

# Stuff Goes Bad: Erlang in Anger

Fred Hébert 著、elixir.jp 訳

2018 年 6 月 23 日

Git commit ID: [08a1a4c](#)



# STUFF GOES BAD: ERLANG IN ANGER

STUFF GOES BAD: ERLANG IN ANGER is a trademark of the publisher. All rights reserved. No part of this publication may be reproduced without the publisher's permission.



Fred Hébert および Heroku 社著の **Stuff Goes Bad: Erlang in Anger** は [クリエイティブ・コモンズ 表示 - 非営利 - 継承 4.0 国際ライセンス](#) として公開されています。また日本語訳もライセンス条件は原文に従います。

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 Aloï**、**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。

はじめに	vi
第 I 部 Writing Applications	1
第 1 章 コードベースへの飛び込み方	2
1.1 生の Erlang	2
1.2 OTP アプリケーション	3
1.3 OTP リリース	7
1.4 演習	8
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	9
2.1 プロジェクト構造	9
2.2 スーパーバイザーと start_link セマンティクス	12
2.3 演習	16
第 3 章 過負荷のための計画をたてる	18
3.1 よくある過負荷の原因	19
3.2 入力を制限する	22
3.3 Discarding Data	24
3.4 演習	30
第 II 部 Diagnosing Applications	31
第 4 章 リモートノードへの接続	32

4.1	ジョブ制御モード . . . . .	33
4.2	Remsh . . . . .	34
4.3	SSH デーモン . . . . .	34
4.4	名前付きパイプ . . . . .	35
4.5	演習 . . . . .	36
第 5 章	ランタイムメトリクス . . . . .	37
5.1	グローバルビュー . . . . .	38
5.2	Digging In . . . . .	43
5.3	演習 . . . . .	51
第 6 章	クラッシュダンプを読む . . . . .	53
6.1	一般的な見方 . . . . .	53
6.2	Full Mailboxes . . . . .	56
6.3	非常に多い（もしくは非常に少ない）プロセス . . . . .	57
6.4	大量のポート数 . . . . .	57
6.5	メモリ割り当てができない . . . . .	57
6.6	Exercises . . . . .	58
第 7 章	Memory Leaks . . . . .	59
7.1	Common Sources of Leaks . . . . .	59
7.2	Binaries . . . . .	64
7.3	Memory Fragmentation . . . . .	67
7.4	Exercises . . . . .	74
第 8 章	CPU and Scheduler Hogs . . . . .	76
8.1	Profiling and Reduction Counts . . . . .	76
8.2	System Monitors . . . . .	78
8.3	Exercises . . . . .	79
第 9 章	トレース . . . . .	81
9.1	トレースの原則 . . . . .	82
9.2	Recon によるトレース . . . . .	83
9.3	実行例 . . . . .	85
9.4	演習 . . . . .	87
Conclusion		89

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。	7
7.1	Erlang's Memory allocators and their hierarchy. Not shown is the special <b>super carrier</b> , optionally allowing to pre-allocate (and limit) all memory available to the Erlang VM since R16B03. . . . .	69
7.2	Example memory allocated in a specific sub-allocator . . . . .	70
7.3	Example memory allocated in a specific sub-allocator . . . . .	71
7.4	Example memory allocated in a specific sub-allocator . . . . .	72
9.1	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 . . . . .	83



## 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。<sup>1</sup> 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」<sup>2</sup>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<sup>3</sup>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。

<sup>1</sup> 生命に関わるシステムは通常この議論の対象外です。

<sup>2</sup> Erlang 界限の人々は、最近で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。

<sup>3</sup> Jim Gray の [Why Do Computers Stop and What Can Be Done About It?](#)によれば、132 個中 131 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。

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。<sup>45</sup>

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<sup>6</sup>などです。

---

<sup>4</sup> 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。

<sup>5</sup> 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。

<sup>6</sup> 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**



## 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 4 章)、取得できる基本的な実行時のメトリクス (第 5 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 6 章)、メモリーリークの検出方法と修正方法 (第 7 章)、そして暴走した CPU 使用率の検出方法 (第 8 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`<sup>7</sup> を使ってトレースする方法を説明します。(第 9 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。

---

<sup>7</sup> <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<sup>1</sup> 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。<sup>2</sup> 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

---

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

<sup>2</sup> 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。

```
{application, dispcount, [
  {description, "A dispatching library for resources and task "
    "limiting based on shared counters"},
  {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。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。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。<sup>3</sup>

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状態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、何かおかしい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。<sup>4</sup>
- **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。

---

<sup>3</sup> 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。

<sup>4</sup> 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。



- `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。<sup>5</sup>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ではありません。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見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きな地図を描くのに役立つでしょう。似たダイア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。<sup>6</sup>似たダイアグラムが `observer`<sup>7</sup>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。

<sup>5</sup> 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。

<sup>6</sup> このスクリプトは `graphviz` に依存しています。

<sup>7</sup> [http://www.erlang.org/doc/apps/observer/observer\\_ug.html](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数は大小違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<sup>8</sup> や rebar3<sup>9</sup> のドキュメントサイトや erlang.mk<sup>10</sup> 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<sup>11</sup>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](#)。<sup>12</sup>

## 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](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。ダイア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。

---

<sup>8</sup> <https://github.com/erlware/relx/wiki>

<sup>9</sup> <https://www.rebar3.org/docs/releases>

<sup>10</sup> <http://erlang.mk/guide/relx.html>

<sup>11</sup> 多数

<sup>12</sup> 訳注: 日本語訳版 [https://www.ymotongpoo.com/works/lyse-ja/ja/24\\_release\\_is\\_the\\_word.html](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。<sup>1</sup>

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。

### 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

---

<sup>1</sup> 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。

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。<sup>2</sup>

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、<sup>3</sup>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。

---

```
1 {deps, [  
2   %% Hex.pm Packages  
3   myapp,  
4   {myapp, "1.0.0"},  
5   %% source dependencies
```

---

<sup>2</sup> 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。

<sup>3</sup> 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。

```
6 {myapp, {git, "git://github.com/user/myapp.git", {ref, "aef728"}}},
7 {myapp, {git, "https://github.com/user/myapp.git", {branch, "master"}}},
8 {myapp, {hg, "https://othersite.com/user/myapp", {tag, "3.0.0"}}}
9 ]}.
```

---

依存するものは git（または hg）のソースあるいは [hex.pm](https://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。<sup>4</sup>

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

---

<sup>4</sup> 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>



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<sup>5</sup>、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<sup>6</sup>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。スーパーバイザーは再起動を行うだけではありません。

Erlang のスーパ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、さもないと頻繁に失敗することになります。

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。

---

<sup>5</sup> <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](https://www.ymotong-poo.com/works/lyse-ja/ja/24_release_is_the_word.html)

<sup>6</sup> <http://mononocqc.tumblr.com/post/35165909365/why-do-computers-stop>

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良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 すべては保証のため

プロセスを再起動するということは、プロセスを安定した、既知の状態に戻すことです。そこから、再度処理を試します。もし初期化が安定していなければ、監視の価値は非常に低いでしょう。初期化プロセスは何が起きても安定しているべきです。そのような理由から、あるプロセスの兄弟プロセスや従兄弟プロセスが後に起動したときに、それらのプロセスはシステムにおいて自分が起動する以前に立ち上がった部分は健康であると知った状態で立ち上げられるでしょう。

もしこのような安定した状態を提供しない場合、あるいはシステム全体を非同期で起動しようとした場合、ループ内での `try ... catch` では提供されないような、このディレクトリ構造による利益をほとんど享受できないでしょう。

監視されたプロセスは**ベストエフォートではなく**、初期化フェーズが行われることを**保証します**。つまりあなたがデータベースやサービスのクライアントを書いている場合、何が起きても常に接続可能であると言いたい場合を除いて、初期化フェーズの一部として接続が確立したかを書く必要はありません。

たとえば、データベースが同じホストにあり、あなたの Erlang システムよりも前に起動されているはずだとすれば、初期化の最中に接続を強制できるでしょう。そうであれば再起動もうまく動くはずですが、これらの保証を壊す理解不能あるいは予期せぬことが起きた場合には、ノードがクラッシュします。それは期待する動作です。システムを起動する前提条件が満たされなかったからです。それはシステム全体に落ちたと伝えるべきアサーションです。

一方で、あなたのデータベースがリモートホストにある場合、接続に失敗する可能性があります。うまく行かない<sup>7</sup>というのは分散システムの現実です。このような場合、クライアントプロセスでできる唯一の保証は、クライアントはリクエストはさばけるということだけであり、データベースとの通信に関してはそのような保証は出来ません。クライアントは通信の断絶が起きている間は、たとえばすべての呼び出しに対して `{error, not_connected}` を返すといったことができるでしょう。

その上でデータベースへの再接続はクールダウンでもバックオフでも何でもいいですが、とりあえずあなたが最適だと思うそういうものを使って、システムの安定性を損なわずに行えるでしょう。最適化の一貫として初期化フェーズに行くこともできるでしょうが、何かが切断された場合にプロセスがあとになっても再接続できるようにすべきです。

外部サービスが落ちることが予想されるのであれば、システムでそのサービスの存在を保証すべ

---

<sup>7</sup> あるいはレイテンシが限りなく遅くなって障害状態と見分けがつかなくなったり

きではありません。私たちが扱っているのは現実世界であって、外部依存先に障害が起きることは常にあり得るのです。

### 2.2.2 副作用

もちろん、そのようなクライアントを呼ぶライブラリやプロセスは、データベースなしに動作することを想定していなければ、その後エラーを吐きます。それはまったく異なる問題空間のまったく異なる問題で、ビジネスルールやクライアントで何が出来て何が出来ないかなどに依存するものです。しかし、ワークアラウンド可能なものもあります。たとえば、運用系のメトリクスを保存するサービスのクライアントを考えましょう。—クライアントを呼び出すコードはシステム全体に悪影響を及ぼすことなくエラーを無視できるでしょう。

初期化と監視での手法の違いは、クライアント自身ではなくクライアントの呼び出し側が障害にどの程度耐えられるか決められるという点です。障害耐性のあるシステムを設計する場合にこの点は非常に重要な特徴です。そうです、スーパーバイザーとは再起動についてですが、それは既知の安定した状態への再起動であるべきです。

### 2.2.3 例: 接続を保証しない初期化

次のコードはプロセスの状態として接続を保証しようとしています。

---

