Stuff Goes Bad:
Erlang in Anger

Fred Hébert 著、elixir.jp 訳

2018 年 6 月 20 日

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

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

[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。

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。

対象読者

本書は初心者向けではありません。たいていのチュートリアルや参考書、トレーニング講習などから実際に本番環境でシステムを走らせてそれを運用し、検査し、デバッグできるようになるまでには隔たりがあります。プログラマーが新しい言語や環境を学ぶ中で一般的なガイドラインから逸脱して、コミュニティの多くの人々が同様に取り組んでいる実世界の問題へと踏み出すまでには、明文化されていない手探りの期間が存在します。

本書は、読者は Erlang と OTP フレームワークの基礎には熟達していることを想定しています。Erlang/OTP の機能は—通常私がややこしいと思ったときには—私が適していると思うように説明しています。通常の Erlang/OTP の資料を読んで混乱してしまった読者には、必要に応じて何を参照すべきか説明があります。[4][5]

本書を読むにあたり前提知識として必ずしも想定していないものは、Erlang 製ソフトウェアのデバッグ方法、既存のコードベースの読み進め方、あるいは本番環境への Erlang 製プログラムのデプロイのベストプラクティス[6]などです。

本書の読み進め方

本書は二部構成です。

第 I 部ではアプリケーションの書き方に焦点を当てます。この部ではコードベースへの飛び込み方 (第 1 章)、オープンソースの Erlang 製ソフトウェアを書く上での一般的な秘訣 (第 2 章)、そし

[4] 無料の資料が必要であれば Learn You Some Erlang や通常の Erlang ドキュメントをおすすめします。
[5] 訳注:日本語資料としては、Learn you some Erlang for great good!日本語訳とその書籍版をおすすめします。
[6] Erlang を screen や tmux のセッションで実行する、というのはデプロイ戦略では**ありません**

てシステム設計における過負荷への計画の仕方 (第**??**章) を説明します。

第 II 部では Erlang 衛生兵になって、既存の動作しているシステムに取り組みます。この部では実行中のノードへの接続方法の解説 (第**??**章)、取得できる基本的な実行時のメトリクス (第**??**章) を説明します。またクラッシュダンプを使ったシステムの検死方法 (第**??**章)、メモリリークの検出方法と修正方法 (第**??**章)、そして暴走した CPU 使用率の検出方法 (第**??**章) を説明します。最終章では問題がシステムを落としてしまう前に理解するために、本番環境での Erlang の関数呼び出しを `recon`[7]を使ってトレースする方法を説明します。(第 3 章)

各章のあとにはすべてを理解したか確認したりより深く理解したい方向けに、いくつか補足的に質問やハンズオン形式の演習問題が付いてきます。

[7] http://ferd.github.io/recon/ ──本書を薄くするために使われるライブラリで、一般的に本番環境で使っても安心なものです

第I部

Writing Applications

第1章

コードベースへの飛び込み方

「ソースを読め」というフレーズは言われるともっとも煩わしい言葉ではありますが、Erlang プログラマとしてやっていくのであれば、しばしばそうしなければならないでしょう。ライブラリのドキュメントが不完全だったり、古かったり、あるいは単純にドキュメントが存在しなかったりします。また他の理由として、Erlang プログラマは Lisper に近しいところが少しあって、ライブラリを書くときには自身に起こっている問題を解決するために書いて、テストをしたり、他の状況で試したりということはあまりしない傾向にあります。そしてそういった別のコンテキストで発生する問題を直したり、拡張する場合は自分で行う必要があります。

したがって、仕事で引き継ぎがあった場合でも、自分のシステムと連携するために問題を修正したりあるいは中身を理解する場合でも、何も知らないコードベースに飛び込まなければならなくなることはまず間違いないでしょう。これは取り組んでいるプロジェクトが自分自身で設計したわけではない場合はいつでも、たいていの言語でも同様です。

世間にある Erlang のコードベースには主に 3 つの種類があります。1 つめは生の Erlang コードベース、2 つめは OTP アプリケーション、3 つめは OTP リリースです。この章ではこれら 3 つのそれぞれに見ていき、それぞれを読み込んでいくのに役立つ秘訣をお教えします。

1.1 生の Erlang

生の Erlang コードベースに遭遇したら、各自でなんとかしてください。こうしたコードはなにか特に標準に従っているわけでもないので、何が起きているかは自分で深い道に分け入っていかなければなりません。

つまり、`README.md` ファイルの類がアプリケーションのエントリーポイントを示してくれていて、さらにいえば、ライブラリ作者に質問するための連絡先情報などがあることを願うのみということです。

幸いにも、生の Erlang に遭遇することは滅多にありません。あったとしても、だいたいが初心者のプロジェクトか、あるいはかつて Erlang 初心者によって書かれた素晴らしいプロジェクトで

真剣に書き直しが必要になっているものです。一般的に、rebar3 やその前身 [1] のようなツールの出現によって、ほとんどの人が OTP アプリケーションを使うようになりました。

1.2 OTP アプリケーション

OTP アプリケーションを理解するのは通常かなり単純です。OTP アプリケーションはみな次のようなディレクトリ構造をしています。

```
doc/
ebin/
src/
test/
LICENSE.txt
README.md
rebar.config
```

わずかな違いはあるかもしれませんが、一般的な構造は同じです。

各 OTP アプリケーションは *app* **ファイル** を持っていて、ebin/<AppName>.app か、あるいはしばしば src/<AppName>.app.src という名前になっているはずです。[2]app ファイルには主に 2 つの種類があります。

```
{application, useragent, [
  {description, "Identify browsers & OSes from useragent strings"},
  {vsn, "0.1.2"},
  {registered, []},
  {applications, [kernel, stdlib]},
  {modules, [useragent]}
]}.
```

