Stuff Goes Bad:
Erlang in Anger

Fred Hébert 著、elixir.jp 訳

2018 年 6 月 20 日
Git commit ID: 14780a6

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 3 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 4 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

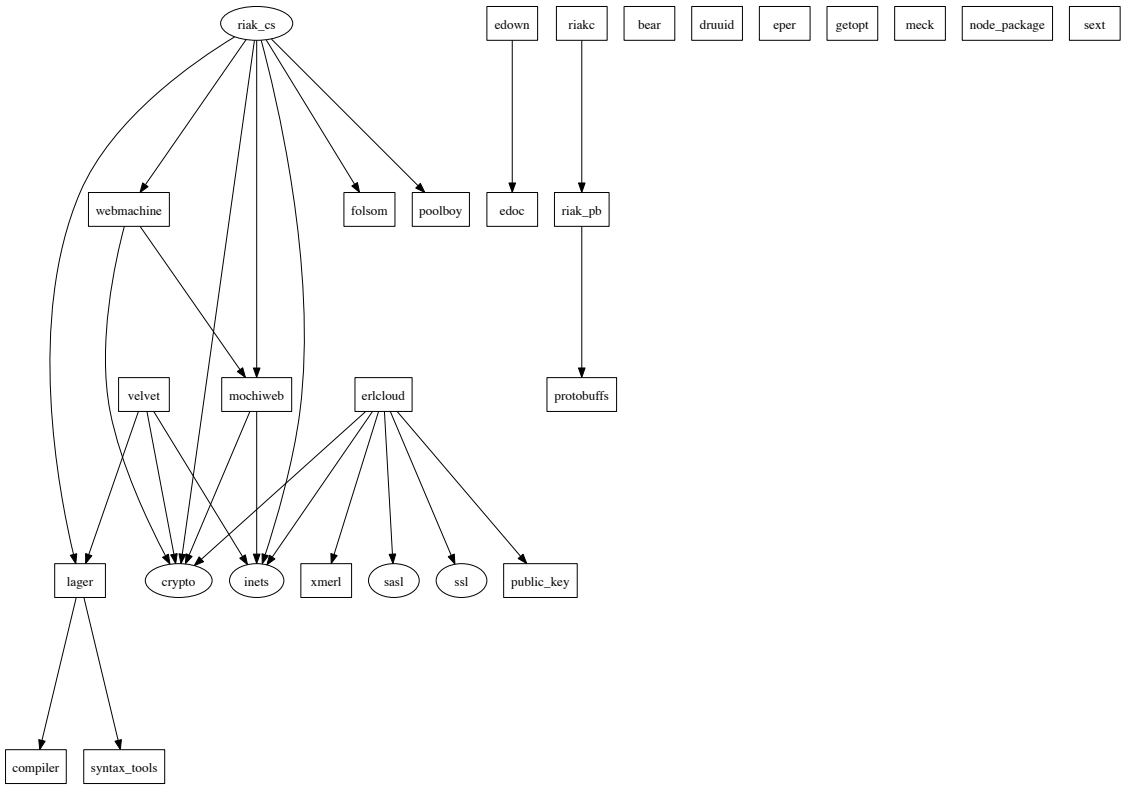


図 1.1　Basho のオープンソースクラウドライブラリである riak_cs の依存関係を表したグラフです。このグラフは kernel や stdlib といった必ず依存するようなものは除いています。楕円はアプリケーションで、四角はライブラリアプリケーションです。

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

オープンソースのErlang製ソフトウェアをビルドする

多くの Erlang に関する書籍は Erlang/OTP アプリケーションのビルド方法に関しては説明していますが、Erlang のコミュニティが開発しているオープンソースとの連携方法まで含めた深い解説を行っているものはほとんどありません。中には意図的にその話題を避けているものさえあります。本章では Erlang でのオープンソースとの連携に関して簡単に案内します。

世間で見かけるオープンソースコードの大半が OTP アプリケーションです。事実、OTP リリースをビルドする多くの人はひとかたまりの OTP アプリケーションとしてビルドしています。

あなたが書いているものがプロジェクトを作っている誰かに使われる可能性がある独立したコードであれば、おそらくそれは OTP アプリケーションでしょう。あなたが作っているものがユーザーがそのままの形でデプロイするような単独で動作するプロダクトであれば、それは OTP リリースでしょう。[1]

サポートされている主なビルドツールは `rebar3` と `erlang.mk` です。前者はビルドツールでありパッケージマネージャーで、Erlang ライブラリと Erlang 製システムを繰り返し使える形で簡単に開発してリリースできるようにしてくれるものです。一方で後者は特殊な makefile で本番用やリリースにはそれほど向いていませんが、より柔軟な記述が出来るようになっています。本章では、`rebar3` がデファクトスタンダードになっていること、自分がよく知っていること、また `erlang.mk` は `rebar3` の依存先としてサポートされている事が多いといった理由から、`rebar3` を使ったビルドに焦点をあてます。