```
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     %% ループで再接続を試みる
10    case connect(Opts) of
11        {ok, New} -> {noreply, S#state{sock=New}};
12        _ -> self() ! reconnect, {noreply, S}
13    end;
```

---

かわりに、次のように書き換えることを検討してみましょう。

```

1  init(Args) ->
2      Opts = parse_args(Args),
3      %% いずれにせよここでベストエフォートで接続を試みて
4      %% 接続が出来なかった場合には備えましょう。
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;

```

---

初期化をより少ない保証で行えます。つまり**接続可能**から**接続マネージャー利用可能**という保証に変わりました。

## 2.2.4 要約

私に関わってきた本番システムは両方の手法が混ざったものでした。

設定ファイル、ファイルシステムへのアクセス (たとえばログ目的)、依存しているローカルのリソース (ログ用に UDP ポートを開ける)、ディスクやネットワークから安定した状態を復元するなどといったものはスーパーバイザーの要件として、どれだけ長時間かかっても同期的に読み込むことに決めるでしょう。(アプリケーションによってはまれに起動に 10 分以上かかることもあるでしょうが、それはおそらく間違った情報を配信してしないよう、基準の状態として動作するために**必要な**ギガバイト単位のデータを同期しているので構わないのです。)

一方で、ローカルではないデータベースや外部のサービスに依存しているコードは、より素早い監視ツリーのブートとともに部分的な起動を適用させるでしょう。その理由は、もし通常の処理の最中に障害が何度も起きることが予想されるのであれば、それが始めに起きるか後で起きるかに違いはありません。いずれにせよその対応をしなければならないことには変わりはなく、またシステムのその部分に関して、保証の厳格さが少ないほうがしばしば良い解決策になります。

## 2.2.5 アプリケーション戦略

何があろうとも、ノードで障害が連続して起きることはノードの死刑ではありません。システムが様々な OTP アプリケーションに分割されてしまえば、ノードにとってどのアプリケーションが致命的でどれがそうでないかを決められるようになります。各 OTP アプリケーションは3つの方法で起動できます。temporary (一時的)、transient (暫定的)、permanent (永続的) のどれか一つを選択でき、`application:start(Name, Type)` と手動で起動するか、あるいはリリース内の設定ファイルに書くことで可能です。

- **permanent**: `application:stop/1` を手動で実行した場合を除いて、アプリケーションが終了したとき、システム全体が落ちます。
- **transient**: アプリケーションが `normal` が原因で終了した場合は問題ありません。他の理由で終了した場合はシステム全体を終了させます。
- **temporary**: アプリケーションはいかなる理由でも停止できます。停止したことは報告されますが、何も悪いことは起きません。

また**インクルードされたアプリケーション**としてアプリケーションを起動することも可能です。これは自分のスーパーバイザーの下で起動し、再起動も独自の戦略で行います。

## 2.3 演習

### 復習問題

1. Erlang の監視ツリーは深さ優先で起動されますか。それとも幅優先ですか。同期的ですか、非同期ですか。
2. 3つのアプリケーション戦略はなんでしょう。それぞれ何をするものでしょう。
3. アプリケーションとリリースのディレクトリ構造の主な違いはなんでしょう。
4. リリースを使うべき場面はいつでしょう。
5. プロセスの `init` 関数に含まれるべき状態の例を2つ挙げてください。また含まれるべきでない状態の例も2つ挙げてください。

### ハンズオン

[https://github.com/ferd/recon\\_demo](https://github.com/ferd/recon_demo) にあるコードを使って次に答えてください。

1. リリースにホストされている `main` アプリケーションを抜き出して独立したアプリケーションにして、他のプロジェクトにインクルード可能にしてください。

2. アプリケーションをどこかにホストしてください。(GitHub、Bitbucket、ローカルサーバーなど) そしてそのアプリケーションに依存するリリースをビルドしてください。
3. main アプリケーションのワーカー (council\_member) はサーバーの起動とそこへの接続を自身の `init/1` 関数で行っています。この接続を `init` 関数の外で行なえますか。このアプリの用途においてそうすることの利点はあるでしょうか。



## 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。<sup>1</sup> 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。

<sup>1</sup> メッセージキューが問題になる事例は 6 章、特に 6.2 節で説明されます。

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？

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ブ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）に変換しようとしています。

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つの幅広い戦略に分類されます：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 よくある過負荷の原因

どんなにうまく取り組んでもたいていの人が遅かれ早かれ遭遇する、キューを爆発させ Erlang システムを過負荷にさせるよくある原因がいくつかあります。そうした原因は通常システムを大きくさせスケールアップさせる何かしらの助けが必要な兆候を表していたり、はたまた想定よりもずっと厳しく連鎖してしまう予期せぬ障害だったりします。

### 3.1.1 error\_logger の爆発

皮肉なことにエラーログの責任を担うプロセスがもっとも壊れやすい部分の一つです。デフォルトの Erlang のインストールでは、`error_logger`<sup>2</sup> プロセスは優雅にディスクやネットワーク越しにログを書き込み、それはエラーが生成されるよりもずっと遅い速度での書き込みになってしまいます。

特に (エラーではなく) ユーザーが生成したログメッセージや大きなプロセスがクラッシュしたときのログではこうしたことが起きます。前者に関しては、`error_logger` は任意のレベルのメッセージが継続的にやってくることを想定していないからです。例外的な場合のみを想定していて、大量のトラフィックが来る場合は想定していないのです。後者に関しては、(プロセスのメールボックスも含めた) プロセス全体の状態がログされるためにコピーされるからです。たった数行のメッセージでもメモリを大量に増やす原因となりますし、もしそれがノードを Out Of Memory (OOM) にさせるに至らなくても、追加のメッセージの書き込み速度を下げるには十分です。

これに対する現行執筆時点での最適解は `lager` を代替のログライブラリとして使うことです。

`lager` がすべての問題を解決するわけではない一方で、分量が多いログメッセージを切り詰めて、補足的にある閾値を超えたときには OTP が生成したエラーメッセージを破棄して、ユーザーが送ったメッセージ用に自主規制のために自動的に非同期モードと同期モードを切り替えます。

ユーザーが送ったメッセージのサイズが非常に大きくかつワンオフのプロセスからやってくる、というような非常に固有の状況には対応出来ません。しかしながら、これは他の状況に比べるとずっと起こりにくいもので、プログラマーがずっと管理しやすい状況です。

### 3.1.2 ロックとブロック操作

ロック操作とブロック操作は新しいタスクを継続的に受け取るプロセス内で予想外に実行が長くなってしまうときに、しばしば問題になります。

私が発見してきたもっともよくある例は、接続を受け入れる間、あるいは TCP ソケットでメッセージを待つ間プロセスがブロックするというものです。このようなブロック操作の間、メッセージは好きなだけメッセージキューに積み上がっていきます。

特に悪い状況の例が、`lhttpc` ライブラリのフォークの中で私が書いた HTTP 接続用のプールマネージャー内にありました。私達の運用状況においてはたいていはうまく動いていて、さらに接続のタイムアウトも 10 ミリ秒に設定して、接続待ちが長くなりすぎないようにしていました。<sup>3</sup> 数週間完璧に稼働したあとに、リモートサーバーの一つが落ちたときに HTTP クライアントプールが供給できない状態になりました。

---

<sup>2</sup> [http://www.erlang.org/doc/man/error\\_logger.html](http://www.erlang.org/doc/man/error_logger.html) で定義されています。

<sup>3</sup> 10 ミリ秒は非常に短いですが、リアルタイムビiddingに使用される共用サーバーでは問題ありません。

このデグレの背後にある原因は、リモートサーバーが落ちたときに、突然すべての接続操作が最低 10 ミリ秒かかるようになりました。この 10 ミリ秒は接続試行がまだ断念されるよりも短い時間です。中央のプロセスに秒間 9,000 メッセージが届くようになったあたりでは、各接続試行は通常 5 ミリ秒以下で、この障害と同様の状況になったのは、およそ秒間 18,000 メッセージが届くようになったあたりで、このあたりで手に負えなくなりました。

私達がいたった解決策は呼び出し元のプロセスに接続のタスクを譲って、プールマネージャーが自動で制限したかのように制限を強制することにしました。これでブロック操作はライブラリの全ユーザーに分散され、プールマネージャーによって行われるべき仕事はずっと少なくなり、より多くのリクエストを受け付けられるようになりました。

