Stressfreies Multithreading mit Erlang

Michael Gebetsroither

19. April 2008

Inhalt der Präsentation

1. Intro
   - History
   - Besonderheiten
2. Basics
   - Datentypen
   - Sequenziell
3. Parallel
   - Prozesse
   - Distribution
   - OTP

History

- 1987: Erste Experimente mit Erlang
- 1990: Erste Implementierung am Ericsson Computer Science Laboratory
- 1993: Clusterfähig
- 1998: Erlang wird Open-source!

Erlang ist anders

- Funktionale Sprache
- Single Assignment
- Prozess-orientiert
- Skalierbar
- Clusterfähig
- Höchste Zuverlässigkeit als Designkriterium

Wieso ist Erlang anders

- Praktisch kein Prozesslimit
- Asynchrones Messagepassing
- Hot code-reloading
- Keine veränderbaren Datenstrukturen
- Sehr viele einfach verkettete Listen
- *Kein* Zuweisungs Operator
- Pattern-matching
- Tail-rekursive Funktionen
- Gesamter State am Stack

Collections

list

```
[1, 2, 3] = [1 | [2, 3]]
```

tuple

```
{fixed, sized, struct}
```

Basisdatentypen

- *atom*: Konstanter String, wird oft auch '*gequotet*'
- *binary*:

```
<<"testdaten als String", 1234:32/big>>
```

- *integer*: unlimitiert
- *float*: 64bit floating point
- *ref*: Eine garantiert einzigartige ID

```
> make_ref().
#Ref<0.0.0.47>
```

Andere Datentypen

- *fun*: Anonyme Funktion, kann gespeichert, serialisiert und zu anderen Nodes im Cluster geschickt werden.
- *pid*: Referenz zu einem Prozess, cluster aware

```
> self().
<0.45.0>
```

- *port*: Referenz zu einem Treiber, Socket, Pipe, ähnlich pid

```
> Port = erlang:open_port({spawn, "echo foo"}, []).
#Port<0.163>
> receive Data -> erlang:display(Data) end.
{#Port<0.163>,{data,"foo\n"}}
```

Es gibt kein(e) ...

- *boolean*: Werden durch die atoms true/false ausgedrückt
- *character*: Nur ein Integer

  ```
  $c = 99
  ```

- *string*: Nur eine Liste von Integern, oder ein Binary

  ```
  "abc" = [$a, $b, $c] = [97, 98, 99]
  ```

Variablen

- Beginnen mit einem Großbuchstaben
- Man kann einer Variable nur einmal einen Wert zuweisen (single-assignment)
- Zuweisung durch den Pattern-matching Operator

Es gibt keine direkte Zuweisung

'=' ist der Pattern-matching Operator!

Atoms

- Beginnen mit einem Kleinbuchstaben
- Müssen gequotet werden wenn sie mit einem Großbuchstaben beginnen
- Werden nicht garbage-collected

```
test
'Foo'
list_to_atom("test")
atom_to_list(test)
```

Patternmatching

```
A = foo
B = foo

A = B

{foo, foo} = {A, B}

% _ matched alles
{A, _} = {foo, bar}

% error, bad-match
A = bar.
```

Clouses, Guards

```
case date() of

  {2008, 4, D} when D == 19 ->
    glt08;

  {2009, 4, D} when D >= 5, D <= 28 ->
    probably_glt09;

  _ ->
    not_gt

end.
```

Bit-syntax

```
<<SeqNum:32/unsigned,
  SenderID:16/unsigned, Mux:16/unsigned,
  Rest/binary>> = Binary,
{SeqNum, SenderID, Mux, Rest}.

> <<Length:32, Rest:Length/binary>> = <<4:32, "test">>.
<<0,0,0,4,116,101,115,116>>
```

, ; oder doch .

- *Beistrich* ','
  Trennt Expressions
- *Strichpunkt* ';'
  Trennt verschiedene Sätze
- *Punkt* '.'
  Beendet den letzten oder einzigen Satz

Integer Accumulator (JavaScript)

```
function sum(list) {
    var acc = 0;
    for (var i = 0; i < list.length; i++) {
        acc += list[i];
    }
    return acc;
}
```

Schlechter Integer Accumulator

Kein Tail-call und benötigt daher O(n) Stack.

```
sum([]) ->
    0;
sun([Head | Tail]) ->
    N + sum(Tail).
```

Integer Accumulator

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

sum([], Acc) ->
    Acc;
sum([Head | Tail], Acc) ->
    sum(Tail, Acc + Head).