[1] OTP アプリケーションと OTP リリースのビルドの仕方についてはお手元にある Erlang の入門書に譲ります。

2.1 プロジェクト構造

OTP アプリケーションと OTP リリースのプロジェクト構造は異なります。OTP アプリケーションは（あるとすれば）トップレベルスーパーバイザーを 1 つ持っていると想定できます。そしておそらく依存しているものがその下に固まってぶら下がっていると想定できます。OTP リリースは通常複数の OTP アプリケーションから成り、それらがお互いに依存していることもあればそうでないこともあります。これらの事実からアプリケーションの構成をする際に主に 2 つの方法に落ち着きます。

2.1.1 OTP アプリケーション

OTP アプリケーションでは、適切な構造は 1.2 の節で説明したとおりです。

```
1 _build/
2 doc/
3 src/
4 test/
5 LICENSE.txt
6 README.md
7 rebar.config
8 rebar.lock
```

ここで新しいのは_build/ディレクトリと rebar.lock ファイルです。これらは rebar3 によって自動的に生成されます。[2]

このディレクトリに rebar3 がプロジェクトのすべてのビルドアーティファクトを置きます。動作させるのに必要なライブラリやパッケージのローカルコピーなどもそこに含まれます。主要な Erlang ツールはパッケージをグローバルにはインストールせず、[3]かわりにプロジェクト間での衝突を避けるためにすべてをプロジェクトローカルに保存します。

このような依存関係は rebar.config に数行設定を追加するだけで rebar3 に指定できます。

[2] 人によっては rebar3 をアプリケーション内に直接パッケージします。これは rebar3 やその前身を使ったことがない人がライブラリやプロジェクトとブートストラップできるようになされていたものです。自分のシステムのグローバルに rebar3 をインストールして問題ありません。自分のシステムをビルドするのに特定のバージョンが必要であればローカルにコピーを持っておいても良いでしょう。

[3] まだビルドされていないパッケージのローカルキャッシュは除きます。

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

依存するものは git（または hg）のソースあるいは hex.pm からレベル順の幅優先探索で直接取得されます。その後コンパイルされて、特定のコンパイルオプションが設定ファイルの {erl_opts, List}. オプションとともに追加されます。[4]

rebar3 compile を呼んで、すべての依存物をダウンロードし、一度にそれらとあなたのアプリをビルドします。

あなたのアプリケーションのコードを公開するときは、_build/ディレクトリを**含まずに**配布しましょう。他の開発社があなたのアプリケーションと同じものに依存している可能性は高く、何度もそれを配布するのは意味がありません。その場にあるビルドシステム（この場合は rebar3）が重複した項目を見つけ出して、必要なものは 1 度だけしか取得しないようにしてくれるでしょう。

2.1.2 OTP リリース

OTP リリースの場合、構造は少し異なります。リリースはアプリケーションの集まりで、その構造はそれを反映したものになっています。

src にトップレベルアプリケーションを持つ代わりに、アプリケーションは app や lib 内で一階層下にあります。

```
_build/
apps/
  - myapp1/
     - src/
  - myapp2/
     - src/
doc/
```

[4] より詳しい話はこちらを参照してください。https://www.rebar3.org/docs/configuration

```
LICENSE.txt
README.md
rebar.config
rebar.lock
```

この構造は複数の OTP アプリケーションが 1 つのコードレポジトリで管理されているような OTP リリースを生成するときに役立っています。`rebar3` と `erlang.mk` はともにリリースをまとめるときに `relx` ライブラリに依存しています。Systool や Reltol といった他のツールも以前はカバーされていて[5]、ユーザーに多くの力を提供します。

（`rebar.config` 内の）上のようなディレクトリ構造の場合の `relx` 設定タプルは次のようになります。

```
1 {relx, [
2   {release, {demo, "1.0.0"},
3     [myapp1, myapp2, ..., recon]},
4
5   {include_erts, false} % will use local Erlang install
6 ]}
```

`rebar3 release` を呼ぶとリリースをビルドして、`_build/default/rel/`ディレクトリに置かれます。`rebar3 tar` を呼ぶと tarball を`_build/default/rel/demo/demo-1.0.0.tar.gz` に配置し、デプロイ可能となります。

2.2 スーパーバイザーと start_link セマンティクス

複雑な本番システムでは、多くの障害やエラーは一時的なもので、処理を再実行するのは良いことです—Jim Gray の論文[6]では、一時的なバグを扱う場合、システムの *Mean Times Between Failures*（**障害間隔平均時間**）(MTBF) で見た時の 4 回に 1 回程度行うのが良いと引用していました。スーバーバイザーは再起動を行うだけではありません。