プログラム内でメッセージを受信するための中央的なハブになっている場所が**一つでも**あれば、できれば時間がかかるタスクはそこから取り除くべきでしょう。より多くのプロセスを追加することで予測可能な過負荷<sup>4</sup>に対応する—ブロック操作に対処する、またはかわりに”main” プロセスがブロックする間のバッファとして機能する—のは良いアイデアです。

本質的には並行ではない処理に対してより多くのプロセスを管理する複雑さが増すので、守りのプログラミングを始める前に確実にその実装が必要であることを確認しましょう。

他の選択肢としては、ブロッキングタスクを非同期なものに変形させることです。もし処理がそうした変更を受け入れられるものであれば、長時間実行されるジョブを起動して、そのジョブの一意な識別子としてのトークンともともとのリクエスト元を保持します。リソースが使えるようになったら、リソースからサーバーに対して先述のトークンと一緒にメッセージを送り返させます。結果サーバーはメッセージを受け取り、トークンとリクエスト元を対応させ、その間他のリクエストにブロックされることなく結果を返します。<sup>5</sup>

この選択肢は多くのプロセスを使う方法に比べてあいまいなものになりがちで、すぐにコールバック地獄になりえますが、使うリソースは少なくなるでしょう。

### 3.1.3 予期せぬメッセージ

OTP アプリケーションを使っている場合は知らないメッセージを受け取ることはまれです。なぜなら OTP ビヘイビアはすべてを `handle_info/2` にある節で処理することを期待しているので、予期せぬメッセージがたまることはあまりないでしょう。

しかしながら、あらゆる OTP 互換システムはビヘイビアを実装していないプロセスやメッセージハンドリングを頑張るビヘイビアではない方向で実装してしまったプロセスを持つことになりま

---

<sup>4</sup> 本番環境で事実に基づいて過負荷になるもの

<sup>5</sup> **redo** アプリケーションはこうした処理を行うライブラリの例です。redo の `redo_block` モジュールにその処理があります。このあまりドキュメント化されていないモジュールは、パイプライン接続をブロッキングう接続にしますが、呼び出し元に対してパイプラインの面を維持している間だけそうします。—これによって呼び出し元はタイムアウトが発生したときに、サーバーがリクエストの受け入れを止めることなく、すべての未達の呼び出しが失敗したわけではなく 1 つの呼び出しだけが失敗したとわかります。

す。あなたが十分に幸運であれば、監視ツール<sup>6</sup>が定常的なメモリ増加を示していて、大きなキューサイズを点検することで<sup>7</sup>どのプロセスに問題があるかわかるでしょう。そのあと、メッセージを必要のように処理することで問題を修正できます。

## 3.2 入力を制限する

Erlang システム内のメッセージキューが大きくなるのを管理する最も単純な方法は入力を制限することです。基本的にユーザーとのやり取りを遅くさせ（バックプレッシャーをかけています）ていることを意味しているため、追加の最適化の必要もなくすぐに問題を修正するという理由から最も単純な手法なのです。一方で、ユーザーにとっては本当にひどい体験となります。

データの入力を制限する最もよく知られた方法は、無制限に同期的に成長する可能性があるキューを持ったプロセスを呼び出すことです。次のリクエストに移る前にレスポンスを求めるようにすれば、一般的に問題の直接の原因は確実に軽減されるでしょう。

この手法の難しい部分はキューの成長の原因となるボトルネックは通常システムの周辺部ではなく、システムの深層部にあって、この問題が顕在化する前に行うほぼすべての最適化のあとに見つけることになります。このようなボトルネックはだいたいデータベースの操作、ディスクの操作、あるいはネットワーク越しのサービスでしょう。

これはつまり同期的な動作をシステムの深層部に導入すると、おそらく各階層ごとにバックプレッシャーに対応し、システムの周辺部までたどり着いてユーザーに「もう少しゆっくりしてください」と言えるようになるまで対応する必要が出て来るということです。このパターンを理解した開発者はしばしばユーザーごとにシステムのエン트리ポイントに API 制限を設けようとします。<sup>8</sup>特に基本的なシステムに対するサービスのクオリティ（QoS）を保証でき、リソースを公平に（あるいは不公平に）望んだとおりにリソースを割り当てられるので、これは正当なやり方です。

### 3.2.1 タイムアウトはどれくらいの長さであるべきか

同期呼び出しによる過負荷に対処するためにバックプレッシャーを適用することにおいて特に扱いづらいのは、システムがどのタイミングでタイムアウトするべきかよりも、処理は通常どれくらいの時間がかかるのかを決定しなければならないことです。

この問題を最もうまく表現しようと思うと、最初のタイマーがシステムの周辺部で開始されるけ

---

<sup>6</sup> 5.1 の節を見ましょう

<sup>7</sup> 5.2.1 の節を見ましょう

<sup>8</sup> すべてのリクエストを等しく遅くするか、あるいは速度制限を設けるかはトレードオフがあって、ともに有効です。より多くの新規ユーザーがシステムにアクセスしてきたときに、ユーザーごとに速度制限をするということは依然としてキャパシティを大きくする必要がある、あるいはすべてのユーザーの限度を下げる必要があります。一方で、無差別にブロックする同期的なシステムはもっと簡単にあらゆる負荷に適用できますが、おそらく不公平でしょう。

れども、致命的な処理はシステムの深層部で起きているというような具合です。つまり、システムの周辺部にあるタイマーはシステムの深層部にあるタイマーよりも長い時間待つ必要があるでしょう。ただし、深層部で処理が成功していたとしても周辺部では操作はタイムアウトしたということにしたいのであれば別です。

この状況を脱する簡単な方法はタイムアウトを無期限にすることです。Pat Helland<sup>9</sup>がこれに対して興味深い回答をしています。

アプリケーション開発者によってはタイムアウトを設定せず、無制限に待機しても良いと主張するかもしれません。私がよく、彼らはタイムアウトを 30 年に設定した、ということにしています。そして次に私は愚かではなく合理的である必要があるんだという返事が来ます。**なぜ 30 年は愚かで、無制限は合理的なんでしょう？** 私は無制限に返事を待つようなメッセージアプリを見たことがありません...

つまり、究極的にはケースバイケースな問題です。多くの場合、フロー制御には異なる機構を使うほうがより実用的でしょう。<sup>10</sup>

### 3.2.2 許可を求める

バックプレッシャーのもう少し単純な例はブロックしたいリソース、それも速く出来ずビジネスやユーザーにとって致命的なリソースの確認です。呼び出し元がリソース要求の権利を求めたりそのリソースを使うようなモジュールやプロシージャの裏で、これらのリソースのロックしましょう。使われる変数は様々あって、メモリ、CPU、全体の負荷、呼び出し回数の上限、並行処理、応答時間、これらの組み合わせ、などです。

**Safety Valve**<sup>11</sup>アプリケーションはシステム全体に及ぶフレームワークで、バックプレッシャーが必要だとわかっているときに使えます。

サービスやシステムの障害により関係しそうなユースケースには、多くのサーキットブレーカーアプリケーションがあります。たとえば **breaky**<sup>12</sup>、**fuse**<sup>13</sup>、Klarna の **circuit\_breaker**<sup>14</sup> といった具合です。

または、プロセス、ETS、あるいはその他のツールなどを使ってアドホックな解決策をすることも出来ます。重要なのは、システムの周辺部（あるいはサブシステム）がデータをブロックして、データを処理する権利を求めるけれども、コード内の致命的なボトルネックは権利が許可されるか

---

<sup>9</sup> [Idempotence is Not a Medical Condition](#) 2012 年 4 月 14 日発行

<sup>10</sup> Erlang では、**infinity** という値を使うことでタイマーを作ることを回避でき、それによってリソースを多少節約出来ます。これを使う場合は一連の呼び出しの中のどこかで最低一つはきちんとタイムアウトを設定することを肝に命じておきましょう。

<sup>11</sup> <https://github.com/jlouis/safetyvalve>

<sup>12</sup> <https://github.com/mmzeeman/breaky>

<sup>13</sup> <https://github.com/jlouis/fuse>

<sup>14</sup> [https://github.com/klarna/circuit\\_breaker](https://github.com/klarna/circuit_breaker)



どうかを決めるものです。

このように進める利点は、タイマーや抽象化のあらゆる層を同期的にするというような厄介なものを単純すべて避けられることです。かわりに、ボトルネックや周辺部の特定の場所あるいは制御点に見張りを置いて、その間にあるものはすべて最も読みやすい形で表現できます。

### 3.2.3 ユーザーが見るもの

バックプレッシャーで厄介なのはそれを報告することです。バックプレッシャーが暗黙的に同期呼び出し経由で行われたとき、それが過負荷によって起きていると知る唯一の方法はシステムが遅くなってあまり使えなくなる場合だけです。悲しいことに、これはハードウェアやネットワークの不調や、関係ない過負荷、そして遅いクライアントに起こりうる兆候でもあります。

システムの応答性によってシステムがバックプレッシャーを適用しているかを知ろうとすることは、その人に熱があるという診察結果から病気を診断しようとするようなものです。

機構として、許可を求めることは、一般的に何が起きているかを明示的に報告できるようなインターフェースを定義できるようにしています。たとえばシステム全体が過負荷になっていたり、あるいは処理を実行したり適宜調整できる限界の速度になったりした場合の報告などです。