% oder besser
lists:sum(List).
```

Modul

- Flat namespace
- Exportierte Funktionen explizit angeben
- Modulname ist ein Atom
- Werden zu Bytecode compiliert (BEAM files)
- Können zur Laufzeit durch eine neue Version ersetzt werden.

Beispiel Modul

```
-module(hello_world).
-export([hello_world/0,
         hello_world/1]).

hello_world() ->
    {ok, "hello_world"}.

hello_world(Name) ->
    {ok, "hello_" ++ Name}.
```

Beispiel Modul aus der Shell

```
> c(hello_world).
{ok,hello_world}

> hello_world:hello_world().
{ok,"hello_world"}

> hello_world:hello_world("mars").
{ok,"hello_mars"}

% oops
> hello_world:hello_world('mars').
{ok,[104,101,108,108,111,95|mars]}
```

Beispiel Modul mit Guards

```
hello_world() ->
    {ok, "hello_world"}.

hello_world(Name)
    when is_list(Name), is_integer(hd(Name)) ->
    {ok, "hello_" ++ Name}.

> hello_world:hello_world('mars').
** exception error: no function clause matching \
    hello_world:hello_world(mars)
```

Prozesse

- Sehr schnell zu erzeugen (spawn)
- Kommunikation *nur* über asynchrones Message-passing
- dh. *kein* gemeinsamer State
- Selektives empfangen von Nachrichten

Basics

- Nicht defensiv coden
- Wenn etwas nicht funktioniert wird der Prozess beendet
- . . . und er wird vom Supervisor neu gestartet
- Trennung von Code und Fehlerbehandlung
- . . . führt zu sauberen leicht lesbaren Code
- Fehlertoleranter Code

Prozesse starten

```
-module(adder).
-export([start/0, loop/1, add/2]).

start() ->
    spawn(?MODULE, loop, [0]).
```

Prozess Code

```
loop(Value) ->
    receive
        {add, N, Absender} ->
            Absender ! {add_result, N + Value},
            ?MODULE:loop(N + Value);
        Other ->
            io:format("Unknown message: ~p~n", [Other]),
            ?MODULE:loop(Value)
    end.
```

Prozess API

```
add(Pid, Value) ->
    Pid ! {add, Value, self()},
    receive
        {add_result, Result} -> {ok, Result}
    after
        10 -> {error, timeout}
    end.
```

Prozess in der Shell

```
> c(adder).
ok

> Pid = adder:start().
<0.59.0>

> adder:add(Pid, 5).
{ok,5}

> Pid ! foo.
Unknown message: foo

> adder:add(Pid, 10).
{ok,15}
```

Registrierte Prozesse

```
> register(unser_adder, adder:start()).
true

> whereis(unser_adder).
<0.64.0>

> adder:add(unser_adder, 10).
{ok,10}

> exit(whereis(unser_adder), kill).
true
```

Unser Adder im Cluster

```
% erl -sname foo@localhost -setcookie geheim
(foo@localhost)1> register(unser_adder, adder:start()).
true
(foo@localhost)2> adder:add(unser_adder, 10).
{ok,10}

% erl -sname bar@localhost -setcookie geheim
(bar@localhost)1> adder:add({adder, foo@localhost}, 5).
{ok,15}
(bar@localhost)2> nodes().
[foo@localhost]
```

Monitoring im Cluster

```
(FOO)1> register(unser_adder, adder:start()).
true

(BAR)1> erlang:monitor(process, {adder, foo@localhost}).
#Ref<0.0.0.71>

(FOO)2> exit(whereis(adder), kill).
true

(BAR)4> receive Msg -> erlang:display(Msg) end.
{'DOWN',#Ref<0.0.0.62>,process,{adder,'foo@localhost'}, \
    killed}
```

Verlinkte Prozesse

- Werden mit spawn_link gestartet.
- Wenn einer der beiden Prozesse stirbt werden beide mit einer Exception beendet
- . . . außer wenn ein Prozess mit process_flag(trap_exit, true) sich die Exceptions als normale Nachrichten schicken lässt

Was ist OTP

- OTP == Open Telecom Plattform
- Bibliotheken um das schreiben von verteilten Anwendungen zu erleichtern
- vordefinierte Architekturmuster (gen_server, supervisor, gen_fsm)
- Enthält u.a auch eine soft-realtimefähige Datenbank mit Multi-Master Replizierung (mnesia)

Supervisor

- OTP-Applicationen sind ein Baum aus Supervisors
- Sie starten, überwachen und restarten ihre Child-prozesse
- Child-prozesse können selbst wieder Supervisors sein

gen_server Adder API