Erlang のスーバーバイザーとその監視ツリーの非常に重要な点の一つに、**起動フェーズが同期的に行われる**こと、があります。各 OTP プロセスは兄弟や従兄弟のプロセスの起動を妨げる可能性があります。もしプロセスが死んだら、起動を何度も何度も繰り返され、最終的に起動できるようになる、さもなければ頻繁に失敗することになります。

[5] http://learnyousomeerlang.com/release-is-the-word 訳註: 日本語訳版は https://www.ymotong-poo.com/works/lyse-ja/ja/24_release_is_the_word.html

[6] http://mononcqc.tumblr.com/post/35165909365/why-do-computers-stop

この点が人々がよくある間違いをおかしがちなところです。クラッシュした子プロセスを再起動する前にバックオフやクールダウンの期間はありません。ネットワーク系アプリケーションが初期化フェーズで接続を確立しようとしてリモートサービスが落ちているとき、アプリケーションは無意味な再起動を多数繰り返した後に失敗します。そしてシステムが停止します。

多くの Erlang 開発者がスーパーバイザーにクールダウン期間を持たせるほうがいいという方向の主張をします。私は一つの単純な理由からそれにいは反対です。その理由とは**保証がすべて**ということです。

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

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

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.

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:

```
init(Args) ->
    Opts = parse_args(Args),
    %% you could try connecting here anyway, for a best
    %% effort thing, but be ready to not have a connection.
    self() ! reconnect,
    {ok, #state{sock=undefined, opts=Opts}}.

[...]

handle_info(reconnect, S = #state{sock=undefined, opts=Opts}) ->
    %% try reconnecting in a loop
    case connect(Opts) of
        {ok, New} -> {noreply, S#state{sock=New}};
        _ -> self() ! reconnect, {noreply, S}
    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.

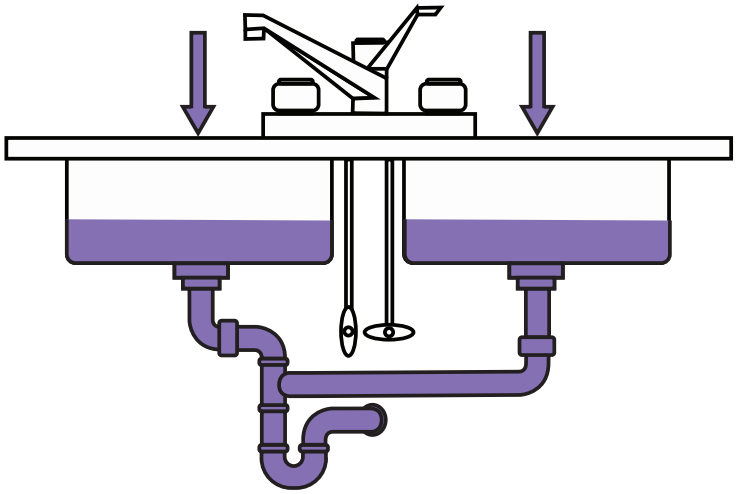
3. The main application's workers (`council_member`) starts a server and connects to it in its `init/1` function. Can you make this connection happen outside of the init function's? Is there a benefit to doing so in this specific case?

第3章

過負荷のための計画をたてる

私が実際に遭遇した最も一般的な障害原因は、圧倒的に稼働中ノードの OutOfMemory です。さらにそれは通常、境界外に出るメッセージキューに関連します。[1] これに対処する方法はたくさんありますが、自身が開発しているシステムを適切に理解することで、どう対処するかを決めることができます。

事象をとても単純化するために、私が取り組むプロジェクトのほとんどは、大きな浴室のシンクとして視覚化することができます。ユーザーとデータ入力が蛇口から流れています。Erlang システム自体はシンクとパイプであり、出力（データベース、外部 API、外部サービスなど）は下水道です。



キューがオーバーフローして Erlang ノードが死んでしまった場合、誰の責任かを解明できます。誰かがシンクに水を入れすぎましたか？ 下水道が渋滞していますか？ あまりにも小さなパイプを設計しましたか？

[1] メッセージキューが問題になる事例は **??** 章、特に **??** 節で説明されます。

どのキューが爆発したかを判断することは必ずしも困難ではありません。これはクラッシュダンプから見つけられる情報です。ただし爆破原因の解明は少し複雑です。プロセスや runtime の調査に基づいて、高速にキューが溢れたのか、ブロックされたプロセスがメッセージを十分高速に処理できないか、などの原因を把握することができます。

最も難しい部分は、それをどのように修正するか決めることです。シンクがあまりにも詰まると、私たちは通常、浴室をもっと大きくすることから始めます（クラッシュしたプログラムの近辺から)。次に、シンクのドレインが小さすぎると分かり、それを最適化します。それから、パイプそのものが狭すぎることが分かり、それを最適化します。下水道がそれ以上受け取ることができなくなるまで過負荷はシステムのさらに下に押し込まれます。そのタイミングで、システム全体への入力処理を助けるために、シンクを追加したり、浴槽を追加したりすることもあります。

そうこうしたところで、もはや浴室の範囲内では事象を改善できないこともあります。あまりにも多くのログが送信されるため、一貫性を必要とするデータベースにボトルネックがあったり、または単に事象を解決するための組織内の知識や人材が不足していることもあります。

こういった点を見つけることで、システムの **真のボトルネック** が何であるかを特定し、過去の全ての良いと思っていた（そして対応コストが高いかもしれない）最適化は、多かれ少なかれは無駄であることも特定できます。

私たちはより賢くなる必要があります。そして、世もレベルアップしています。私たちは、システムに入る情報をより軽いもの (圧縮、より良いアルゴリズムとデータ表現、キャッシングなど) に変換しようとします。

それでもあまりの過負荷が来ることがあります、そしてシステムへの入力を制限するか、入力を廃棄するか、システムがクラッシュするサービスレベルを受け入れるか、を決めなければなりません。これらのメカニズムは、2 つの幅広いストラテジーに分類されます：バックプレッシャーと負荷分散。

この章では、Erlang システムの爆発を引き起こす一般的なイベントとともに、それらを追求します。

3.1 Common Overload Sources

There are a few common causes of queues blowing up and overload in Erlang systems that most people will encounter sooner or later, no matter how they approach their system. They're usually symptomatic of having your system grow up and require some help scaling up, or of an unexpected type of failure that ends up cascading much harder than it should have.

3.1.1 error_logger Explodes

Ironically, the process in charge of error logging is one of the most fragile ones. In a default Erlang install, the `error_logger`[2] process will take its sweet time to log things to disk or over the network, and will do so much more slowly than errors can be generated.

This is especially true of user-generated log messages (not for errors), and for crashes in large processes. For the former, this is because `error_logger` doesn't really expect arbitrary levels of messages coming in continually. It's for exceptional cases only and doesn't expect lots of traffic. For the latter, it's because the entire state of processes (including their mailboxes) gets copied over to be logged. It only takes a few messages to cause memory to bubble up a lot, and if that's not enough to cause the node to run Out Of Memory (OOM), it may slow the logger enough that additional messages will.

