

Erlang's let-it-crash philosophy - applicable elsewhere? [closed]

Asked 14 years ago Modified 12 days ago Viewed 17k times



76



Closed. This question is [opinion-based](#). It is not currently accepting answers.



Want to improve this question? Update the question so it can be answered with facts and citations by [editing this post](#).

Closed 9 days ago.

The community is reviewing whether to reopen this question as of 9 days ago.

[Improve this question](#)

Erlang's (or Joe Armstrong's?) advice ***NOT to use defensive programming*** and to let processes crash (rather than pollute your code with needless guards trying to keep track of the wreckage) makes so much sense to me now that I wonder why I wasted so much effort on error handling over the years!

What I wonder is - is this approach only applicable to platforms like Erlang? Erlang has a VM with simple native support for process supervision trees and restarting processes is *really* fast. Should I spend my development

efforts (when not in the Erlang world) on recreating supervision trees rather than bogging myself down with top-level exception handlers, error codes, null results etc etc etc.

Do you think this change of approach would work well in (say) the .NET or Java space?

java

.net

erlang

defensive-programming

Share

edited Dec 8, 2010 at 23:39

Improve this question

Follow

asked Dec 8, 2010 at 22:57



Andrew Matthews

3,146 ● 2 ● 30 ● 47

-
- 1 I wrote this some time ago:
mazenharake.wordpress.com/2009/09/14/let-it-crash-the-right-way maybe you can find something useful there as well.
– [Mazen Harake](#) Dec 9, 2010 at 21:28

Thanks Mazen. That's a good post! I get the point of the philosophy you're describing - what I wonder is whether the threading, process or appdomains of .NET (say) are up to the task of restarting as a form of control construct...?

– [Andrew Matthews](#) Dec 10, 2010 at 4:01

I think this can be applied everywhere. I'm however out on thin ice here because I can't prove it :) So for me this is just a feeling or a guess, I haven't tried it in another language to know :) – [Mazen Harake](#) Feb 16, 2011 at 23:25

To whoever saw fit to close this question - WTH? Why? Why is "opinion" a bad thing in the context of architecture and design? In areas of design, there is seldom one true answer to a question. The point of this is to elicit as many viewpoints as possible. Software engineers who are not familiar with the technique will find a broad range of perspectives on the idea, and inspiration on how to apply it in their own situation. If you confine the conversation to purely factual answers, you will undermine the value of this platform. – [Andrew Matthews](#)

Dec 13 at 5:40

9 Answers

Sorted by:

Highest score (default)



43



[It's applicable everywhere](#). Whether or not you write your software in a "let it crash" pattern, it will crash anyway, e.g., when hardware fails. "Let it crash" applies anywhere where you need to withstand reality. Quoth James Hamilton:



If a hardware failure requires any immediate administrative action, the service simply won't scale cost-effectively and reliably. The entire service must be capable of surviving failure without human administrative interaction. Failure recovery must be a very simple path and that path must be tested frequently. Armando Fox of Stanford has argued that the best way to test the failure path is never to shut the service down normally. Just hard-fail it. This sounds counter-

intuitive, but if the failure paths aren't frequently used, they won't work when needed.

This doesn't precisely mean "never use guards," though. But don't be afraid to crash!

Share Improve this answer

edited May 16, 2012 at 13:28

Follow

answered Dec 8, 2010 at 23:01



Craig Stuntz

127k ● 12 ● 256 ● 275

BUT: is it cheap enough to use hard failure as a control construct in VMs other than Erlang's? – [Andrew Matthews](#)

Dec 8, 2010 at 23:34

2 In mission-critical systems, a common solution is to have a "watchdog" process which monitors the primary application. The primary application is designed to fail-fast (thus avoiding problems re corruption of program state), and the watchdog can restart it fresh (or fail-over to another system if using a hot-backup design). – [Stephen Cleary](#) Dec 8, 2010 at 23:34

3 @Andrew: I would say yes. I've used fail-fast on .NET and native Win32 code (my background is production-critical automation programming). Microsoft's Windows Error Reporting system is designed for fail-fast applications. – [Stephen Cleary](#) Dec 8, 2010 at 23:37