システムを設計するときには選択をしなければなりません。ユーザーにアカウントごとの制限を設けるか、あるいはシステム全体で一つの制限を持つかです。

システム全体あるいはノード全体での制限というのは通常実装が簡単ですが、不公平に成りうるという欠点があります。あるユーザーがリクエストの 99% を行っていると、そのせいで他の大多数のユーザーがプラットフォームを使えない状態になってしまうでしょう。

一方で、アカウントごとの制限は非常に公平で、通常の制限よりもゆるい制限になるプレミアムユーザーを設定するというような気の利いたこともできるようになります。これは極めて素晴らしいことですが、欠点もあって、システムを使うユーザーの数が増えるほど、快適に動作するためのシステム全体の制限は引き上げられます。1 分間に 100 リクエストを投げられるユーザー 100 人がいると全体では 1 分間に 10000 リクエストとなります。同じ制限速度で 20 人新規ユーザーを追加すると、とたんにクラッシュが大量に発生するようになります。

システムを設計したときに作ったエラーに対するセーフマージンはユーザーの数が増えるにつれ徐々に失われていきます。その観点からビジネスが耐えられるかのトレードオフを考慮することが重要です。なぜなら、ユーザーはシステム全体がときどき落ちてしまうことよりもずっと、自分に割り当てられた使用量がいつも使えないことのほうが嫌だと思うからです。

## 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<sup>15</sup> 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 its mail box and put them in a queue data structure<sup>16</sup> 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.

---

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

<sup>16</sup> The `queue` module in Erlang provides a purely functional queue data structure that can work fine for such a buffer.

Whenever the queue grows beyond a certain size<sup>17</sup> 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.<sup>18</sup>

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**<sup>19</sup> 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<sup>20</sup> 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<sup>21</sup>. 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 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.

---

<sup>17</sup> 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

<sup>18</sup> 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.

<sup>19</sup> 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

<sup>20</sup> Erlang lists **are** stacks, for all we care. They provide push and pop operations that take  $O(1)$  complexity and are very fast

<sup>21</sup> <http://queue.acm.org/detail.cfm?id=2071893>

### 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`<sup>22</sup> 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<sup>23</sup>. 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`<sup>24</sup> 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<sup>25</sup>.

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 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.

---

<sup>22</sup> The `lhttpc_lb` module in this library implements it.

<sup>23</sup> By using `ets:update_counter/3`.

<sup>24</sup> <https://github.com/ferd/dispcount>

<sup>25</sup> 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."

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 演習

### 復習問題

1. Erlang システム内での過負荷のよくある原因を挙げてください
2. 過負荷の対応策の主な戦略を 2 つ答えてください
3. 長時間稼働する処理はどのようにして安全にできますか
4. 同期処理を行うとき、タイムアウトはどのように設定しますか
5. タイムアウト以外の代替案は为什么呢
6. どんなときにスタックバッファの前にキューバッファを選びますか

### 自由回答問題

1. **真のボトルネック**とは为什么呢。それはどうやったら見つけられるでしょうか。
2. サードパーティの API を呼び出すアプリケーションでは、応答時間は他のサーバーの正常性によって大きく変化します。同じサービスへの他の並行呼び出しをブロックすることにより時折発生する遅いリクエストを予防するようにするにはシステムをどう設計すればよいでしょうか。
3. データがスタックバッファ内にバックアップされた過負荷のレイテンシに敏感なサービスに対して新しいリクエストを送ったら、そのリクエストには何が起きるでしょうか。
4. 部分的に送信停止する過負荷機構をバックプレッシャーも提供できるような機構に変えるにはどうしたらいいでしょうか
5. バックプレッシャー機構を部分的に送信停止する機構に変えるにはどうしたらいいでしょうか。
6. ユーザーにとって、リクエストをドロップしたりブロックするときにどういうリスクがあるでしょうか。メッセージの重複やメッセージの喪失をどうやって防げばよいでしょうか。
7. API 設計をする際に過負荷対策を忘れてしまっていたあとに、急にバックプレッシャーや部分的な送信停止を追加する必要が出てきたら、設計にどういう影響があるでしょうか。

## 第II部

# Diagnosing Applications



## 第 4 章

### リモートノードへの接続

旧来、稼働中のサーバープログラムとやり取りする方法は二つあります。一つは、インタラクティブシェルを利用可能にした `screen` や `tmux` のセッションをバックグラウンドに残しておき、それに接続することです。もう一つは、管理関数や、動的にリロードする一括設定ファイルを実装する方法です。

インタラクティブセッションを利用する方法は通常、ソフトウェアが厳密な REPL(Read-Eval-Print-Loop, 対話型評価環境) で動作している限りは問題のない方法です。プログラムによる管理・設定を行う方法では、あなたが必要と思うどんなタスクであっても慎重に計画し、またそれを正しく理解している必要があります。ほとんどすべてのシステムで後者の方法は試されていますので、説明は飛ばすことにします。私がもっと興味があるのは、事態が既に悪化していて、その事態に対処する関数が存在しない場合です。

Erlang は REPL というより”インタラクタ”に近いものを使っています。基本的には、Erlang 仮想マシンは REPL を必要とせず、バイトコードがうまく具合に動作するよう専念すれば良いのです。シェルの必要はありませんね。しかしながら、同時実行とマルチプロセッシングを可能にする仕組みや素晴らしい分散サポートのおかげで、Erlang では任意の Erlang プロセスで動作するソフトウェア内包の REPL を利用できます。

これはつまり、一つの `screen` セッションと一つのシェルを使うのとは異なり、たくさんの Erlang シェルを好きなだけ同時につかって、一つの仮想マシンとのやり取りを行えるということです。<sup>1</sup>

この機能のもっとも一般的な利用方法は、接続したい二ノードが持つ `cookie` を利用する方法<sup>2</sup>ですが、`cookie` を利用しない方法もいくつかあります。多くの方法では名前付きノードが必要で、どんな方法でも **アプリアリな** 接続確認手段は必要です。

<sup>1</sup> このメカニズムの詳細は <http://ferd.ca/repl-a-bit-more-and-less-than-that.html> 参照

<sup>2</sup> 詳細は <http://learnyousomeerlang.com/distribunomicon#cookies> か [http://www.erlang.org/doc/reference\\_manual/distributed.html#id83619](http://www.erlang.org/doc/reference_manual/distributed.html#id83619) 参照

## 4.1 ジョブ制御モード

ジョブ制御モード (Job Control Mode, JCL mode) は、Erlang シェルで `^G` を押した時に得られるメニューのことです。このメニューの中に、リモートシェルの接続を許可するオプションがあります。

---