The best solution for this at the time of writing is to use `lager` as a substitute logging library.

While `lager` will not solve all your problems, it will truncate voluminous log messages, optionally drop OTP-generated error messages when they go over a certain threshold, and will automatically switch between asynchronous and synchronous modes for user-submitted messages in order to self-regulate.

It won't be able to deal with very specific cases, such as when user-submitted messages are very large in volume and all coming from one-off processes. This is, however, a much rarer occurrence than everything else, and one where the programmer tends to have more control.

3.1.2 Locks and Blocking Operations

Locking and blocking operations will often be problematic when they're taking unexpectedly long to execute in a process that's constantly receiving new tasks.

One of the most common examples I've seen is a process blocking while accepting a connection or waiting for messages with TCP sockets. During blocking operations of this kind, messages are free to pile up in the message queue.

One particularly bad example was in a pool manager for HTTP connections that I had written in a fork of the `lhttpc` library. It worked fine in most test cases we had, and we even had a connection timeout set to 10 milliseconds to be sure it never took too long[3]. After a few weeks of perfect uptime, the HTTP client pool caused an outage when one of the remote

[2] Defined at http://www.erlang.org/doc/man/error_logger.html

[3] 10 milliseconds is very short, but was fine for collocated servers used for real-time bidding.

servers went down.

The reason behind this degradation was that when the remote server would go down, all of a sudden, all connecting operations would take at least 10 milliseconds, the time before which the connection attempt is given up on. With around 9,000 messages per second to the central process, each usually taking under 5 milliseconds, the impact became similar to roughly 18,000 messages a second and things got out of hand.

The solution we came up with was to leave the task of connecting to the caller process, and enforce the limits as if the manager had done it on its own. The blocking operations were now distributed to all users of the library, and even less work was required to be done by the manager, now free to accept more requests.

When there is *any* point of your program that ends up being a central hub for receiving messages, lengthy tasks should be moved out of there if possible. Handling predictable overload[4] situations by adding more processes — which either handle the blocking operations or instead act as a buffer while the "main" process blocks — is often a good idea.

There will be increased complexity in managing more processes for activities that aren't intrinsically concurrent, so make sure you need them before programming defensively.

Another option is to transform the blocking task into an asynchronous one. If the type of work allows it, start the long-running job and keep a token that identifies it uniquely, along with the original requester you're doing work for. When the resource is available, have it send a message back to the server with the aforementioned token. The server will eventually get the message, match the token to the requester, and answer back, without being blocked by other requests in the mean time.[5]

This option tends to be more obscure than using many processes and can quickly devolve into callback hell, but may use fewer resources.

3.1.3 Unexpected Messages

Messages you didn't know about tend to be rather rare when using OTP applications. Because OTP behaviours pretty much expect you to handle anything with some clause in `handle_info/2`, unexpected messages will not accumulate much.

However, all kinds of OTP-compliant systems end up having processes that may not implement a behaviour, or processes that go in a non-behaviour stretch where it overtakes message

[4] Something you know for a fact gets overloaded in production

[5] The `redo` application is an example of a library doing this, in its redo_block module. The [under-documented] module turns a pipelined connection into a blocking one, but does so while maintaining pipeline aspects to the caller — this allows the caller to know that only one call failed when a timeout occurs, not all of the in-transit ones, without having the server stop accepting requests.