そして

```
{application, dispcount, [
  {description, "A dispatching library for resources and task "
                "limiting based on shared counters"},
```

[1] https://www.rebar3.org ―第 2 章で簡単に紹介されるビルドツールです。

[2] ビルドシステムが最終的に ebin にファイルを生成します。この場合、多くの src/<AppName>.app.src ファイルはモジュールを示すものではなく、ビルドシステムがモジュール化の面倒を見ることになります。

```
  {vsn, "1.0.0"},
  {applications, [kernel, stdlib]},
  {registered, []},
  {mod, {dispcount, []}},
  {modules, [dispcount, dispcount_serv, dispcount_sup,
             dispcount_supersup, dispcount_watcher, watchers_sup]}
]}.
```

の 2 種類です。

最初のケースは **ライブラリアプリケーション** と呼ばれていて、2 つめのケースは標準 **アプリケーション** と呼ばれています。

1.2.1 ライブラリアプリケーション

ライブラリアプリケーションは通常 *appname*_something というような名前のモジュールと、*appname* という名前のモジュールを持っています。これは通常ライブラリの中心となるインターフェースモジュールで、提供される大半の機能がそこに含まれています。

モジュールのソースを見ることで、少しの労力でモジュールがどのように動作するか理解できます。もしモジュールが特定のビヘイビア (gen_server や gen_fsm など) を何度も使っているようであれば、おそらくスーパーバイザーの下でプロセスを起動して、然るべき方法で呼び出すことが想定されているでしょう。ビヘイビアが一つもなければ、そこにあるのは関数のステートレスなライブラリです。この場合、モジュールのエクスポートされた関数を見ることで、このライブラリの目的を素早く理解できるでしょう。

1.2.2 標準アプリケーション

標準的な OTP アプリケーションでは、エントリーポイントとして機能する 2 つの潜在的なモジュールがあります。

1. *appname*
2. *appname*_app

最初のファイルはライブラリアプリケーションで見たものと似た使われ方 (エントリーポイント) をします。一方で、2 つめのファイルは application ビヘイビアを実装するもので、アプリケーションの階層構造の頂点を表すものになります。状況によっては最初のファイルは同時に両方の役割を果たします。

そのアプリケーションを単純にあなたのアプリケーションの依存先として追加しようとしているのであれば、*appname* の中を詳しく見てみましょううう。そのアプリケーションの運用や修正を行う必要があるのであれば、かわりに *appname*_app の中を見てみましょう。

アプリケーションはトップレベルのスーパーバイザーを起動して、その *pid* を返しますう。このトップレベルのスーパーバイザーはそれが自動で起動するすべての子プロセスの仕様を含んでいます。[3]

プロセスが監視ツリーのより上位にあれば、アプリケーションの存続にとってより致命的になってきます。またプロセスの重要性は起動開始の早さによっても予測可能です。(監視ツリー内の子プロセスはすべて順番に深さ優先で起動されています。) プロセスが監視ツリー内であとの方で起動されたとしたら、おそらくそれより前に起動されたプロセスに依存しているでしょう。

さらに、同じアプリケーション内で依存しあっているワーカープロセス (たとえば、ソケット通信をバッファしているプロセスと、その通信プロトコルを理解するための有限ステートマシンにそのデータをリレーするプロセス) は、おそらく同じスーパーバイザーの下で再グループ化されていて、何かおかしなことが起きたらまとめて落ちるでしょう。これは熟慮の末の選択で、通常どちらかのプロセスがいなくなったり状態がおかしくなってしまったときに、両方のプロセスを再起動してまっさらな状態から始めるほうが、どう回復するかを考えるよりも単純だからです。

スーパーバイザーの再起動戦略はスーパーバイザー以下のプロセス間での関係性に影響を与えます。

- `one_for_one` と `simple_one_for_one` は、失敗は全体としてアプリケーションの停止に関係してくるものの、お互いに直接依存しあっていないプロセスに使われます。[4]
- `rest_for_one` はお互いに直列に依存しているプロセスを表現するうときに使われます。
- `one_for_all` は全体がお互いに依存しあっているプロセスに使われます。

この構造の意味するところは、OTP アプリケーションを見るときは監視ツリーを上から順にたどるのが最も簡単であるということです。

監視された各ワーカープロセスでは、それが実装しているビヘイビアがそのプロセスの目的を知る上で良い手がかりとなります。

- `gen_server` はリソースを保持して、クライアント・サーバーパターン (より一般的にはリクエスト・レスポンスパターン) に沿っています。
- `gen_fsm` は有限ステートマシンなので一連のイベントやイベントに依存する入力と反応を扱います。プロトコルを実装するときによく使われます。
- `gen_event` はコールバック用のイベントのハブとして振る舞ったり、通知を扱う方法とし

[3] 場合によっては、そのスーパーバイザーが子プロセスをまったく指定しないこともあります。その場合、子プロセスはその API の関数あるいはアプリケーションの起動プロセス内で動的に起動される、あるいはそのスーパーバイザーが (アプリケーションファイルの `env` タプル内の)OTP の環境変数が読み込まれるのを許可するためだけに存在しているかのどちらかです。

[4] 開発者によっては `rest_for_one` がより適切な場面で `one_for_one` を使ったりします。起動順を正しく行うことを求めてそうするわけですが、先に言ったような再起動時や先に起動されたプロセスが死んだときの起動順については忘れてしまうのです。

て使われます。

これらのモジュールはすべてある種の構造を持っています。通常はユーザーに晒されたインターフェースを表すエクスポートされた関数、コールバックモジュール用のエクスポートされた関数、プライベート関数の順です。

監視関係や各ビヘイビアの典型的な役割を下地に、他のモジュールに使われているインターフェースや実装されたビヘイビアを見ることで、いま読み込んでいるプログラムに関するたくさんの情報が明らかになります。

