

Articles



You are here » [Home Page](#) » [Articles](#) » Link and Monitor differences in Erlang

Manage Your
Clients' Maps.

Control any endpoint quickly
with SolarWinds® RMM's
Take Control feature.



solarwinds
msp

START YOUR
FREE TRIAL!

Link and Monitor differences in Erlang

Tweet


Link or Monitor processes in an Erlang Project

[link/1](#) and [monitor/2](#) are 2 different ways of notifying (or know) that a process died. Thing is, these are really very different in nature and these differences are not widely understood by beginners. So let's shed some light on this subject!

Linking to a process

[Links](#) are what makes it possible to have [supervisor trees](#). As stated in the [Error Handling section of the processes reference manual](#):

If you liked this or other articles (and feel generous), you can make a donation:

 [Click here to lend your support to: General and make a donation at \[pledgie.com\]\(#\) !](#)

Beat cyberattacks at their own game.

They never take a break — but neither does SolarWinds® RMM's managed antivirus protection.



solarwinds
msp

START YOUR FREE TRIAL!

Erlang has a built-in feature for error handling between processes. Terminating processes will emit exit signals to all linked processes, which may terminate as well or handle the exit in some way.

The signal in question is the exit signal, and the links make this signal propagate through processes (i.e: up in the supervisor hierarchy). By default, this signal makes your processes terminate as well. Assume the following code:

```

1  -module(mymodule).
2  start_link() ->
3      gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).
4  ...
5  crash() ->
6      gen_server:cast(?MODULE, crash).
7  ...
8  handle_cast(crash, State) ->
9      {stop, error, State};
10 ...

```

[mymodule.erl](#) hosted with ♥ by [GitHub](#)

[view raw](#)

Let's spawn of those, and try to link our shell to it:

```

1  1> self().
2  <0.31.0>
3  2> mymodule:start_link().
4  {ok,<0.34.0>}
5  3> mymodule:crash().
6
7  =ERROR REPORT==== 30-Dec-2012::15:36:47 ===
8  ** Generic server a terminating
9  ** Last message in was {'$gen_cast',crash}
10 ** When Server state == []
11 ** Reason for termination ==
12 ** error

```

Related Articles

Serve PHP applications with Erlang and Cowboy using FastCGI

Call FastCGI applications from Erlang

Erlang tutorial for PHP developers

ABNF Grammars in Elixir

Common Test: Cobertura Coverage report for Jenkins using covertool

Common Test: Generating JUnit style xml reports for Jenkins

Epers, Erlang, and Domain Events when persisting your entities

Persisting your entities in Erlang

Erlang Special Processes without behaviours

Erlang Websocket Server using Cowboy

Common Test: Code coverage on subdirectories

How to generate code coverage for eunit tests when using rebar

How to skip compile and eunit test for dependencies in rebar

```
13  ** exception exit: error
14  4> self().
15  <0.36.0>
```

log.txt hosted with ❤ by [GitHub](#)

[view raw](#)

As you can see, using `gen_server:start_link/4` automatically creates a link between our shell and the newly started process. So when this new process terminates, our shell gets an exit signal, also crashes, but it gets automatically restarted (note how the `self()` code returned 2 different pids).

The **Receiving Exit Signals** section of the **processes reference manual** gives some more information:



Die größte Investition



Ad danach war die Absetzung ...

lov-e-is-rare

Mehr

The default behaviour when a process receives an exit signal with an exit reason other than normal, is to terminate and in turn emit exit signals with the same exit reason to its linked processes. An exit signal with reason normal is ignored.

A process can be set to trap exit signals by calling:

```
process_flag(trap_exit, true)
```

When a process is trapping exits, it will not terminate when an exit signal is received. Instead, the signal is transformed into a message `{EXIT, FromPid, Reason}` which is put into the mailbox of the process just like a regular message.

Let's now try the same thing, but capturing the exit signal with `process_flag/2`

:

```
1  1> process_flag(trap_exit, true).
```

```
2 false
3 2> self().
4 <0.31.0>
5 3> mymodule:start_link().
6 {ok,<0.35.0>}
7 4> mymodule:crash().
8 ok
9 5>
10 =ERROR REPORT==== 30-Dec-2012::15:51:20 ===
11 ** Generic server mymodule terminating
12 ** Last message in was {'$gen_cast',crash}
13 ** When Server state == []
14 ** Reason for termination ==
15 ** error
16
17 5> self().
18 <0.31.0>
19 6> receive X->X end.
20 {'EXIT',<0.35.0>,error}
```

log.txt hosted with ❤ by [GitHub](#)

[view raw](#)


As you can see, the shell didn't died, but got a message instead. Cool!