handling. If you're lucky enough, monitoring tools[6] will show a constant memory increase, and inspecting for large queue sizes[7] will let you find which process is at fault. You can then fix the problem by handling the messages as required.

3.2 Restricting Input

Restricting input is the simplest way to manage message queue growth in Erlang systems. It's the simplest approach because it basically means you're slowing the user down (applying *back-pressure*), which instantly fixes the problem without any further optimization required. On the other hand, it can lead to a really crappy experience for the user.

The most common way to restrict data input is to make calls to a process whose queue would grow in uncontrollable ways synchronously. By requiring a response before moving on to the next request, you will generally ensure that the direct source of the problem will be slowed down.

The difficult part for this approach is that the bottleneck causing the queue to grow is usually not at the edge of the system, but deep inside it, which you find after optimizing nearly everything that came before. Such bottlenecks will often be database operations, disk operations, or some service over the network.

This means that when you introduce synchronous behaviour deep in the system, you'll possibly need to handle back-pressure, level by level, until you end up at the system's edges and can tell the user, "please slow down." Developers that see this pattern will often try to put API limits per user[8] on the system entry points. This is a valid approach, especially since it can guarantee a basic quality of service (QoS) to the system and allows one to allocate resources as fairly (or unfairly) as desired.

3.2.1 How Long Should a Time Out Be

What's particularly tricky about applying back-pressure to handle overload via synchronous calls is having to determine what the typical operation should be taking in terms of time, or rather, at what point the system should time out.

The best way to express the problem is that the first timer to be started will be at the edge

[6] See Section **??**

[7] See Subsection **??**

[8] There's a tradeoff between slowing down all requests equally or using rate-limiting, both of which are valid. Rate-limiting per user would mean you'd still need to increase capacity or lower the limits of all users when more new users hammer your system, whereas a synchronous system that indiscriminately blocks should adapt to any load with more ease, but possibly unfairly.

of the system, but the critical operations will be happening deep within it. This means that the timer at the edge of the system will need to have a longer wait time that those within, unless you plan on having operations reported as timing out at the edge even though they succeeded internally.

An easy way out of this is to go for infinite timeouts. Pat Helland[9] has an interesting answer to this:

> Some application developers may push for no timeout and argue it is OK to wait indefinitely. I typically propose they set the timeout to 30 years. That, in turn, generates a response that I need to be reasonable and not silly. *Why is 30 years silly but infinity is reasonable?* I have yet to see a messaging application that really wants to wait for an unbounded period of time⋯

This is, ultimately, a case-by-case issue. In many cases, it may be more practical to use a different mechanism for that flow control.[10]

3.2.2 Asking For Permission

A somewhat simpler approach to back-pressure is to identify the resources we want to block on, those that cannot be made faster and are critical to your business and users. Lock these resources behind a module or procedure where a caller must ask for the right to make a request and use them. There's plenty of variables that can be used: memory, CPU, overall load, a bounded number of calls, concurrency, response times, a combination of them, and so on.

The *SafetyValve*[11] application is a system-wide framework that can be used when you know back-pressure is what you'll need.

For more specific use cases having to do with service or system failures, there are plenty of circuit breaker applications available. Examples include `breaky`[12], `fuse`[13], or Klarna's `circuit_breaker`[14].