```
(somenode@ferdmbp.local)1>
User switch command
--> h
  c [nn]          - connect to job
  i [nn]          - interrupt job
  k [nn]          - kill job
  j              - list all jobs
  s [shell]       - start local shell
  r [node [shell]] - start remote shell
  q              - quit erlang
  ? | h          - this message
--> r 'server@ferdmbp.local'
--> c
Eshell Vx.x.x (abort with ^G)
(server@ferdmbp.local)1>
```

---

こうすると、ジョブ管理や行編集はローカルシェルにより実行されますが、実際の評価はリモートで行われます。このリモートシェルが評価を行った結果、出力はすべて、ローカルシェルに転送されます。

シェルを終了するには `^G` を再び押して、ジョブ制御モードに戻ってください。先程も行ったように、ジョブ制御はローカルで行われますから、`^G q` により終了させても安全です。

---

```
(server@ferdmbp.local)1>
User switch command
--> q
```

---

全クラスタに自動的に接続することがないように、最初のシェルを hidden モード (これには引数 `-hidden` が使えます) にしても良いですね。

## 4.2 Remsh

呼び出し方は異なりますが、ジョブ制御モードを通じた方法にきわめて似た仕組みがあります。ジョブ制御モードの一連の手順は、ロングネームを使う場合、以下のようにシェルを起動することで省略することができます。

---

```
1 erl -name local@domain.name -remsh remote@domain.name
```

---

ショートネームを使う場合は、以下のようになります。

---

```
1 erl -sname local@domain -remsh remote@domain
```

---

その他の引数 (例えば `-hidden` や `-setcookie $COOKIE`) も利用可能です。下で動いている仕組みはジョブ制御モードの時と同じですが、最初のシェルはローカルではなくリモートで立ち上がります (ジョブ制御では依然としてローカルです)。リモートシェルから抜ける際にも最も安全な方法は先ほどと変わらず、`^G` です。

## 4.3 SSH デーモン

Erlang/OTP には SSH の実装が標準で備わっており、サーバーとクライアントのどちらの方式でも動作します。SSH の実装の一部として用意されているデモアプリケーションは、Erlang で動作するリモートシェルを提供します。

これを起動するには普通、SSH リモート接続を行うためにあなたの鍵を所定の位置に配置しなければなりませんが、簡単なテスト目的なら、以下のようにして実行することができます。

---

```
$ mkdir /tmp/ssh
$ ssh-keygen -t rsa -f /tmp/ssh/ssh_host_rsa_key
$ ssh-keygen -t rsa1 -f /tmp/ssh/ssh_host_key
$ ssh-keygen -t dsa -f /tmp/ssh/ssh_host_dsa_key
$ erl
1> application:ensure_all_started(ssh).
{ok,[crypto,asn1,public_key,ssh]}
2> ssh:daemon(8989, [{system_dir, "/tmp/ssh"},
2>                  {user_dir, "/home/ferd/.ssh"}]).
{ok,<0.52.0>}
```

---

ここではほんの少しオプションを設定しただけです。`system_dir` はホストファイルの置き場、`user_dir` は SSH 設定ファイルの置き場を設定しています。この他にも多くのオプションがあり、特定のパスワードを許可したり、公開鍵の取り扱いを変えたりすることが可能です<sup>3</sup>。

デーモンに接続するには、どんな SSH クライアントでも、以下のようにします。

---

```
$ ssh -p 8989 ferd@127.0.0.1
Eshell Vx.x.x (abort with ^G)
1>
```

---

こうすれば、Erlang を手元のマシンでインストールしていなくても、Erlang を利用できますね。シェルから立ち去るときには、(ターミナルを閉じて) SSH セッションを切れば十分です。`q()` や `init:stop()` のような関数は**実行しない**でください。リモートホストが終了してしまいます<sup>4</sup>。

もし接続する際に問題が発生したら、`ssh` にオプションとして `-oLogLevel=DEBUG` をつければ、デバッグ出力が見られます。

## 4.4 名前付きパイプ

あまり知られてはいませんが、分散 Erlang を明示的に用いずに Erlang ノードに接続する方法として、名前付きパイプを利用するものがあります。これは Erlang を `run_erl` で起動すれば実現できます。`run_erl` が名前付きパイプの中に Erlang をラップしてくれます<sup>5</sup>。

---

```
$ run_erl /tmp/erl_pipe /tmp/log_dir "erl"
```

---

第一引数は名前付きパイプとして振る舞うファイル名です。第二引数はログの保存場所になります<sup>6</sup>。

ノードに接続するには、`to_erl` プログラムを使います。

---

```
$ to_erl /tmp/erl_pipe
Attaching to /tmp/erl_pipe (^D to exit)