1.2.3 依存関係

すべてのアプリケーションには依存するものが存在します。[5]そして、これらの依存先にはそれぞれの依存が存在します。OTP アプリケーションには通常状態を共有するものはありません。したがって、コードのある部分が他の部分にどのように依存しているかは、アプリケーションの開発者が正しく実装していると想定すれば、アプリケーションファイルをみるだけで知ることが出来ます。図 1.1 は、アプリケーションファイルを見ることで生成できるダイアグラムで、OTP アプリケーションの構造の理解に役立ちます。

こうした依存関係を使って各アプリケーションの短い解説を見ることで、何がどこにあるかの大まかな地図を描くのに役立つでしょう。似たダイヤグラムを生成するためには、`recon` の script ディレクトリ内のツールを使って `escript script/app_deps.erl` を実行してみましょう。[6]似たダイヤグラムが `observer`[7]アプリケーションを使うことで得られますが、各監視ツリーのものになります。これらをまとめることで、コードベースの中で何が何をしているかを簡単に見つけられるようになるでしょう。

[5] どんなに少なくとも `kernel` アプリケーションと `stdlib` アプリケーションに依存しています。
[6] このスクリプトは graphviz に依存しています。
[7] http://www.erlang.org/doc/apps/observer/observer_ug.html

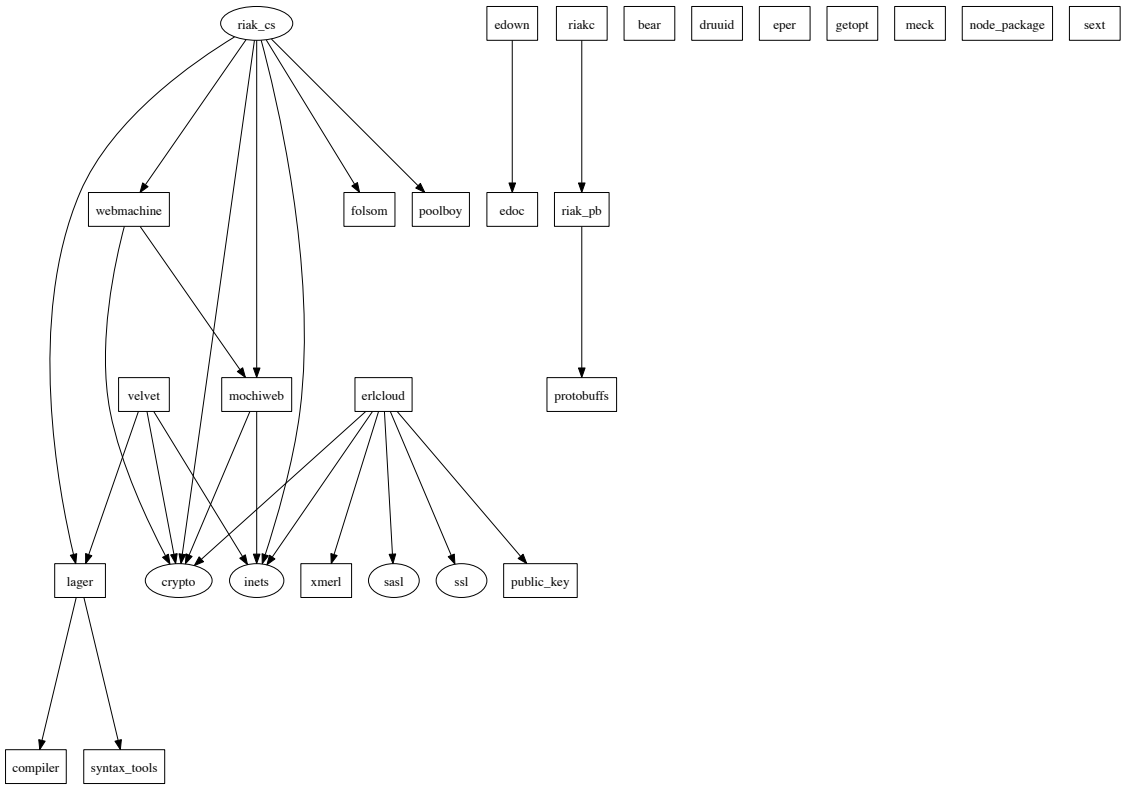

図 1.1　Basho のオープンソースクラウドライブラリである riak_cs の依存関係を表したグラフです。このグラフは kernel や stdlib といった必ず依存するようなものは除いています。楕円はアプリケーションで、四角はライブラリアプリケーションです。

1.3　OTP リリース

OTP リリースは世間で見かけるたいていの OTP アプリケーションよりもそれほど難しいものではありません。OTP リリースは複数の OTP アプリケーションを本番投入可能な状態でパッケージ化したもので、これによって手動でアプリケーションの `application:start/2` を呼び出す必要なく起動と停止行えるようになっています。コンパイルされたリリースは、デフォルトのものよりも含まれるライブラリの数は大小違いいますが自分専用の Erlang VM のコピーを持っていて、単独で起動できるようになっています。もちろん、リリースに関してはまだ話すことはありますが、一般的に OTP アプリケーションのときと同じようなやり方で中身を確認していきます。

OTP リリース内には通常、`relx.config` または `rebar.config` ファイル内の `relx` タプルがあります。ここに、どのトップレベルアプリケーションがリリースに含まれているかとパッケージ

化に関するオプションが書かれています。relx を使ったリリースはプロジェクトの Wiki ページ[8]や rebar3[9]のドキュメントサイトや erlang.mk[10]にあるドキュメントを読めば理解できます。

他のシステムは systools や reltool で使われる設定ファイルに依存しているでしょう。ここにリリースに含まれるすべてのアプリケーションが記述されていて、パッケージに関するオプションが少々[11]書かれています。それらを理解するには、既存のドキュメントを読むことをおすすめします。[12]