Otherwise, ad-hoc solutions can be written using processes, ETS, or any other tool available. The important part is that the edge of the system (or subsystem) may block and ask for the right to process data, but the critical bottleneck in code is the one to determine whether that

[9] Idempotence is Not a Medical Condition, April 14, 2012

[10] In Erlang, using the value `infinity` will avoid creating a timer, avoiding some resources. If you do use this, remember to at least have a well-defined timeout somewhere in the sequence of calls.

[11] https://github.com/jlouis/safetyvalve

[12] https://github.com/mmzeeman/breaky

[13] https://github.com/jlouis/fuse

[14] https://github.com/klarna/circuit_breaker

right can be granted or not.

The advantage of proceeding that way is that you may just avoid all the tricky stuff about timers and making every single layer of abstraction synchronous. You'll instead put guards at the bottleneck and at a given edge or control point, and everything in between can be expressed in the most readable way possible.

3.2.3 What Users See

The tricky part about back-pressure is reporting it. When back-pressure is done implicitly through synchronous calls, the only way to know it is at work due to overload is that the system becomes slower and less usable. Sadly, this is also going to be a potential symptom of bad hardware, bad network, unrelated overload, and possibly a slow client.

Trying to figure out that a system is applying back-pressure by measuring its responsiveness is equivalent to trying to diagnose which illness someone has by observing that person has a fever. It tells you something is wrong, but not what.

Asking for permission, as a mechanism, will generally allow you to define your interface in such a way that you can explicitly report what is going on: the system as a whole is overloaded, or you're hitting a limit into the rate at which you can perform an operation and adjust accordingly.

There is a choice to be made when designing the system. Are your users going to have per-account limits, or are the limits going to be global to the system?

System-global or node-global limits are usually easy to implement, but will have the downside that they may be unfair. A user doing 90% of all your requests may end up making the platform unusable for the vast majority of the other users.

Per-account limits, on the other hand, tend to be very fair, and allow fancy schemes such as having premium users who can go above the usual limits. This is extremely nice, but has the downside that the more users use your system, the higher the effective global system limit tends to move. Starting with 100 users that can do 100 requests a minute gives you a global 10000 requests per minute. Add 20 new users with that same rate allowed, and suddenly you may crash a lot more often.

The safe margin of error you established when designing the system slowly erodes as more people use it. It's important to consider the tradeoffs your business can tolerate from that point of view, because users will tend not to appreciate seeing their allowed usage go down all the time, possibly even more so than seeing the system go down entirely from time to time.

3.3 Discarding Data

When nothing can slow down outside of your Erlang system and things can't be scaled up, you must either drop data or crash (which drops data that was in flight, for most cases, but with more violence).

It's a sad reality that nobody really wants to deal with. Programmers, software engineers, and computer scientists are trained to purge the useless data, and keep everything that's useful. Success comes through optimization, not giving up.

However, there's a point that can be reached where the data that comes in does so at a rate faster than it goes out, even if the Erlang system on its own is able to do everything fast enough. In some cases, It's the component *after* it that blocks.

If you don't have the option of limiting how much data you receive, you then have to drop messages to avoid crashing.

3.3.1 Random Drop

Randomly dropping messages is the easiest way to do such a thing, and might also be the most robust implementation, due to its simplicity.

The trick is to define some threshold value between 0.0 and 1.0 and to fetch a random number in that range:

