FRED HEBERT

STUFF GOES BAD: ERLANG IN ANGER

次の皆様のサポート、レビュー、そして編集に感謝します。

Jacob Vorreuter、*Seth Falcon*、*Raoul Duke*、*Nathaniel Waisbrot*、*David Holland*、*Alisdair Sullivan*、*Lukas Larsson*、*Tim Chevalier*、*Paul Bone*、*Jonathan Roes*、*Roberto Aloi*、*Dmytro Lytovchenko*、*Tristan Sloughter*。

表紙の画像は sxc.hu に掲載されている drouu による fallout shelter を改変したものです。

v1.1.0 *2016-04-06*

目次

はじめに 1

第 I 部 Writing Applications 4

第 II 部 Diagnosing Applications 5

図目次

はじめに

ソフトウェアを実行するにあたって

他のプログラミング言語と比較して、Erlang には障害が起きた場合の対処方法がかなり独特な部分があります。他のプログラミング言語には、その言語自体や開発環境、開発手法といったものがエラーを防ぐためにできる限りのことをしてくれる、という共通の考え方があります。実行時に何かがおかしくなるということは予防する必要があるもので、予防できなかった場合には、人々が考えてきたあらゆる解決策の範囲を超えてしまいます。

プログラムは一度書かれると、本番環境に投入され、そこではあらゆることが発生するでしょう。エラーがあったら、新しいバージョンを投入する必要がでてきます。

一方で、Erlang では障害というものは、それが開発者によるもの、運用者によるもの、あるいはハードウェアによるもの、それらのどれであろうとも起きるものである、という考え方に沿っています。プログラムやシステム内のすべてのエラーを取り除くというのは非実用的かつ不可能に近いものです。[1]エラーをあらゆるコストを払って予防するのではなく、エラーにうまく対処できれば、プログラムのたいていの予期せぬ動作もその「なんとかする」手法でうまく対応できるでしょう。

これが「Let it Crash」[2]という考え方の元になっています。この考えを元にすると障害にうまく対処出来ること、かつシステム内のすべての複雑なバグが本番環境で発生する前に取り除くコストが極めて高いことから、プログラマーは対応方法がわかっているエラーだけ対処すべきで、それ以外は他のプロセス (やスーパーバイザー) や仮想マシンに任せるべきです。

たいていのバグが一時的なものであると仮定する[3]と、エラーに遭遇したときに単純にプロセスを再起動して安定して動いていた状態に戻すというのは、驚くほど良い戦略になりえます。

Erlang というのは人体の免疫システムと同様の手法が取られているプログラミング環

---

[1] 生命に関わるシステムは通常この議論の対象外です。

[2] Erlang 界隈の人々は、最近は不安がらせないようにということで「Let it Fail」のほうを好んで使うようです。

[3] Jim Gray の Why Do Computers Stop and What Can Be Done About It?によれば、132 個中 131 このバグが一時的なもの (非決定的で調査するときにはなくなっていて、再実行することで問題が解決するもの) です。

境です。一方で、他のたいていの言語は体内に病原菌が一切入らないようにするような衛生についてだけを考えています。どちらも私にとって極めて重要なプログラミング環境ものです。ほぼすべての環境でそれぞれに衛生状況が異なります。実行時のエラーがうまく対処されて、そのまま生き残れるような治癒の仕組みを持っているプログラミング環境はErlang の他にほとんどありません。

Erlang ではシステムになにか悪いことが起きてもすぐにはシステムが落ちないので、Erlang/OTP ではあなたが医者のようにシステムを診察する術も提供してくれます。システムの内部に入って、本番環境のその場でシステム内部を確認してまわって、実行中に内部をすべて注意深く観察して、ときには対話的に問題を直すことすらできるようになっています。このアナロジーを使い続けると、Erlang は、患者に診察所に来てもらったり、患者の日々の生活を止めることなく、問題を検出するための広範囲に及ぶ検査を実行したり、様々な種類の手術 (非常に侵襲性の高い手術でさえも) できるようにしてくれています。

本書は戦時において Erlang 衛生兵になるためのちょっとしたガイドになるよう書かれました。本書は障害の発生原因を理解する上で役立つ秘訣や裏ワザを集めた初めての書籍であり、また Erlang で作られた本番システムを開発者がデバッグするときに役立った様々なコードスニペットや実戦経験をあつめた辞書でもあります。

Who is this for?

This book is not for beginners. There is a gap left between most tutorials, books, training sessions, and actually being able to operate, diagnose, and debug running systems once they've made it to production. There's a fumbling phase implicit to a programmer's learning of a new language and environment where they just have to figure how to get out of the guidelines and step into the real world, with the community that goes with it.

This book assumes that the reader is proficient in basic Erlang and the OTP framework. Erlang/OTP features are explained as I see fit — usually when I consider them tricky — and it is expected that a reader who feels confused by usual Erlang/OTP material will have an idea of where to look for explanations if necessary[4].

What is not necessarily assumed is that the reader knows how to debug Erlang software, dive into an existing code base, diagnose issues, or has an idea of the best practices about deploying Erlang in a production environment[5].

[4] I do recommend visiting Learn You Some Erlang or the regular Erlang Documentation if a free resource is required

[5] Running Erlang in a screen or tmux session is *not* a deployment strategy.

How To Read This Book

This book is divided in two parts.

Part I focuses on how to write applications. It includes how to dive into a code base (Chapter **??**), general tips on writing open source Erlang software (Chapter **??**), and how to plan for overload in your system design (Chapter **??**).

Part **??** focuses on being an Erlang medic and concerns existing, living systems. It contains instructions on how to connect to a running node (Chapter **??**), and the basic runtime metrics available (Chapter **??**). It also explains how to perform a system autopsy using a crash dump (Chapter **??**), how to identify and fix memory leaks (Chapter **??**), and how to find runaway CPU usage (Chapter **??**). The final chapter contains instructions on how to trace Erlang function calls in production using `recon`[6] to understand issues before they bring the system down (Chapter **??**).

Each chapter is followed up by a few optional exercises in the form of questions or hands-on things to try if you feel like making sure you understood everything, or if you want to push things further.

[6] http://ferd.github.io/recon/ — a library used to make the text lighter, and with generally production-safe functions.

第I部

Writing Applications

第II部

Diagnosing Applications