1>
```

---

<sup>3</sup> ちゃんとした SSH リモート接続を行うには、詳細な手順と全オプションの説明 <http://www.erlang.org/doc/man/ssh.html#daemon-3> を参照してください。

<sup>4</sup> これは、どんな方法を使うにせよ、リモートの Erlang ノードとやり取りする際には共通です。

<sup>5</sup> `"erl"` が実行されるコマンドです。引数はその後に追加できます。例えば `"erl +K true"` でカーネルポーリングが有効になります。

<sup>6</sup> この方法を使うと出力のたびに `fsync` が呼びだされますので、標準出力を介して多くの入出力が発生する場合は、性能にかなり影響するかもしれません。

---

シェルに接続できました。標準入出力を閉じる (コマンドは`^D`) ことで、シェルを実行させたまま切断ができます。

## 4.5 演習

### 復習問題

1. リモートノードに接続する 4 つの方法はなんですか？
2. 名前のないノードに接続することはできますか？
3. ジョブ制御モードに入るコマンドはなんですか？
4. 標準出力に多くのデータを吐き出すシステムの場合、リモートシェルの接続方法で避けるべき方法はなんですか？
5. リモート接続をした際に、`^G` で切断してはいけない場合とはどんな時ですか？
6. セッションを切断するときに決して行ってはいけないコマンドはなんですか？
7. この章で説明した方法はすべて、複数ユーザーが同じ Erlang ノードに問題なく接続できるような方法でしょうか？

## 第 5 章

# ランタイムメトリクス

実運用を考えた際、Erlang VM で最も良いセールスポイントの一つとして、ありとあらゆる内部調査やデバッグ、プロファイリング、実行時分析の透過性が挙げられます。

プログラム上で取得できるランタイムメトリクスがある利点として、これらメトリクスに依存したツールを作ることも簡単ですし、何らかのタスクや監視を自動化するのも同じように単純です<sup>1</sup>。それに、必要であれば、ツールを介さずに VM から直接情報を受け取ることも可能です。

システムを健全に保ちつつ成長させる実用的な方法は、すべての角度から一大域的にも、局所的にも一監視できるようにしておくことです。将来起きることが通常の挙動なのか否かを先んじて知らせる一般的な方法はありません。

あなたのシステムが通常の状態下でどのように見えているのか。考えを形にするには、多くのデータを保持して、それらをことあるごとに観察したくなるでしょう。何かがうまく行かなくなったその日、あなたがこれまで培ってきたすべての観察方法を使えば、どこが不調で何を修正すべきかを簡単に見つけだせます。

この章（また、このあとの章のほとんど）では、紹介する概念や機能のほとんどは、標準ライブラリ—正規 OTP ディストリビューションの一部—に含まれるコードからアクセス可能です。

しかしながら、これら機能は一箇所にまとまっているわけではありませんし、システムの中で自らの足を撃ちぬくことも非常に簡単にできてしまいます。これらは便利ツールというより、基本的な構成要素に近いものにもなりがちです。

したがって、本書をより軽く、利用しやすくするために、よく使う操作は、実運用で利用しても安全な **recon**<sup>2</sup> ライブラリにまとめ直されています。

<sup>1</sup> 自動化プロセスが何かしらは正措置を行おうとして、暴走したり、やり過ぎたりしないか保証するほうが、より一層複雑です

<sup>2</sup> <http://ferd.github.io/recon/>

## 5.1 グローバルビュー

大域的に VM を見るなら、どんなコードが動いているかはさておき、VM の一般的なメトリクスの統計情報を監視すると役に立ちます。更に言うなら、各メトリクスを長期的に見るソリューションをめざすべきです — 一部の問題は幾週にも渡る非常に長い蓄積で発生しますし、これを短期間の情報表示画面から検知するのは不可能です。

長期的に見ることで問題が明るみになる良い例にはメモリやプロセスのリークがありますが、それだけでなく、一日もしくは一週間のうちの特定時刻に関連する活動の中で、周期的または不定期に発生するスパイクも良い例です。確証を持つためには、何ヶ月ものデータがしばしば必要になります。

このようなケースにおいて、既存の Erlang メトリクス アプリケーションは有用です。一般的な選択肢としては、以下のようなものがあります。

- `folsom`<sup>3</sup>。メトリクスを VM 内のメモリに保存する。グローバル領域、アプリケーション領域のどちらも可能。
- `vmstats`<sup>4</sup> と `statsderl`<sup>5</sup>。 `statsd`<sup>6</sup> を通じてノードメトリクスを graphite に送る。
- `exometer`<sup>7</sup>。気取ったメトリクスシステムで、(とりわけ) `folsom` と統合できる他、多くのバックエンド (graphite、collectd、`statsd`、Riak、SNMP など) と統合できます。この島では新参者ですね。
- `ehmon`<sup>8</sup> は直接標準出力に書き出すときに使います。出力結果は専用のエージェントなり、splunk などで取得します。
- お手製のカスタムソリューション。一般的には、ETS テーブルと定期的にデータをダンプするプロセスを使います。<sup>9</sup>
- もし何も用意がない状態で障害になったら、シェルループで諸々の出力を行う関数<sup>10</sup>。

これらを少し調べてみて、どれか一つを使い、あなたの興味あるメトリクスを刻々と見せてくれる永続レイヤーを用意するのが、一般的には良い方法です。

---

<sup>3</sup> <https://github.com/boundary/folsom>

<sup>4</sup> <https://github.com/ferd/vmstats>

<sup>5</sup> <https://github.com/lpgauth/statsderl>

<sup>6</sup> <https://github.com/etsy/statsd/>

<sup>7</sup> <https://github.com/Feuerlabs/exometer>

<sup>8</sup> <https://github.com/heroku/ehmon>

<sup>9</sup> よくあるパターンでは、`ectr` アプリケーションが適しています。<https://github.com/heroku/ectr> を参照してください。

<sup>10</sup> あなたがピンチのときにも諸々の出力ができるよう、`recon` アプリケーションは関数 `recon:node_stats_print/2` を備えています

### 5.1.1 メモリ

大抵のツールで Erlang VM から受けている memory の情報は、`erlang:memory()` から取られる変数です。

---

```
1> erlang:memory().
[{total,13772400},
 {processes,4390232},
 {processes_used,4390112},
 {system,9382168},
 {atom,194289},
 {atom_used,173419},
 {binary,979264},
 {code,4026603},
 {ets,305920}]
```

---

これは説明が必要ですね。

まず第一に、すべての返り値は byte 単位の値となっていて、メモリの**割り当てられた量** (Erlang VM が実際に使用しているメモリ量のことで、OS が Erlang VM のために確保した量ではありません) を表します。遅かれ早かれ、OS が示す値よりはるかに少ない値だとわかるはずです。

`total` の項には、`processes` と `system` で利用してるメモリ量の和が入ります (VM を備えない限り、完璧な和にはなりません!)。`processes` の項は、Erlang プロセスと、スタックやヒープとして使用されるメモリ量です。`system` の項は残りです。つまり、ETS テーブルや、VM の中で使われるアトムや、参照カウントされたバイナリ<sup>11</sup>、失われた隠しデータなどです。

仮想マシンが保有する総メモリ量を知りたいければ、システム限界 (`ulimit`) を突破する量と同じで、VM 内部から取得するのはより困難です。もし `top` や `htop` を使うことなしにこのデータが欲しければ、あなたは VM のメモリアロケータを深堀して見つけ出すしかありません<sup>12</sup>。

運が良いことに、この値を取るために `recon` は関数 `recon_alloc:memory/1` を用意しています。引数には以下のようなものがあります。

- `used`。使用中の Erlang データが割り当てられているメモリの総量
- `allocated`。VM により予約されているメモリ量。使用中のメモリ量だけでなく、OS から与えられた未使用な領域も含む。あなたが `ulimit` や OS の示す値を取り扱うなら、この値がほしい値でしょう。
- `unused`。VM により予約されているがまだ割り当てられていない容量。`allocated - used`

---

<sup>11</sup> 詳しくは 7.2 節

<sup>12</sup> 詳しくは 7.3.2 節を見てください



と等価です。

- `usage`。割当てられたメモリ量に対する使用中のメモリ量の割合 (0.0 から 1.0)。

利用可能なオプションは他にもありますが、7 の章でメモリリークの調査を行うにはこのくらいで十分でしょう。

### 5.1.2 CPU

Erlang 開発者にとって残念なことに、CPU の統計データを取ることは非常に困難です。それには少し理由があります。

- VM はスケジューリングの際にプロセスとは無関係のタスクをたくさん行っています。高度なスケジューリングタスクと Erlang プロセスによる大量のタスクは、特徴付けすることが難しいのです。
- VM は内部で **リダクション** に基づくモデルを使っています。このモデルは任意の数のタスクで表せます。すべての (BIF を含む) 関数呼び出しでプロセスリダクション数がインクリメントされ、所与のリダクション数に到達すると、プロセスが未スケジュール状態にもどります。
- 負荷の低いタスクがすぐスリープ状態にならないように、Erlang スケジューラの扱うスレッドはしばらくビジーウェイト状態になります。これは、突然負荷が上がるケースでも遅延が大きくなり過ぎないための配慮です。この値を変更するには、VM フラグ (+sbtw none|very\_short|short|medium|long|very\_long) が使えます。

これらの要素が組み合わさっているため、実行中の Erlang コードによる CPU 使用量をしっかりと測る方法を見つけるのは大変です。Erlang ノードは実運用において、ほとんどのタスクでも CPU を大量に使用するように見えますが、負荷が上がっても、大量のタスクを残りの CPU リソースでまかなえることがよくあります。

このデータを最も適切に表現するのは、スケジューラの総経過時間です。これは標準では無効のメトリクスで、ノードごとに手動で有効に設定すれば、一定間隔で取得されるようになります。取得される値からわかるのは、スケジューラがプロセス、普通の Erlang コード、NIF、BIF、ガベージコレクションなどを走らせる時間と、スケジューラがプロセススケジュールを試みているか、アイドル状態になっている時間の割合です。

これは CPU 使用率というより、どちらかといえば**スケジューラ使用率**を表すものです。高い値であるほど、高い負荷がかかっていることを表します。

基本的な使い方は Erlang/OTP のリファレンスマニュアル<sup>13</sup>に記載されていますが、`recon` を使ってもこの値は取得できます。

---

<sup>13</sup> [http://www.erlang.org/doc/man/erlang.html#statistics\\_scheduler\\_wall\\_time](http://www.erlang.org/doc/man/erlang.html#statistics_scheduler_wall_time)

---