1 @Craig Stuntz - Very interesting read - and very pertinent to my situation. Thanks! – [Andrew Matthews](#) Dec 9, 2010 at 0:07

- 2 @Andrew: Not in the BCL, but the [Windows API Code Pack](#) includes

`ApplicationRestartRecoveryManager.RegisterForApplicationRestart`, which restarts your app automatically if it ends up in Windows Error Reporting (only available on Vista and later). – [Stephen Cleary](#) Dec 9, 2010 at 14:46



28



Yes, it is applicable everywhere, but it is important to note in which context it is meant to be used. It does **not** mean that the application as a whole crashes which, as @PeterM pointed out, can be catastrophic in many cases. The goal is to build a system which as a whole never crashes but can handle errors internally. In our case it was telecomms systems which are expected to have downtimes in the order of minutes per year.

The basic design is to layer the system and isolate central parts of the system to monitor and control the other parts which do the work. In OTP terminology we have *supervisor* and *worker* processes. Supervisors have the job of monitoring the workers, and other supervisors, with the goal of restarting them in the correct way when they crash while the workers do all the actual work. Structuring the system properly in layers using this principle of strictly separating the functionality allows you to isolate most of the error handling out of the workers into the supervisors. You try to end up with a **small** fail-safe error kernel, which if correct can handle errors anywhere in the rest of the system. It is in this context where the "let-it-crash" philosophy is meant to be used.

You get the paradox of where you are thinking about errors and failures everywhere with the goal of actually handling them in as few places as possible.

The best approach to handle an error depends of course on the error and the system. Sometimes it is best to try and catch errors locally within a process and trying to handle them there, with the option of failing again if that doesn't work. If you have a number of worker processes cooperating then it is often best to crash them all and restart them again. It is a supervisor which does this.

You do need a language which generates errors/exceptions when something goes wrong so you can trap them or have them crash the process. Just ignoring error return values is not the same thing.

[Share](#) [Improve this answer](#)

[Follow](#)

[edited Jan 26, 2018 at 18:02](#)



[goncalotomas](#)

1,000 ● 1 ● 14 ● 29

[answered Dec 9, 2010 at 22:29](#)



[rvirding](#)

20.9k ● 2 ● 38 ● 57

I understand that throw and bomb-out is not what the philosophy is about. My question pertains to the performance implications of a properly implemented 'let it crash' approach in systems OTHER THAN Erlang. ;-) Erlang seems to be uniquely tuned/designed to exploit this philosophy, whereas .NET (e.g.) doesn't seem to be. I'm after counter examples/frameworks that can disprove this contention. Clearly targeted designs will always be needed to exploit 'fast fail'. If I take ages to load all the state, dependencies etc

before I can retry or resume, then it's not a viable option.

– [Andrew Matthews](#) Dec 10, 2010 at 4:13

@Andrew Matthews: There are (at least) two different problems here. If you want to use processes for error handling the same way as in Erlang then the concurrency should be lightweight as in Erlang so as to minimise the time when part of the system is not working. You also have the problem you mentioned of handling state, I would say this is a design issue but the design will most likely be language/system specific to exploit features of the language. For example in Erlang you could have a supervisor managing an ETS table so that when a worker crashes it will not have to be reloaded. – [rvirding](#) Dec 13, 2010 at 11:21

I see. And because the supervisor now handles a shared resource, *IT* must be supervised? In practice where do you cut off the infinite regress? I see whole new vistas of trade-offs to be negotiated. ;-) – [Andrew Matthews](#) Dec 14, 2010 at 2:15



6

It is called fail-fast. It's a good paradigm provided you have a team of people who can respond to the failure (and do so quickly).



In the NAVY all pipes and electrical is mounted on the exterior of a wall (preferably on the more public side of a wall). That way, if there is a leak or issue, it is more likely to be detected quickly. In the NAVY, people are punished for not responding to a failure, so it works very well: failures are detected quickly and acted upon quickly.



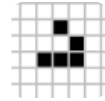
In a scenario where someone cannot act on a failure quickly, it becomes a matter of opinion whether it is more

beneficial to allow the failure to stop the system or to swallow the failure and attempt to continue onward.