1.4 演習

復習問題

1. コードベースがアプリケーションがリリースかはどうやって確認できますか
2. ライブラリアプリケーションとアプリケーションはどの点が異なりますか
3. 監視において one_for_all 戦略で管理されるプロセスとはどういうプロセスですか
4. gen_server ビヘイビアではなく gen_fsm ビヘイビアを使うのはどういう状況ですか

ハンズオン

https://github.com/ferd/recon_demo のコードをダウンロードしてください。このコードは本書内の演習問題のテストベッドとして使われます。このコードベースにまだ詳しくないという前提で、この章で説明された秘訣や裏ワザを使ってこのコードベースを理解できるか見てみましょう。

1. このアプリケーションはライブラリですか。スタンドアロンシステムですか。
2. このアプリは何をしますか。
3. 依存するものはありますか。あるとすればなんですか。
4. このアプリケーションの README では非決定的である。これは真でしょうか。その理由も説明してください。
5. このアプリケーションの依存関係の連鎖を表現できますか。ダイヤグラムを生成してください。
6. README で説明されているメインアプリケーションにより多くのプロセスを追加できますか。

[8] https://github.com/erlware/relx/wiki
[9] https://www.rebar3.org/docs/releases
[10] http://erlang.mk/guide/relx.html
[11] 多数
[12] 訳注: 日本語訳版 https://www.ymotongpoo.com/works/lyse-ja/ja/24_release_is_the_word.html

第2章

Building Open Source Erlang Software

Most Erlang books tend to explain how to build Erlang/OTP applications, but few of them go very much in depth about how to integrate with the Erlang community doing Open Source work. Some of them even avoid the topic on purpose. This chapter dedicates itself to doing a quick tour of the state of affairs in Erlang.

OTP applications are the vast majority of the open source code people will encounter. In fact, many people who would need to build an OTP release would do so as one umbrella OTP application.

If what you're writing is a stand-alone piece of code that could be used by someone building a product, it's likely an OTP application. If what you're building is a product that stands on its own and should be deployed by users as-is (or with a little configuration), what you should be building is an OTP release.[1]

The main build tools supported are `rebar3` and `erlang.mk`. The former is a build tool and package manager trying to make it easy to develop and release Erlang libraries and systems in a repeatable manner, while the latter is a very fancy makefile that offers a bit less for production and releases but allows more flexibility. In this chapter, I'll mostly focus on using `rebar3` to build things, given it's the de-facto standard, is a tool I know well, and that `erlang.mk` applications tend to also be supported by `rebar3` as dependencies (the opposite is also true).

2.1 Project Structure

The structures of OTP applications and of OTP releases are different. An OTP application can be expected to have one top-level supervisor (if any) and possibly a bunch of dependencies

[1] The details of how to build an OTP application or release is left up to the Erlang introduction book you have at hand.

that sit below it. An OTP release will usually be composed of multiple OTP applications, which may or may not depend on each other. This will lead to two major ways to lay out applications.

2.1.1 OTP Applications

For OTP applications, the proper structure is pretty much the same as what was explained in 1.2:

```
_build/
doc/
src/
test/
LICENSE.txt
README.md
rebar.config
rebar.lock
```

What's new in this one is the `_build/` directory and the `rebar.lock` file, which will be generated automatically by `rebar3`[2].

This is the directory where `rebar3` places all build artifacts for a project, including local copies of libraries and packages required for it to work. No mainstream Erlang tool installs packages globally[3], preferring to instead keep everything project-local to avoid inter-project conflicts.

Such dependencies can be specified for `rebar3` by adding a few config lines to `rebar.config`:

```
{deps, [
  %% Hex.pm Packages
  myapp,
  {myapp,"1.0.0"},
  %% source dependencies
```

[2] Some people package `rebar3` directly in their application. This was initially done to help people who had never used rebar3 or its predecessors use libraries and projects in a boostrapped manner. Feel free to install rebar3 globally on your system, or keep a local copy if you require a specific version to build your system.

[3] Except as a local cache of unbuilt packages.

```
  {myapp, {git, "git://github.com/user/myapp.git", {ref, "aef728"}}},
  {myapp, {git, "https://github.com/user/myapp.git", {branch, "master"}}},
  {myapp, {hg, "https://othersite.com/user/myapp", {tag, "3.0.0"}}}
 ]}.
```

Dependencies are fetched directly from a `git` (or `hg`) source or as a package from `hex.pm` in a level-order traversal. They can then be compiled, and specific compile options can be added with the `{erl_opts, List}.` option in the config file[4].

You can call `rebar3 compile`, which will download all dependencies, and then build them and your app at once.

When making your application's source code public to the world, distribute it *without* the `_build/` directory. It's quite possible that other developers' applications depend on the same applications yours do, and it's no use shipping them all multiple times. The build system in place (in this case, `rebar3`) should be able to figure out duplicated entries and fetch everything necessary only once.

2.1.2 OTP Releases

For releases, the structure can a bit different. Releases are collections of applications, and their structures may reflect that.

Instead of having a top-level app alone in `src`, applications can be nested one level deeper in a `apps` or `lib` directory:

```
_build/
apps/
  - myapp1/
     - src/
  - myapp2/
     - src/
doc/
LICENSE.txt
README.md
rebar.config
rebar.lock
```

[4] More details on https://www.rebar3.org/docs/configuration

This structure lends itself to generating releases where multiple OTP applications under your control under a single code repository. Both `rebar3` and `erlang.mk` rely on the `relx` library to assemble releases. Other tools such as Systool and Reltool have been covered before[5], and can allow the user plenty of power if they do not like what they would get otherwise.