```
1> recon:scheduler_usage(1000).
[{1,0.9919596133421669},
 {2,0.9369579039389054},
 {3,1.9294092120138725e-5},
 {4,1.2087551402238991e-5}]
```

---

関数 `recon:scheduler_usage(N)` は  $N$  ミリ秒 (ここでは 1 秒) 調査を行い、各スケジューラの値を出力します。今回の場合は、VM は高負荷なスケジューラが 2 つ (それぞれ 99.2% と 93.7%) と、1% にも満たないくらいほとんど使われていないスケジューラが 2 つありますね。しかし、`htop` などのツールは CPU コアごとにこれと似たような値を出力してくれました。

---

1	[	70.4%]
2	[	20.6%]
3	[	100.0%]
4	[	40.2%]

---

結果として、(スケジューラが実行タスクを選択する状態でなく、ビジーウェイト状態だと想定すると) Erlang のタスクスケジューリングでほとんど使っていない CPU 領域があっても、OS からは高い使用率を表示されます。

もうひとつの面白い挙動として、スケジューラは OS が示す値よりも高い割合 (1.0) を表示することもあります。スケジューラが OS リソースを待っている場合、それ以上のタスクを扱えないことから使用状態と認識される場合がその一例です。他にも、OS 自身が CPU 以外のタスクで詰まっている場合にも、Erlang スケジューラも仕事を行えないことから、1.0 の割合が表示されることになります。

これらのふるまいに関する考察は容量計画を行う際には特に大事で、CPU 使用率や負荷を見る以上に経営判断の良い指標になるでしょう。

### 5.1.3 プロセス

プロセスに関するグローバルビューがあると、**タスク**の観点で VM はがどのくらいの仕事を完了したか知るのに役立ちます。Erlang において一般的に良い方法となるのは、真に同時実行されているプロセスを用いることです — Web サーバでは普段、1 リクエストもしくは 1 接続ごとに 1 プロセスを持つでしょうし、状態を持つシステムであれば、ユーザごとに 1 つのプロセスを追加しているかもしれません — したがって、1 ノード中に存在するプロセス数がメトリクスとして利用できます。

5.1 の章で説明するツールのほとんどが、何らかの方法でプロセス数を監視していますが、自前

でプロセスカウントが必要であれば、以下のようにすれば十分です。

---

```
1> length(processes()).  
56535
```

---

この値を刻々と監視することは、どんなメトリクスよりも、負荷を特徴づけたり、プロセスリークを発見するのに非常に役立ちます。

#### 5.1.4 ポート

プロセスと似たような理由で、**ポート**も監視すべきです。ポートはありとあらゆる種類のコネクションや、外部に開いているソケット — TCP ソケット、UDP ソケット、SCTP ソケット、ファイルディスクリプタなど — を包含するデータ型です。

これをカウントする一般的な関数として、`length(erlang:ports())` があります (この関数を用意されているところも、プロセスの時の話と似ていますね)。しかしながら、この関数では全種類のポートをひとつのエンティティにまとめてしまっています。ポートの種類ごとに値を取りたいければ、この関数の代わりに `recon` が使えます。

---

```
1> recon:port_types().  
[{"tcp_inet",21480},  
 {"efile",2},  
 {"udp_inet",2},  
 {"0/1",1},  
 {"2/2",1},  
 {"inet_gethost 4 ",1}]
```

---

このリストはポートの種類ごとに、タイプ名と数値を格納しています。タイプ名は Erlang VM 自身によって定義されている文字列です。

普通、すべての\*\_inet ポートはソケットで、接頭語が使用プロトコル (TCP, UDP, SCTP) を表します。efile タイプはファイルのためのもので、"0/1"と"2/2"はそれぞれ、標準入出力チャンネル (`stdin` と `stdout`) と標準エラー出力チャンネル (`stderr`) へのファイルディスクリプタを表します。

その他のタイプのほとんどは、ポートがやり取りしているドライバ名が入ります。例としては `port programs`<sup>14</sup>や `port drivers`<sup>15</sup>などです。

繰り返しになりますが、これらの値を監視することはシステム使用率や負荷を見たり、リークを

---

<sup>14</sup> [http://www.erlang.org/doc/tutorial/c\\_port.html](http://www.erlang.org/doc/tutorial/c_port.html)

<sup>15</sup> [http://www.erlang.org/doc/tutorial/c\\_portdriver.html](http://www.erlang.org/doc/tutorial/c_portdriver.html)

見たり、その他いろいろなことに役立ちます。

## 5.2 Digging In

Whenever some 'in the large' view (or logging, maybe) has pointed you towards a potential cause for an issue you're having, it starts being interesting to dig around with a purpose. Is a process in a weird state? Maybe it needs tracing<sup>16</sup>! Tracing is great whenever you have a specific function call or input or output to watch for, but often, before getting there, a lot more digging is required.

Outside of memory leaks, which often need their own specific techniques and are discussed in Chapter 7, the most common tasks are related to processes, and ports (file descriptors and sockets).

### 5.2.1 Processes

By all means, processes are an important part of a running Erlang system. And because they're so central to everything that goes on, there's a lot to want to know about them. Fortunately, the VM makes a lot of information available, some of which is safe to use, and some of which is unsafe to use in production (because they can return data sets large enough that the amount of memory copied to the shell process and used to print it can kill the node).

All the values can be obtained by calling `process_info(Pid, Key)` or `process_info(Pid, [Keys])`<sup>17</sup>. Here are the commonly used keys<sup>18</sup>:

#### Meta

`dictionary` returns all the entries in the process dictionary<sup>19</sup>. Generally safe to use, because people shouldn't be storing gigabytes of arbitrary data in there.

`group_leader` the group leader of a process defines where IO (files, output of `io:format/1-3`) goes.<sup>20</sup>

`registered_name` if the process has a name (as registered with `erlang:register/2`), it is given here.

---

<sup>16</sup> See Chapter 9

<sup>17</sup> In cases where processes contain sensitive information, data can be forced to be kept private by calling `process_flag(sensitive, true)`

<sup>18</sup> For all options, look at [http://www.erlang.org/doc/man/erlang.html#process\\_info-2](http://www.erlang.org/doc/man/erlang.html#process_info-2)

<sup>19</sup> See <http://www.erlang.org/course/advanced.html#dict> and <http://ferd.ca/on-the-use-of-the-process-dictionary-in-erlang.html>

<sup>20</sup> See <http://learnyouesomeerlang.com/building-otp-applications#the-application-behaviour> and [http://erlang.org/doc/apps/stdlib/io\\_protocol.html](http://erlang.org/doc/apps/stdlib/io_protocol.html) for more details.

**status** the nature of the process as seen by the scheduler. The possible values are:

- exiting** the process is done, but not fully cleared yet;
- waiting** the process is waiting in a `receive ... end`;
- running** self-descriptive;
- runnable** ready to run, but not scheduled yet because another process is running;
- garbage\_collecting** self-descriptive;
- suspended** whenever it is suspended by a BIF, or as a back-pressure mechanism because a socket or port buffer is full. The process only becomes runnable again once the port is no longer busy.

## Signals

**links** will show a list of all the links a process has towards other processes and also ports (sockets, file descriptors). Generally safe to call, but to be used with care on large supervisors that may return thousands and thousands of entries.

**monitored\_by** gives a list of processes that are monitoring the current process (through the use of `erlang:monitor/2`).

**monitors** kind of the opposite of **monitored\_by**; it gives a list of all the processes being monitored by the one polled here.

**trap\_exit** has the value `true` if the process is trapping exits, `false` otherwise.

## Location

**current\_function** displays the current running function, as a tuple of the form `{Mod, Fun, Arity}`.

**current\_location** displays the current location within a module, as a tuple of the form `{Mod, Fun, Arity, [{File, FileName}, {line, Num}]}`.

**current\_stacktrace** more verbose form of the preceding option; displays the current stacktrace as a list of 'current locations'.

**initial\_call** shows the function that the process was running when spawned, of the form `{Mod, Fun, Arity}`. This may help identify what the process was spawned as, rather than what it's running right now.

## Memory Used

**binary** Displays the all the references to refc binaries<sup>21</sup> along with their size. Can be unsafe to use if a process has a lot of them allocated.

**garbage\_collection** contains information regarding garbage collection in the process. The content is documented as 'subject to change' and should be treated as such. The information tends to contains entries such as the number of garbage collections

---

<sup>21</sup> See Section 7.2

the process has went through, options for full-sweep garbage collections, and heap sizes.

**heap\_size** A typical Erlang process contains an 'old' heap and a 'new' heap, and goes through generational garbage collection. This entry shows the process' heap size for the newest generation, and it usually includes the stack size. The value returned is in **words**.

**memory** Returns, in **bytes**, the size of the process, including the call stack, the heaps, and internal structures used by the VM that are part of a process.

**message\_queue\_len** Tells you how many messages are waiting in the mailbox of a process.

**messages** Returns all of the messages in a process' mailbox. This attribute is **extremely** dangerous to request in production because mailboxes can hold millions of messages if you're debugging a process that managed to get locked up. **Always** call for the **message\_queue\_len** first to make sure it's safe to use.

**total\_heap\_size** Similar to **heap\_size**, but also contains all other fragments of the heap, including the old one. The value returned is in **words**.

## Work

**reductions** The Erlang VM does scheduling based on **reductions**, an arbitrary unit of work that allows rather portable implementations of scheduling (time-based scheduling is usually hard to make work efficiently on as many OSes as Erlang runs on). The higher the reductions, the more work, in terms of CPU and function calls, a process is doing.

Fortunately, for all the common ones that are also safe, recon contains the **recon:info/1** function to help:

---