Share Improve this answer

answered Dec 8, 2010 at 23:02

Follow



[Edwin Buck](#)

70.8k ● 7 ● 101 ● 142

5 The navy are experts at handling pipes apparently – [Falmarri](#) Dec 8, 2010 at 23:05

10 Genuine question: Why do people spell "Navy" as "NAVY" - Navy is not an acronym? – [L. Cornelius Dol](#) Dec 8, 2010 at 23:06

6 The capitalization makes me sing "In the Navy, in the Navy" in my head all the time...damn you. – [Jim Brissom](#) Dec 8, 2010 at 23:41

1 Automatic systems fail. To back them up the Navy uses men. Some things are more automatic than others, and to back those items up more similar systems are used. For ship electrical, the cost of a electrical distribution system (which typically is computer controlled) and redundant wiring would be prohibitive (and not necessarily more reliable unless you added personnel). While the government contractors might let costs overrun with ease, the actual Navy is quite a penny pincher. – [Edwin Buck](#) Dec 9, 2010 at 15:18

3 The all caps NAVY is common usage, but is probably incorrect. The Navy is heavily acronym driven NAVSTA (Naval Station), NAS (Naval Air Station), etc. I guess eventually people in the Navy become accustomed to the all caps acronyms and type NAVY. I have nothing more definitive to back this up, except my time in service. – [Edwin Buck](#) Dec 9, 2010 at 15:20



5



I write programs that rely on data from real world situations and if they crash they can cause big \$\$ in physical damage (not to mention big \$\$ in lost revenue). I would be out of a job in a flash if I did not program defensively.

With that said I think that Erlang must be a special case that not only can you restart things instantly, that a restarted program can pop up, look around and say "ahhh .. that was what I was doing!"

Share Improve this answer

answered Dec 8, 2010 at 23:03

Follow



Peter M


7,483 ● 3 ● 53 ● 94

2 @Andrew Mathews - Well if you flush the corrupted state, and the program restarts with the same inputs, aren't you setting yourself up for the same situation that caused the crash in the first place? Or is a crash considered a transient event, and hence not repeatable? – [Peter M](#) Dec 8, 2010 at 23:55

4 @Peter M : If your code is side-effect free and you feed it the same input then it will crash with the same error. Erlang supervisors have parameters that control how many times a failing process will be started in a given time period. If the process crashes outside of the supervisor's parameters then the supervisor will crash, and its supervisor will be notified. But this is still better than what you get in a mutable language. The difference in Erlang is that you can still have the process handle non-failing calls, and you can fix the bug and hot-load it without bringing the system down. – [Tim](#) Dec 9, 2010 at 18:21

4 @Peter M: The goal is to build a robust system which can handle errors and crashes internally but **never** as a whole go

down. In our case it was telecoms system which are expected to have downtimes in the order of minutes per year. The basic design is to layer the system in such way that each layer can handle errors in the next layer with the goal of having a small error kernel. This means that large parts of the system don't have to handle errors, which makes them safer and the system as a whole more robust. It is in this context which the "let-it-crash" philosophy is used. – [rvirding](#) Dec 9, 2010 at 21:43

- 1 @rvirding - WRT to layering: (without wishing to grumble) I've seen in more than one place that 'exception neutral' code (that may carefully manage atomicity in the face of exceptions) mutates over time in the hands of support engineers et al from exception-neutral -> catch-and-log-then-rethrow -> catch-and-log-then-swallow. 'Mature' code bases can end up with mishmashes of error handling techniques and most of those inadvertantly destroy the atomicity of the original design. Not sure how to re-assert the proper layering in the face of entropy. – [Andrew Matthews](#) Dec 10, 2010 at 3:55 
-

- 1 @Andrew Matthews: That is a known problem for which there is no real known solution. :-) It is alleviated if most programmers work in the crashable part of the system and only the select few in the error handling part, but it is not solved. – [rvirding](#) Dec 10, 2010 at 10:38
-



5

My colleagues and myself thought about the topic not especially technology wise but more from a domain perspective and with a safety focus.



The question is "Is it safe to let it crash?" or better "Is it even possible to apply a robustness paradigm like Erlang's "let it crash" to safety-related software projects?".



In order to find an answer we did a small research project using a close-to-reality scenario with industrial and especially medical background. Take a look here (http://bit.ly/Z-Blog_let-it-crash). There is even a paper for download. Tell me what you think!

