en la clausura.

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – Procesos registrados

```
1> register(agrimensor,area:start()).
true

2> whereis(agrimensor).
<0.46.0>

3> area:area({square,9},agrimensor).
{ok,81}

4> exit(whereis(agrimensor),kill).
true.
```

- Cualquier átomo sirve para identificar al proceso.
- El proceso se registra por máquina virtual.
- `kill` es una de múltiples señales posibles – ¡atrapables!

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – Tolerancia a fallas

- `catch` permite atrapar excepciones

```
> X = (catch 1/0).
{'EXIT',{badarith,divide_by_zero}}
> b().
X = {'EXIT',{badarith,divide_by_zero}}
```

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – Tolerancia a fallas

- `catch` permite atrapar excepciones

```
> X = (catch 1/0).
{'EXIT',{badarith,divide_by_zero}}
> b().
X = {'EXIT',{badarith,divide_by_zero}}
```

- Cualquier estructura se vuelve excepción con `throw`.

```
throw({oh_noes,"help!"}).
```

- Aprovechar *Pattern matching*

```
case catch foo(X) ->
  {oh_noes,Error} -> ...;
  Normal          -> ...;
end.
```

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – Tolerancia a fallas

- Los procesos pueden enlazarase para definir cadenas de propagación de errores – `spawn_link`.

```
process_flag(trap_exits,true),
P = spawn_link(Node,Module,Func,Args),
...
```

- Cuando un proceso muere, los procesos enlazados reciben una indicación de terminación.
- Los procesos pueden reaccionar ante la señal – terminar, reiniciar, reportar, ...

```
receive
  {'EXIT',P,Why} -> ...
end
```

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – Distribuido

- Cada máquina virtual Erlang es un **nodo** – se le asigna un nombre.

```
$ erl -name foo@192.168.0.1 -cookie s3cr3t
1> register(magic,area:start()).
true
2> area:area({square,2},magic).
{ok,4}
```

- Accesibles usando nombre y dirección.

```
$ erl -name bar@192.168.0.2 -cookie s3cr3t
1> area:area({square,2},{magic,'foo@192.168.0.1'}).
{ok,4}
```

- Múltiples nodos en una misma máquina para probar – en diferentes máquinas para despliegues reales.

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – separando concurrencia de comportamiento

```
start(Name, Data, Fun) ->
    register(Name,
             spawn(fun() ->
                           loop(Data, Fun)
                   end)).
loop(Data, F) ->
     receive
         {query, Pid, Tag, Q} ->
             {Reply, Data1} = F(Q, Data),
             Pid ! {Tag, Reply},
             loop(Data1, F)
     end.
```

- Este servidor "hace" lo que haga `Fun` con `Data`.
- Detalles de concurrencia separados del comportamiento particular.

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – Reemplazo de código

```
loop(Data, F) ->
  receive
    {request, Pid, Q} ->
       {Reply, Data1} = F(Q, Data),
       Pid ! Reply,
       loop(Data1, F);
    {change_code, F1} ->
       loop(Data, F1)
  end
```

- El "trabajo" lo hace `F` – ¡cualquier F!
- Un comportamiento (`change_code`) permite *cambiar* la función por otra nueva – *non-stopping upgrade*.
- El recolector de basura hace desaparecer la vieja `F` cuando ningún proceso la esté utilizando.

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – Comportamientos

- **Comportamiento** (*behaviour*) es el nombre Erlang para servidores genéricos.
- **OTP** (*Open Telecom Platform*) es la librería estándar Erlang que provee comportamientos listos para usar.
  - Servidor HTTP.
  - Servidor FTP.
  - *Object Request Broker* CORBA.
  - Librerías H.248, SNMP, ASN.1, ...
- `gen_server` provee todo el andamiaje de concurrencia, tolerancia a fallas y distribución – uno escribe las funciones secuenciales (*hooks*).

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – datos distribuidos

Memoria compartida intra-nodo

- **ETS** (*Erlang Term Storage*) y **DETS** (*Disk Erlang Term Storage*)
- Diccionarios para almacenar cualquier tipo y cantidad de datos Erlang en un nodo.
  - Cada "tabla" almacena tuplas.
  - Semántica de conjuntos, multiconjunto o conjunto ordenado.
  - Tiempo constante de búsqueda – logarítmico en el peor caso.
  - Pueden compartirse entre varios procesos – mutables o inmutables.
- **ETS** en memoria – volátiles.
- **DETS** en disco – persistentes.

UNIVERSIDAD SIMÓN BOLÍVAR

Erlang – datos distribuidos

Memoria compartida inter-nodos

- **Mnesia** – la base de datos Erlang.
- Almacenaje de registros en tablas – memoria o disco.
- Acceso utilizando álgebra relacional – listas por comprensión y filtros.
- Transacciones – bloqueo pesimista.
- Replicación – memoria, disco, nodos, ...

```
mnesia:create_table(users,
  [{attributes,{login,password}},
   {disc_copies,[foo(),bar()]},
   {ram_copies,[foo(),baz()]}]).
```

UNIVERSIDAD SIMÓN BOLÍVAR