```
-module(adder_otp).
-behaviour(gen_server).
-export([start/0, add/2, stop/1]).
-export([init/1, terminate/2,
         handle_cast/2, handle_call/3]).

start() ->
    gen_server:start_link(?MODULE, 0, []).

add(Pid, Value) ->
    gen_server:call(Pid, {add, Value}).

stop(Pid) ->
    gen_server:cast(Pid, stop).
```

gen_server Adder Implementierung

```
init(Value) ->
    {ok, Value}.

terminate(_Reason, _Count) ->
    ok.

handle_cast(stop, Value) ->
    {stop, normal, Value}.

handle_call({add, N}, _From, Value) ->
    {reply, N + Value, N + Value}.
```

gen_server Adder in der Shell

```
> c(adder_otp).
> {ok, Pid} = adder_otp:start().
{ok,<0.66.0>}
> adder_otp:add(Pid, 10).
10
> adder_otp:add(Pid, foo).

=ERROR REPORT==== 18-Apr-2008::12:55:17 ===
** Generic server <0.66.0> terminating
** Last message in was {add,foo}
** When Server state == 10
** Reason for termination ==
** {badarith,[{adder_otp,handle_call,3},
              {proc_lib,init_p,5}]}
```

Flames

Aber Erlang ist ja interpretiert und daher langsam!

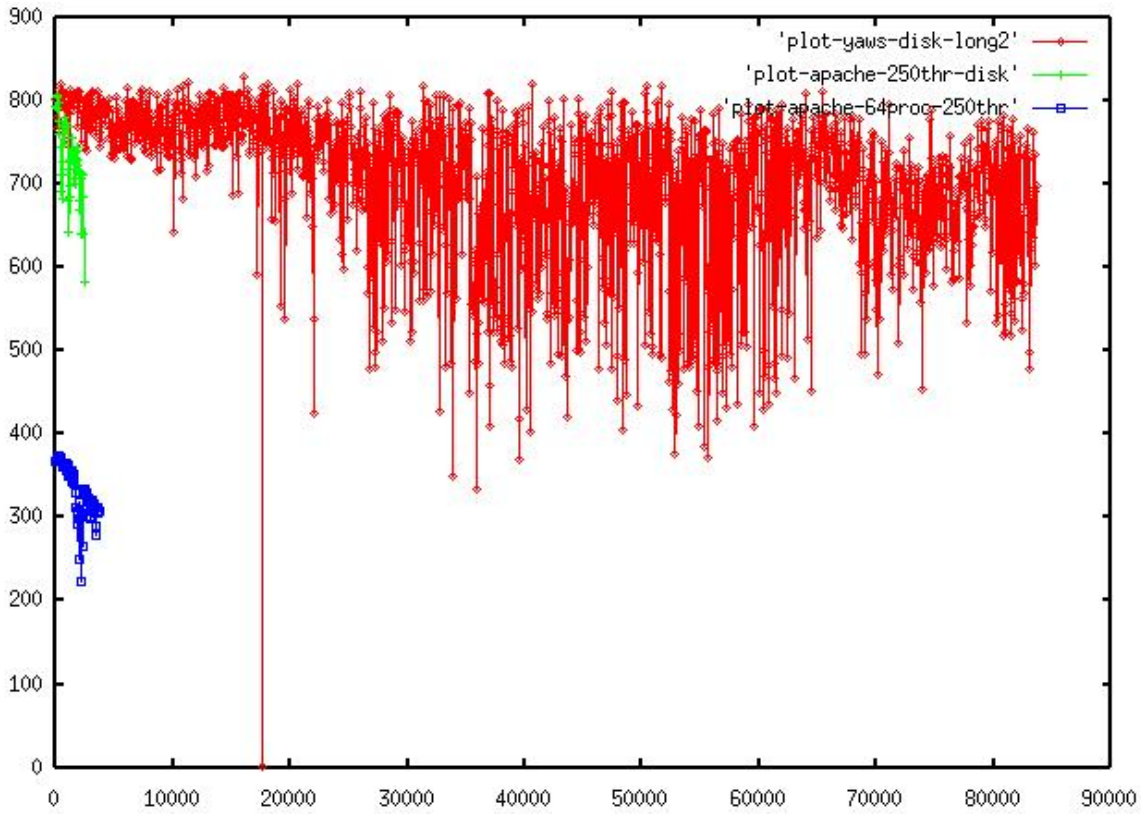

Abbildung: Apache vs. Yaws

Ende

Fragen?