```
-module(drop).
-export([random/1]).

random(Rate) ->
    maybe_seed(),
    random:uniform() =< Rate.

maybe_seed() ->
    case get(random_seed) of
        undefined -> random:seed(erlang:now());
        {X,X,X} -> random:seed(erlang:now());
        _ -> ok
    end.
```

If you aim to keep 95% of the messages you send, the authorization could be written by a call to `case drop:random(0.95) of true -> send(); false -> drop() end`, or a shorter

`drop:random(0.95) andalso send()` if you don't need to do anything specific when dropping a message.

The `maybe_seed()` function will check that a valid seed is present in the process dictionary and use it rather than a crappy one, but only if it has not been defined before, in order to avoid calling `now()` (a monotonic function that requires a global lock) too often.

There is one 'gotcha' to this method, though: the random drop must ideally be done at the producer level rather than at the queue (the receiver) level. The best way to avoid overloading a queue is to not send data its way in the first place. Because there are no bounded mailboxes in Erlang, dropping in the receiving process only guarantees that this process will be spinning wildly, trying to get rid of messages, and fighting the schedulers to do actual work.

On the other hand, dropping at the producer level is guaranteed to distribute the work equally across all processes.

This can give place to interesting optimizations where the working process or a given monitor process[15] uses values in an ETS table or `application:set_env/3` to dynamically increase and decrease the threshold to be used with the random number. This allows control over how many messages are dropped based on overload, and the configuration data can be fetched by any process rather efficiently by using `application:get_env/2`.

Similar techniques could also be used to implement different drop ratios for different message priorities, rather than trying to sort it all out at the consumer level.

3.3.2 Queue Buffers

Queue buffers are a good alternative when you want more control over the messages you get rid of than with random drops, particularly when you expect overload to be coming in bursts rather than a constant stream in need of thinning.

Even though the regular mailbox for a process has the form of a queue, you'll generally want to pull *all* the messages out of it as soon as possible. A queue buffer will need two processes to be safe:

- The regular process you'd work with (likely a `gen_server`);
- A new process that will do nothing but buffer the messages. Messages from the outside should go to this process.

To make things work, the buffer process only has to remove all the messages it can from

[15] Any process tasked with checking the load of specific processes using heuristics such as `process_info(Pid, message_queue_len)` could be a monitor

its mail box and put them in a queue data structure[16] it manages on its own. Whenever the server is ready to do more work, it can ask the buffer process to send it a given number of messages that it can work on. The buffer process picks them from its queue, forwards them to the server, and goes back to accumulating data.

Whenever the queue grows beyond a certain size[17] and you receive a new message, you can then pop the oldest one and push the new one in there, dropping the oldest elements as you go.[18]

This should keep the entire number of messages received to a rather stable size and provide a good amount of resistance to overload, somewhat similar to the functional version of a ring buffer.

The *PO Box*[19] library implements such a queue buffer.

3.3.3 Stack Buffers

Stack buffers are ideal when you want the amount of control offered by queue buffers, but you have an important requirement for low latency.

To use a stack as a buffer, you'll need two processes, just like you would with queue buffers, but a list[20] will be used instead of a queue data structure.

The reason the stack buffer is particularly good for low latency is related to issues similar to bufferbloat[21]. If you get behind on a few messages being buffered in a queue, all the messages in the queue get to be slowed down and acquire milliseconds of wait time. Eventually, they all get to be too old and the entire buffer needs to be discarded.

On the other hand, a stack will make it so only a restricted number of elements are kept waiting while the newer ones keep making it to the server to be processed in a timely manner.

Whenever you see the stack grow beyond a certain size or notice that an element in it is too old for your QoS requirements you can just drop the rest of the stack and keep going from

[16] The `queue` module in Erlang provides a purely functional queue data structure that can work fine for such a buffer.

[17] To calculate the length of a queue, it is preferable to use a counter that gets incremented and decremented on each message sent or received, rather than iterating over the queue every time. It takes slightly more memory, but will tend to distribute the load of counting more evenly, helping predictability and avoiding more sudden build-ups in the buffer's mailbox

[18] You can alternatively make a queue that pops the newest message and queues up the oldest ones if you feel previous data is more important to keep.

[19] Available at: https://github.com/ferd/pobox, the library has been used in production for a long time in large scale products at Heroku and is considered mature

[20] Erlang lists *are* stacks, for all we care. They provide push and pop operations that take O(1) complexity and are very fast

[21] http://queue.acm.org/detail.cfm?id=2071893

there. *PO Box* also offers such a buffer implementation.

A major downside of stack buffers is that messages are not necessarily going to be processed in the order they were submitted — they're nicer for independent tasks, but will ruin your day if you expect a sequence of events to be respected.

3.3.4 Time-Sensitive Buffers

If you need to react to old events *before* they are too old, then things become more complex, as you can't know about it without looking deep in the stack each time, and dropping from the bottom of the stack in a constant manner gets to be inefficient. An interesting approach could be done with buckets, where multiple stacks are used, with each of them containing a given time slice. When requests get too old for the QoS constraints, drop an entire bucket, but not the entire buffer.

It may sound counter-intuitive to make some requests a lot worse to benefit the majority — you'll have great medians but poor 99 percentiles — but this happens in a state where you would drop messages anyway, and is preferable in cases where you do need low latency.