A `relx` configuration tuple (within `rebar.config`) for the directory structure above would look like:

```
{relx, [
  {release, {demo, "1.0.0"},
    [myapp1, myapp2, ..., recon]},

  {include_erts, false} % will use local Erlang install
]}
```

Calling `rebar3 release` will build a release, to be found in the `_build/default/rel/` directory. Calling `rebar3 tar` will generate a tarball at `_build/default/rel/demo/demo-1.0.0.tar.gz`, ready to be deployed.

2.2 Supervisors and start_link Semantics

In complex production systems, most faults and errors are transient, and retrying an operation is a good way to do things — Jim Gray's paper[6] quotes *Mean Times Between Failures* (MTBF) of systems handling transient bugs being better by a factor of 4 when doing this. Still, supervisors aren't just about restarting.

One very important part of Erlang supervisors and their supervision trees is that *their start phases are synchronous*. Each OTP process has the potential to prevent its siblings and cousins from booting. If the process dies, it's retried again, and again, until it works, or fails too often.

That's where people make a very common mistake. There isn't a backoff or cooldown period before a supervisor restarts a crashed child. When a network-based application tries to set up a connection during its initialization phase and the remote service is down, the application fails to boot after too many fruitless restarts. Then the system may shut down.

Many Erlang developers end up arguing in favor of a supervisor that has a cooldown period.

[5] http://learnyousomeerlang.com/release-is-the-word
[6] http://mononcqc.tumblr.com/post/35165909365/why-do-computers-stop

I strongly oppose the sentiment for one simple reason: *it's all about the guarantees.*

2.2.1 It's About the Guarantees

Restarting a process is about bringing it back to a stable, known state. From there, things can be retried. When the initialization isn't stable, supervision is worth very little. An initialized process should be stable no matter what happens. That way, when its siblings and cousins get started later on, they can be booted fully knowing that the rest of the system that came up before them is healthy.

If you don't provide that stable state, or if you were to start the entire system asynchronously, you would get very little benefit from this structure that a `try ... catch` in a loop wouldn't provide.

Supervised processes *provide guarantees* in their initialization phase, *not a best effort.* This means that when you're writing a client for a database or service, you shouldn't need a connection to be established as part of the initialization phase unless you're ready to say it will always be available no matter what happens.

You could force a connection during initialization if you know the database is on the same host and should be booted before your Erlang system, for example. Then a restart should work. In case of something incomprehensible and unexpected that breaks these guarantees, the node will end up crashing, which is desirable: a pre-condition to starting your system hasn't been met. It's a system-wide assertion that failed.

If, on the other hand, your database is on a remote host, you should expect the connection to fail. It's just a reality of distributed systems that things go down.[7] In this case, the only guarantee you can make in the client process is that your client will be able to handle requests, but not that it will communicate to the database. It could return `{error, not_connected}` on all calls during a net split, for example.

The reconnection to the database can then be done using whatever cooldown or backoff strategy you believe is optimal, without impacting the stability of the system. It can be attempted in the initialization phase as an optimization, but the process should be able to reconnect later on if anything ever disconnects.

If you expect failure to happen on an external service, do not make its presence a guarantee of your system. We're dealing with the real world here, and failure of external dependencies is always an option.

[7] Or latency shoots up enough that it is impossible to tell the difference from failure.

2.2.2 Side Effects

Of course, the libraries and processes that call such a client will then error out if they don't expect to work without a database. That's an entirely different issue in a different problem space, one that depends on your business rules and what you can or can't do to a client, but one that is possible to work around. For example, consider a client for a service that stores operational metrics — the code that calls that client could very well ignore the errors without adverse effects to the system as a whole.

The difference in both initialization and supervision approaches is that the client's callers make the decision about how much failure they can tolerate, not the client itself. That's a very important distinction when it comes to designing fault-tolerant systems. Yes, supervisors are about restarts, but they should be about restarts to a stable known state.

2.2.3 Example: Initializing without guaranteeing connections

The following code attempts to guarantee a connection as part of the process' state:

```
1  init(Args) ->
2      Opts = parse_args(Args),
3      {ok, Port} = connect(Opts),
4      {ok, #state{sock=Port, opts=Opts}}.
5
6  [...]
7
8  handle_info(reconnect, S = #state{sock=undefined, opts=Opts}) ->
9      %% try reconnecting in a loop
10     case connect(Opts) of
11         {ok, New} -> {noreply, S#state{sock=New}};
12          _ -> self() ! reconnect, {noreply, S}
13     end;
```

Instead, consider rewriting it as:

```
1  init(Args) ->
2      Opts = parse_args(Args),
```

```
3       %% you could try connecting here anyway, for a best
4       %% effort thing, but be ready to not have a connection.
5       self() ! reconnect,
6       {ok, #state{sock=undefined, opts=Opts}}.
7
8   [...]
9
10  handle_info(reconnect, S = #state{sock=undefined, opts=Opts}) ->
11      %% try reconnecting in a loop
12      case connect(Opts) of
13          {ok, New} -> {noreply, S#state{sock=New}};
14          _ -> self() ! reconnect, {noreply, S}
15      end;
```

You now allow initializations with fewer guarantees: they went from *the connection is available* to *the connection manager is available.*

2.2.4 In a nutshell

Production systems I have worked with have been a mix of both approaches.

Things like configuration files, access to the file system (say for logging purposes), local resources that can be depended on (opening UDP ports for logs), restoring a stable state from disk or network, and so on, are things I'll put into requirements of a supervisor and may decide to synchronously load no matter how long it takes (some applications may just end up having over 10 minute boot times in rare cases, but that's okay because we're possibly syncing gigabytes that we *need* to work with as a base state if we don't want to serve incorrect information.)

On the other hand, code that depends on non-local databases and external services will adopt partial startups with quicker supervision tree booting because if the failure is expected to happen often during regular operations, then there's no difference between now and later. You have to handle it the same, and for these parts of the system, far less strict guarantees are often the better solution.