To sum up, and to quote the [Processes](#) entry in the OTP reference manual:

- *Two processes can be linked to each other. A link between two processes `Pid1` and `Pid2` is created by `Pid1` calling the BIF `link(Pid2)` (or vice versa).*
- *Links are **bidirectional** and there can only be one link between two processes. Repeated calls to `link(Pid)` have no effect.*
- *The default behaviour when a process receives an exit signal with an exit reason other than normal, is to terminate and in turn emit exit signals with the same exit reason to its linked processes.*
- *An exit signal with reason normal is ignored.*

As a final note to links, there are a couple of interesting functions to know about:

- `unlink/1`: Removes the link between 2 processes.
- `spawn_link/1`: Spawns a new process, linked.

 **Power Your Search & Big Data Analytics Applications**
elastic Consulting • Implementation • Support

SEARCH TECHNOLOGIES
Part of **ACCENTURE**

START HERE

Monitoring a process

Monitors are not links, they are a more relaxed way of knowing what happened to a process.

They use messages instead of signals, and these messages are not propagated like signals, so nothing happens to your process when a monitored process exits (except that you get a new message in your mailbox). Also, they are unidirectional and allow you to establish as many "monitors" as you want (remember how links limited the number of links between 2 processes to just 1). Quoting the **manual**:

An alternative to links are monitors. A process `Pid1` can create a monitor for `Pid2` by calling the BIF `erlang:monitor(process, Pid2)`. The function returns a reference `Ref`.

If `Pid2` terminates with exit reason `Reason`, a 'DOWN' message is sent to `Pid1`:

`{'DOWN', Ref, process, Pid2, Reason}`

Monitors are unidirectional. Repeated calls to `erlang:monitor(process, Pid)` will create several, independent monitors and each one will send a 'DOWN' message when `Pid` terminates.

Now, to the very same source code we tried above, let's modify it to add a `start/0` function:

```
1 -module(mymodule).  
2 start() ->  
3     gen_server:start({local, ?MODULE}, ?MODULE, [], []).  
4 ...  
5 start_link() ->
```

```
6     gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).
7     ...
8     crash() ->
9         gen_server:cast(?MODULE, crash).
10    ...
11    handle_cast(crash, State) ->
12        {stop, error, State};
13    ...
```

[mymodule.erl](#) hosted with ♥ by [GitHub](#)

[view raw](#)

Note the new start/0 call that uses `gen_server:start/4` instead of `gen_server:start_link/4`, so we can avoid having our shell linked to the new process when trying it.

Let's try it in the shell:

```
1  1> self().
2  <0.31.0>
3  2> mymodule:start().
4  {ok,<0.34.0>}
5  3> process_flag(trap_exit, true).
6  false
7  4> erlang:monitor(process, mymodule).
8  #Ref<0.0.0.43>
9  5> mymodule:crash().
10 ok
11 =ERROR REPORT==== 30-Dec-2012::16:21:29 ===
12 ** Generic server mymodule terminating
13 ** Last message in was {'$gen_cast',crash}
14 ** When Server state == []
15 ** Reason for termination ==
16 ** error
17
```

```
18 6> receive X->X end.  
19 {'DOWN', #Ref<0.0.0.43>, process, {mymodule, nonode@nohost}, error}
```

log.txt hosted with ❤ by [GitHub](#)

[view raw](#)

So our shell, while still was informed about the exit of the other process, didn't die (because it didn't got an exit signal).

A couple of interesting functions related to monitoring:

- [spawn_monitor/1](#): Spawns a new process, monitored.
- [demonitor/1](#): Removes a monitoring.

To link or to monitor a process in your erlang code: That is the question

So this brings up the question: should I link to or monitor my processes? Of course the answer is 42.. I mean, it depends. Use link if you:

- Have a dependency on a process (i.e: you can't run if a specific process dies). This fits great into supervisor trees.
- Have a bidirectional dependency, where a parent can't run if the child dies, and you also want to kill the child if the parent dies in turn.
- Only need 1 link between processes (remember that if A and B are linked, all subsequent calls to link/2 will be ignored).
- You are a supervisor, or you want some kind of physical relationship between your processes in your architecture (i.e: you actually need to die or restart or try something out to fix the situation that led to the death of your child).

Use monitor if you:

- Just want to know if a specific process is running, but it's ok for you to continue execution without it (i.e: you can just send an alarm and keep going).
- Don't need a bidirectional relation (i.e: you want A to know about B dying but you don't need B to know about A).
- You are an external process, just interested in the faith of a particular process

[Read Other Articles](#)

Email: marcelog@gmail.com