3.3.5 Dealing With Constant Overload

Being under constant overload may require a new solution. Whereas both queues and buffers will be great for cases where overload happens from time to time (even if it's a rather prolonged period of time), they both work more reliably when you expect the input rate to eventually drop, letting you catch up.

You'll mostly get problems when trying to send so many messages they can't make it all to one process without overloading it. Two approaches are generally good for this case:

- Have many processes that act as buffers and load-balance through them (scale horizontally)
- use ETS tables as locks and counters (reduce the input)

ETS tables are generally able to handle a ton more requests per second than a process, but the operations they support are a lot more basic. A single read, or adding or removing from a counter atomically is as fancy as you should expect things to get for the general case.

ETS tables will be required for both approaches.

Generally speaking, the first approach could work well with the regular process registry: you take `N` processes to divide up the load, give them all a known name, and pick one of them to send the message to. Given you're pretty much going to assume you'll be overloaded, randomly

picking a process with an even distribution tends to be reliable: no state communication is required, work will be shared in a roughly equal manner, and it's rather insensitive to failure.

In practice, though, we want to avoid atoms generated dynamically, so I tend to prefer to register workers in an ETS table with `read_concurrency` set to `true`. It's a bit more work, but it gives more flexibility when it comes to updating the number of workers later on.

An approach similar to this one is used in the `lhttpc`[22] library mentioned earlier, to split load balancers on a per-domain basis.

For the second approach, using counters and locks, the same basic structure still remains (pick one of many options, send it a message), but before actually sending a message, you must atomically update an ETS counter[23]. There is a known limit shared across all clients (either through their supervisor, or any other config or ETS value) and each request that can be made to a process needs to clear this limit first.

This approach has been used in `dispcount`[24] to avoid message queues, and to guarantee low-latency responses to any message that won't be handled so that you do not need to wait to know your request was denied. It is then up to the user of the library whether to give up as soon as possible, or to keep retrying with different workers.

3.3.6 How Do You Drop

Most of the solutions outlined here work based on message quantity, but it's also possible to try and do it based on message size, or expected complexity, if you can predict it. When using a queue or stack buffer, instead of counting entries, all you may need to do is count their size or assign them a given load as a limit.

I've found that in practice, dropping without regard to the specifics of the message works rather well, but each application has its share of unique compromises that can be acceptable or not[25].

There are also cases where the data is sent to you in a "fire and forget" manner — the entire system is part of an asynchronous pipeline — and it proves difficult to provide feedback to the end-user about why some requests were dropped or are missing. If you can reserve a special type of message that accumulates dropped responses and tells the user "`N` messages

[22] The lhttpc_lb module in this library implements it.

[23] By using `ets:update_counter/3`.

[24] https://github.com/ferd/dispcount

[25] Old papers such as Hints for Computer System Designs by Butler W. Lampson recommend dropping messages: "Shed load to control demand, rather than allowing the system to become overloaded." The paper also mentions that "A system cannot be expected to function well if the demand for any resource exceeds two-thirds of the capacity, unless the load can be characterized extremely well." adding that "The only systems in which cleverness has worked are those with very well-known loads."

were dropped for reason `X`", that can, on its own, make the compromise far more acceptable to the user. This is the choice that was made with Heroku's logplex log routing system, which can spit out L10 errors, alerting the user that a part of the system can't deal with all the volume right now.

In the end, what is acceptable or not to deal with overload tends to depend on the humans that use the system. It is often easier to bend the requirements a bit than develop new technology, but sometimes it is just not avoidable.

3.4 Exercises

Review Questions

1. Name the common sources of overload in Erlang systems
2. What are the two main classes of strategies to handle overload?
3. How can long-running operations be made safer?
4. When going synchronous, how should timeouts be chosen?
5. What is an alternative to having timeouts?
6. When would you pick a queue buffer before a stack buffer?

Open-ended Questions

1. What is a *true bottleneck*? How can you find it?
2. In an application that calls a third party API, response times vary by a lot depending on how healthy the other servers are. How could one design the system to prevent occasionally slow requests from blocking other concurrent calls to the same service?
3. What's likely to happen to new requests to an overloaded latency-sensitive service where data has backed up in a stack buffer? What about old requests?
4. Explain how you could turn a load-shedding overload mechanism into one that can also provide back-pressure.
5. Explain how you could turn a back-pressure mechanism into a load-shedding mechanism.
6. What are the risks, for a user, when dropping or blocking a request? How can we prevent duplicate messages or missed ones?
7. What can you expect to happen to your API design if you forget to deal with overload, and suddenly need to add back-pressure or load-shedding to it?

第II部

Diagnosing Applications

第4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。

4.1 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、4.1 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

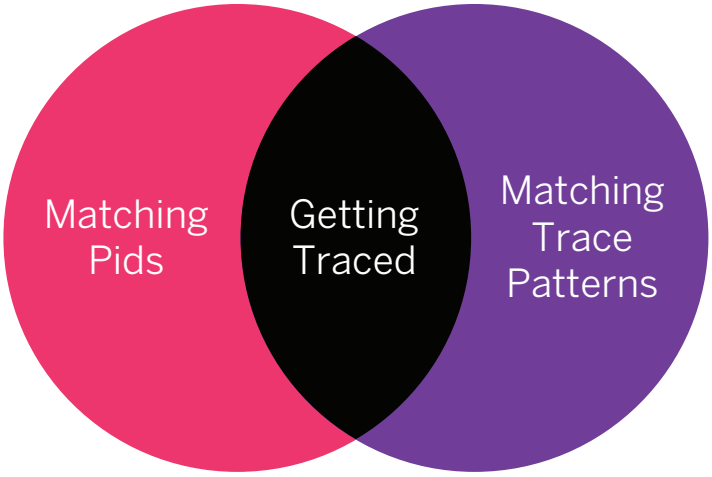


図 4.1　トレースされるのは、pid 指定とトレースパターンの交差した箇所です

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。

4.2　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。

4.3 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。

4.4 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。