2.2.5 Application Strategies

No matter what, a sequence of failures is not a death sentence for the node. Once a system has been divided into various OTP applications, it becomes possible to choose which applications are vital or not to the node. Each OTP application can be started in 3 ways: temporary, transient, permanent, either by doing it manually in `application:start(Name, Type)`, or in the config file for your release:

- `permanent`: if the app terminates, the entire system is taken down, excluding manual termination of the app with `application:stop/1`.
- `transient`: if the app terminates for reason `normal`, that's ok. Any other reason for termination shuts down the entire system.
- `temporary`: the application is allowed to stop for any reason. It will be reported, but nothing bad will happen.

It is also possible to start an application as an *included application*, which starts it under your own OTP supervisor with its own strategy to restart it.

2.3 Exercises

Review Questions

1. Are Erlang supervision trees started depth-first? breadth-first? Synchronously or asynchronously?
2. What are the three application strategies? What do they do?
3. What are the main differences between the directory structure of an app and a release?
4. When should you use a release?
5. Give two examples of the type of state that can go in a process' init function, and two examples of the type of state that shouldn't go in a process' init function

Hands-On

Using the code at https://github.com/ferd/recon_demo:

1. Extract the main application hosted in the release to make it independent, and includable in other projects.
2. Host the application somewhere (Github, Bitbucket, local server), and build a release

with that application as a dependency.

3. The main application's workers (`council_member`) starts a server and connects to it in its `init/1` function. Can you make this connection happen outside of the init function's? Is there a benefit to doing so in this specific case?

第II部

Diagnosing Applications

第3章

トレース

Erlang と BEAM VM の機能で、およそどれぐらいのことをトレースできるかはあまり知られておらず、また全然使われていません。

使えるところが限られているので、デバッガのことは忘れてください。[1] Erlang では開発中あるいは稼働中の本番システムの診断であっても、システムのライフサイクルのすべての場所において、トレースは有効です。

トレースを行ういくつかの Erlang プログラムがあります。

- `sys`[2] は OTP に標準で付属されており、利用者はカスタマイズしたトレース機能や、あらゆる種類のイベントのロギングなどができます。多くの場合、開発用として完全かつ最適です。一方で、IO をリモートシェルにリダイレクトしないですし、メッセージのトレースのレート制限機能を持たないため、本番環境にはあまり向きません。このモジュールのドキュメントを読むことをお勧めします。
- `dbg`[3]も Erlang/OTP に標準で付属しています。使い勝手の面ではインターフェースは少しイケてませんが、必要なことをやるには充分です。問題点としては、**何をやっているのか知らないといけない**ということです。なぜなら `dbg` はノードのすべてをロギングすることや、2 秒もかからずにノードを落とすこともできるからです。
- **トレース** *BIF* は `erlang` モジュールの一部として提供されています。このリストの全てのアプリケーションで使われているローレベルの部品ですが、抽象化が低いため、利用するのは困難です。

---

[1] デバッガでブレークポイントを追加してステップ実行する時の代表的な問題は、多くの Erlang プログラムとうまくやりとりができないことです。あるプロセスがブレークポイントで止まっても、その他のプロセスは動作し続けます。そのため、プロセスがデバッグ対象のプロセスとやりとりが必要なときにはすぐに、プロセス呼び出しがタイムアウトしてクラッシュし、おそらくノード全体を落としてしまいます。ですから、デバックは非常に限定的なものとなります。一方でトレースはプログラムの実行を邪魔することは無く、また必要なデータをすべて取得することができます。

[2] http://www.erlang.org/doc/man/sys.html

[3] http://www.erlang.org/doc/man/dbg.html

- `redbug`[4] は `eper`[5] スイートの一部で、本番環境環境でも安全に使えるトレースライブラリです。内部にレート制限機能を持ち、使いやすい素敵なインターフェースを持っていますが、利用するには `eper` が依存するもの全てを追加する必要があります。ツールキットは包括的で、またインストールは非常に面白いです。
- `recon_trace`[6]は `recon` によるトレースです。`redbug` と同程度の安全性を目的としていましたが、依存関係はありません。インターフェースは異なり、またレート制限のオプションも完全に同じではありません。関数呼び出しもトレースすることができますが、メッセージのトレースはできません[7]。

この章では `recon_trace` によるトレースにフォーカスしていきますが、使われている用語やコンセプトの多くは、Erlang の他のトレースツールにも活用できます。

3.1 トレースの原則

Erlang のトレース BIF は全ての Erlang コードをトレースすることを可能にします[8]。BIF は *pid* **指定**と**トレースパターン**に分かれています。

pid 指定により、ユーザはどのプロセスをターゲットにするかを決めることができます。pid は、特定の pid, **全ての** pid, **既存の** pid, あるいは `newpid`（関数呼び出しの時点ではまだ生成されていないプロセス）で指定できます。

トレースパターンは機能の代わりになります。機能の指定は２つに分かれており、MFA(モジュール、関数、アリティ）と Erlang のマッチの仕様で引数に制約を加えています[9]

特定の関数呼び出しがトレースされるかどうかを定義している箇所は、3.1 にあるように、両者の共通部分です。

pid 指定がプロセスを除外、あるいはトレースパターンが指定の呼び出しを除外した場合、トレースは受信されません。

`dbg`（およびトレース BIF）のようなツールは、このベン図を念頭に置いて作業することを前提としています。pid 指定およびトレースパターンを別々に指定し、その結果が何であろうとも、両者の共通部分が表示されることになります。