Personally I think it is applicable in many cases and even desirable, especially when there is a lot of error handling to do (safety-related systems). You cannot always use Erlang (missing real time features, no real embedded support, costumer wishes ...), but I'm pretty sure you can implement it otherwise (e.g. using threads, exceptions, message passing). I haven't tried it yet though, but I'd like to.

Share Improve this answer

edited Sep 14, 2013 at 12:15

Follow

answered Sep 14, 2013 at 12:10



Christoph Woskowski

51 ● 1 ● 2

The link is broken. Can you fix it? – [Morten Toudahl](#) Apr 9, 2021 at 5:58

Archive for your blog post:

web.archive.org/web/20160308170353/http://blog.zuehlke.com/en/... – [billylanchantin](#) Jun 11, 2021 at 16:45



IMHO Some developers handle/wrap checked exceptions with code which add little value. It is often simpler to allow

3

a method to throw the original exception unless you are going to handle it and add some value.



Share Improve this answer

answered Dec 8, 2010 at 23:02



Follow



Peter Lawrey

533k ● 82 ● 767 ● 1.1k



- 1 It makes debugging substantially easier if you throw a descriptive exception instead of letting lower level details leak through your interface. But I agree that the trade-off is not always in favour of re-wrapping exceptions.

– Konrad Rudolph Dec 8, 2010 at 23:10

Totally agree. My experience is that I often find error handling which does nothing. My personal code catches only when I can do something about it. For example I might catch a timeout on a connection so I can tell the user to try again. But I will not catch a runtime coming from inside a third party API. I'll let it propagate and a top elvel generalised handler deal with it. – drekka Dec 8, 2010 at 23:18



Yes, even in economy, see this article:

[https://www.nytimes.com/2020/04/16/upshot/world-](https://www.nytimes.com/2020/04/16/upshot/world-economy-restructuring-coronavirus.html)

1

[economy-restructuring-coronavirus.html](https://www.nytimes.com/2020/04/16/upshot/world-economy-restructuring-coronavirus.html) . The World became a "spaghetti code" and is suffering a "Global State" issue.



Share Improve this answer

answered Apr 16, 2020 at 9:54



Follow



Henry H.

919 ● 8 ● 11

Thanks Henry, sounds really thought provoking.

– [Andrew Matthews](#) Aug 4, 2020 at 20:22



0



It's applicable everywhere. Conceptually, you want to design your software with many layers (like an onion) and have each layer responsible for progressively larger and larger areas of code. Each layer defining how it will recover from failures it can handle and then propagating failures it can't handle to the outer layer. The real mentality comes from designing the application from the beginning with recovery in mind. People who do a lot of defensive programming actually treat errors as after-thoughts, attempting to "handle" them at every place they think they will encounter them. In any complex application however, there will always be errors in places they didn't think they would encounter them. Instead, one should have the mindset that no matter what you do, you will get an error somewhere and thus you must figure out how to recover from any error, anywhere in the code.

Share Improve this answer

answered Dec 6 at 19:58

Follow



[Sarit Sotangkur](#)

1



0

It is an abstract idea of allowing your software to crash and therefore easily identify where and what the problem is rather than preventing the crash. Due to the abstract nature of this philosophy, its applicability is independent



of the technology it is to be applied to and fully dependent on the seriousness of the problems a crash may cause. If your software is an API that recognizes human voices and for some reason it fails to handle bariton voice and it is fully operational for non-bariton-voiced people while the crash for bariton-voiced people happens, then it makes sense, even though - of course - you want to hide technical details of the error from the user and display something readable, which will not leak any secrets by accident.

BUT if your software is starting slowly, initializing for half-a-day and it manages the cooling of a nuclear plant or manages a heart bypass, then crashes cannot be afforded and the programming needs to be very defensive.

Therefore, while evaluating whether this philosophy works for you, the question you need to ask yourself is this: "how much trouble a crash may cause?" and the feasibility of this philosophy largely depends on the answer to this question.

Share Improve this answer

Follow

answered Dec 8 at 11:44



[Lajos Arpad](#)

75.7k ● 40 ● 115 ● 215