---

[4] https://github.com/massemanet/eper/blob/master/doc/redbug.txt
[5] https://github.com/massemanet/eper
[6] http://ferd.github.io/recon/recon_trace.html
[7] メッセージのトレース機能は将来のバージョンでサポートされるかもしれません。ライブラリの著者は OTP を使っている時には必要性を感じておらず、またビヘイビアと特定の引数へのマッチングにより、ユーザはおよそ同じことを実現できます
[8] プロセスに機密情報が含まれている場合、`process_flag(sensitive, true)` を呼ぶことで、データを非公開にすることを強制できます
[9] http://www.erlang.org/doc/apps/erts/match_spec.html

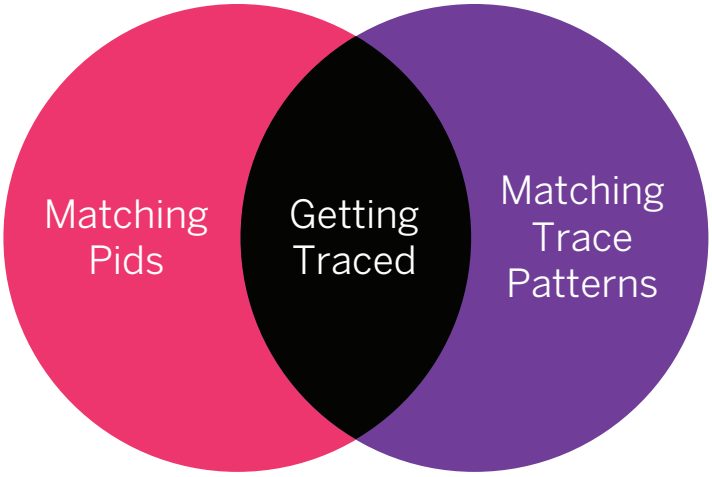

図 3.1　トレースされるのは、pid 指定とトレースパターンの交差した箇所です

一方で `redbug` や `recon_trace` のようなツールでは、これらを抽象化しています。

3.2　Recon によるトレース

デフォルトでは Recon は全てのプロセスにマッチしますが、デバッグ時のほとんどのケースはこれで問題ありません。多くの場合、あなたが遊びたいと思う面白い部分は、トレースするパターンの指定です。Recon ではいくつかの方法をサポートしています。

最も基本的な指定方法は `{Mod, Fun, Arity}`で、`Mod` はモジュール名、`Fun` は関数名、`Arity` はアリティつまりトレース対象の関数の引数の数です。いずれもワイルドカードの（`'_'`）で置き換えることができます。本番環境の実行は明らかに危険なため、Recon は（`{'_','_','_'}`のように) あまりにも広範囲、あるいは全てにマッチするような指定は禁止しています。

より賢明な方法は、アリティを引数のリストにマッチする関数で置き換えることです。その関数は ETS で利用できるもの[10]と同様に、マッチの指定で利用されるものに限定されています。また、複数のパターンをリストで指定して、マッチするパターンを増やすこともできます。

レート制限は２つの方法、静的な値によるカウントもしくは一定期間内にマッチした数、で行うことができます。

より詳細には立ち入らず、ここではいくつかの例と、どうトレースするのかを見ていきます。

```
%% queue モジュールからの全ての呼び出しを、最大で 10 回まで出力
recon_trace:calls({queue, '_', '_'}, 10)
```

[10] http://www.erlang.org/doc/man/ets.html#fun2ms-1

```
%% lists:seq(A,B) の全ての呼び出しを、最大で 100 回まで出力
recon_trace:calls({lists, seq, 2}, 100)

%% lists:seq(A,B) の全ての呼び出しを、最大で 1 秒あたり 100 回まで出力
recon_trace:calls({lists, seq, 2}, {100, 1000})

%% lists:seq(A,B,2) の全ての呼び出し（2 つずつ増えていきます）を、最大で 100 回まで出力
recon_trace:calls({lists, seq, fun([_,_,2]) -> ok end}, 100)

%% 引数としてバイナリを指定して呼び出された iolist_to_binary/1 への全ての呼び出し
%% （意味のない変換をトラッキングしている一例）
recon_trace:calls({erlang, iolist_to_binary,
                   fun([X]) when is_binary(X) -> ok end},
                  10)

%% 指定 Pid から queue モジュールの呼び出しを、最大で 1 秒あたり 50 回まで
recon_trace:calls({queue, '_', '_'}, {50,1000}, [{pid, Pid}])

%% リテラル引数のかわりに、関数のアリティでトレースを出力
recon_trace:calls(TSpec, Max, [{args, arity}])

%% dict と lists モジュールの filter/2 関数にマッチして、かつ new プロセスからの呼び出しのみ
recon_trace:calls([{dict,filter,2},{lists,filter,2}], 10, [{pid, new}])

%% 指定モジュールの handle_call/3 関数の、new プロセスおよび
%% gproc で登録済の既存プロセスからの呼び出しをトレース
recon_trace:calls({Mod,handle_call,3}, {1,100}, [{pid, [{via, gproc, Name}, new]}

%% 指定の関数呼び出しの結果を表示します。重要なポイントは、
%% return_trace() の呼び出しもしくは {return_trace} へのマッチです
recon_trace:calls({Mod,Fun,fun(_) -> return_trace() end}, Max, Opts)
recon_trace:calls({Mod,Fun,[{'_', [], [{return_trace}]}]}, Max, Opts)
```

各呼び出しはそれ以前の呼び出しを上書きし、また全ての呼び出しは recon_trace:clear/0 でキャンセルすることができます。

組み合わせることが可能なオプションはもう少しあります。

{pid, PidSpec}
: トレースするプロセスの指定です。有効なオプションは all, new, existing, あるいはプロセスディスクリプタ ({A,B,C}, "<A.B.C>", 名前をあらわすアトム、{global, Name},

`{via, Registrar, Name}`, あるいは pid) のどれかです。リストにすることで、複数指定することも可能です。

`{timestamp, formatter | trace}`

デフォルトでは formatter プロセスは受信したメッセージにタイムスタンプを追加します。正確なタイムスタンプが必要な場合、`{timestamp, trace}` オプションを追加することで、トレースするメッセージの中のタイムスタンプを使うことを強制できます。

`{args, arity | args}`

関数呼び出しでアリティを表示するか、(デフォルトの）リテラル表現を出力するか

`{scope, global | local}`

デフォルトでは 'global'(明示的な関数呼び出し) だけがトレースされ、内部的な呼び出しはトレースされません。ローカルの呼び出しのトレースを強制するには、`{scope, local}` を渡します。これは、`Module:Fun(Args)` ではなく `Fun(Args)` だけで呼び出される、プロセス内のコード変更をトラッキングしたいときに便利です。

特定の関数の特定の呼び出しやらをパターンマッチするこれらのオプションにより、開発環境・本番環境の多くの問題点をより早く診断できます。

「うーん、このおかしな挙動を引き起こしているのは何なのか、たぶんもっと多くのログを吐けばわかるかもしれない」という発想になったときには、通常はトレースすることが、デプロイや（ログを）読みやすいように変更しなくても必要なデータを入手することができる近道となります。

3.3 実行例

最初に、どこかのプロセスの queue:new 関数をトレースしてみましょう

```
1> recon_trace:calls({queue, new, '_'}, 1).
1
13:14:34.086078 <0.44.0> queue:new()
Recon tracer rate limit tripped.
```

最大 1 メッセージに制限されているため、recon が制限に達したことを知らせてくれます。

全ての `queue:in/2` 呼び出しを見て、queue に挿入される内容をみてみましょう。

```
2> recon_trace:calls({queue, in, 2}, 1).
1
13:14:55.365157 <0.44.0> queue:in(a, {[],[]})
Recon tracer rate limit tripped.
```

希望する内容を見るために、トレースパターンをリスト中の全引数にマッチする fun(_) を使うように変更して、return_trace() を返します。この最後の部分は、リターン値を含む各々の呼び出しのトレースそのものを生成します。

```
3> recon_trace:calls({queue, in, fun(_) -> return_trace() end}, 3).
1

13:15:27.655132 <0.44.0> queue:in(a, {[],[]})

13:15:27.655467 <0.44.0> queue:in/2 --> {[a],[]}

13:15:27.757921 <0.44.0> queue:in(a, {[],[]})
Recon tracer rate limit tripped.
```

引数リストのマッチは、より複雑な方法で行うことができます。

```
4> recon_trace:calls(
4>   {queue, '_',
4>    fun([A,_]) when is_list(A); is_integer(A) andalso A > 1 ->
4>        return_trace()
4>    end},
4>   {10,100}
4> ).
32

13:24:21.324309 <0.38.0> queue:in(3, {[],[]})

13:24:21.371473 <0.38.0> queue:in/2 --> {[3],[]}

13:25:14.694865 <0.53.0> queue:split(4, {[10,9,8,7],[1,2,3,4,5,6]})

13:25:14.695194 <0.53.0> queue:split/2 --> {{[4,3,2],[1]},{[10,9,8,7],[5,6]}}

5> recon_trace:clear().
ok
```

上記のパターンでは、特定の関数（'_'）にはマッチしていないことに注意してください。fun は２つの引数を持つ関数に限定され、また最初の引数はリストもしくは１よりも大きい数値です。

レート制限を緩めて非常に広範囲にマッチするパターン（あるいは制限を非常に高い数値にする）にした場合、ノードの安定性に影響を与える可能性があり、また recon_trace はそれに対し

て何も支援できなくなるかもしれないということに注意してください。同様に、非常に大量の関数呼び出し（関数や `io` の全ての呼び出しなど）をトレースした場合、ライブラリで注意していはいますが、そのノードが処理できるプロセスよりも多くのトレースメッセージが生成されるリスクがあります。

よくわからない場合、最も制限した量でトレースを開始し、少しずつ増やしていってください。

3.4 Exercises

Review Questions

1. Why is debugger use generally limited on Erlang?
2. What are the options you can use to trace OTP processes?
3. What determines whether a given set of functions or processes get traced?
4. How can you stop tracing with `recon_trace`? With other tools?
5. How can you trace non-exported function calls?

Open-ended Questions

1. When would you want to move time stamping of traces to the VM's trace mechanisms directly? What would be a possible downside of doing this?
2. Imagine that traffic sent out of a node does so over SSL, over a multi-tenant system. However, due to wanting to validate data sent (following a customer complain), you need to be able to inspect what was seen clear text. Can you think up a plan to be able to snoop in the data sent to their end through the `ssl` socket, without snooping on the data sent to any other customer?

Hands-On

Using the code at https://github.com/ferd/recon_demo (these may require a decent understanding of the code there):

1. Can chatty processes (`council_member`) message themselves? (*hint: can this work with registered names? Do you need to check the chattiest process and see if it messages itself?*)
2. Can you estimate the overall frequency at which messages are sent globally?
3. Can you crash a node using any of the tracing tools? (*hint: dbg makes it easier due to its greater flexibility*)

おわりに

ソフトウェアの運用とデバッグは決して終わることはありません。新しいバグややややこしい動作がつねにあちこちに出現しつづけるでしょう。いかに整ったシステムを扱う場合でも、おそらく本書のようなマニュアルを何十も書けるくらいのことがあるでしょう。

本書を読んだことで、次に何か悪いことが起きたとしても、**それほど**悪いことにはならないことを願っています。それでも、本番システムをデバッグする機会がおそらく山ほどあることでしょう。いかなる堅牢な橋でも腐食しないように常にペンキを塗り替えるわけです。

みなさんのシステム運用がうまく行くことを願っています。