```
1> recon:info("<0.12.0>").
[{meta,[{registered_name,rex},
        {dictionary,[{'$ancestors',[kernel_sup,<0.10.0>]},
                      {'$initial_call',{rpc,init,1}}]},
        {group_leader,<0.9.0>},
        {status,waiting}}],
 {signals,[{links,[<0.11.0>]},
            {monitors,[]},
            {monitored_by,[]},
            {trap_exit,true}}],
 {location,[{initial_call,{proc_lib,init_p,5}},
             {current_stacktrace,[{gen_server,loop,6,
```

```

                                [{file,"gen_server.erl"},{line,358}]],
                                {proc_lib,init_p_do_apply,3,
                                [{file,"proc_lib.erl"},{line,239}]]}}]],
{memory_used,[{memory,2808},
               {message_queue_len,0},
               {heap_size,233},
               {total_heap_size,233},
               {garbage_collection,[{min_bin_vheap_size,46422},
                                    {min_heap_size,233},
                                    {fullsweep_after,65535},
                                    {minor_gcs,0}]]}],
{work,[{reductions,35}]]]

```

---

For the sake of convenience, `recon:info/1` will accept any pid-like first argument and handle it: literal pids, strings ("`<0.12.0>`"), registered atoms, global names (`{global, Atom}`), names registered with a third-party registry (e.g. with `gproc: {via, gproc, Name}`), or tuples (`{0,12,0}`). The process just needs to be local to the node you're debugging.

If only a category of information is wanted, the category can be used directly:

---

```

2> recon:info(self(), work).
{work,[{reductions,11035}]}

```

---

or can be used in exactly the same way as `process_info/2`:

---

```

3> recon:info(self(), [memory, status]).
[{memory,10600},{status,running}]

```

---

This latter form can be used to fetch unsafe information.

With all this data, it's possible to find out all we need to debug a system. The challenge then is often to figure out, between this per-process data, and the global one, which process(es) should be targeted.

When looking for high memory usage, for example it's interesting to be able to list all of a node's processes and find the top N consumers. Using the attributes above and the `recon:proc_count(Attribute, N)` function, we can get these results:

---

```

4> recon:proc_count(memory, 3).
[<0.26.0>,831448,
 [{current_function,{group,server_loop,3}},

```

```

    {initial_call,{group,server,3}}}],
{<0.25.0>,372440,
 [user,
  {current_function,{group,server_loop,3}},
  {initial_call,{group,server,3}}]},
{<0.20.0>,372312,
 [code_server,
  {current_function,{code_server,loop,1}},
  {initial_call,{erlang,apply,2}}]}}]

```

---

Any of the attributes mentioned earlier can work, and for nodes with long-lived processes that can cause problems, it's a fairly useful function.

There is however a problem when most processes are short-lived, usually too short to inspect through other tools, or when a moving window is what we need (for example, what processes are busy accumulating memory or running code **right now**).

For this use case, Recon has the `recon:proc_window(Attribute, Num, Milliseconds)` function.

It is important to see this function as a snapshot over a sliding window. A program's timeline during sampling might look like this:

```
--w---- [Sample1] ---x-----y----- [Sample2] ---z-->
```

The function will take two samples at an interval defined by `Milliseconds`.

Some processes will live between `w` and die at `x`, some between `y` and `z`, and some between `x` and `y`. These samples will not be too significant as they're incomplete.

If the majority of your processes run between a time interval `x` to `y` (in absolute terms), you should make sure that your sampling time is smaller than this so that for many processes, their lifetime spans the equivalent of `w` and `z`. Not doing this can skew the results: long-lived processes that have 10 times the time to accumulate data (say reductions) will look like huge consumers when they're not one.<sup>22</sup>

The function, once running gives results like follows:

---

```

5> recon:proc_window(reductions, 3, 500).
[<0.46.0>,51728,
 [{current_function,{queue,in,2}},
  {initial_call,{erlang,apply,2}}]}}]

```

---

<sup>22</sup> Warning: this function depends on data gathered at two snapshots, and then building a dictionary with entries to differentiate them. This can take a heavy toll on memory when you have many tens of thousands of processes, and a little bit of time.



```
{<0.49.0>,5728,  
  [{current_function,{dict,new,0}},  
   {initial_call,{erlang,apply,2}}]},  
{<0.43.0>,650,  
  [{current_function,{timer,sleep,1}},  
   {initial_call,{erlang,apply,2}}]}}
```

---

With these two functions, it becomes possible to hone in on a specific process that is causing issues or misbehaving.

### 5.2.2 OTP Processes

When processes in question are OTP processes (most of the processes in a production system should definitely be OTP processes), you instantly win more tools to inspect them.

In general the `sys` module<sup>23</sup> is what you want to look into. Read the documentation on it and you'll discover why it's so useful. It contains the following features for any OTP process:

- logging of all messages and state transitions, both to the shell or to a file, or even in an internal buffer to be queried;
- statistics (reductions, message counts, time, and so on);
- fetching the status of a process (metadata including the state);
- fetching the state of a process (as in the `#state{}` record);
- replacing that state
- custom debugging functions to be used as callbacks

It also provides functionality to suspend or resume process execution.

I won't go into a lot of details about these functions, but be aware that they exist.

### 5.2.3 Ports

Similarly to processes, Erlang ports allow a lot of introspection. The info can be accessed by calling `erlang:port_info(Port, Key)`, and more info is available through the `inet` module. Most of it has been regrouped by the `recon:port_info/1-2` functions, which work using a somewhat similar interface to their process-related counterparts.

#### Meta

`id` internal index of a port. Of no particular use except to differentiate ports.

---

<sup>23</sup> <http://www.erlang.org/doc/man/sys.html>

**name** type of the port — with names such as "tcp\_inet", "udp\_inet", or "efile", for example.

**os\_pid** if the port is not an inet socket, but rather represents an external process or program, this value contains the os pid related to the said external program.

### Signals

**connected** Each port has a controlling process in charge of it, and this process' pid is the **connected** one.

**links** ports can be linked with processes, much like other processes can be. The list of linked processes is contained here. Unless the process has been owned by or manually linked to a lot of processes, this should be safe to use.

**monitors** ports that represent external programs can have these programs end up monitoring Erlang processes. These processes are listed here.

### IO

**input** the number of bytes read from the port.

**output** the number of bytes written to the port.

### Memory Used

**memory** this is the memory (in bytes) allocated by the runtime system for the port.

This number tends to be small-ish and excludes space allocated by the port itself.

**queue\_size** Port programs have a specific queue, called the driver queue<sup>24</sup>. This return the size of this queue, in bytes.

### Type-Specific

**Inet Ports** Returns inet-specific data, including statistics<sup>25</sup>, the local address and port number for the socket (**sockname**), and the inet options used<sup>26</sup>

**Others** currently no other form than inet ports are supported in recon, and an empty list is returned.

The list can be obtained as follows:

---

```
1> recon:port_info("#Port<0.818>").
[{meta, [{id, 6544}, {name, "tcp_inet"}, {os_pid, undefined}]},
 {signals, [{connected, <0.56.0>},
            {links, [<0.56.0>]}],
```

---

<sup>24</sup> The driver queue is available to queue output from the emulator to the driver (data from the driver to the emulator is queued by the emulator in normal Erlang message queues). This can be useful if the driver has to wait for slow devices etc, and wants to yield back to the emulator.

<sup>25</sup> <http://www.erlang.org/doc/man/inet.html#getstat-1>

<sup>26</sup> <http://www.erlang.org/doc/man/inet.html#setopts-2>

```

        {monitors, []}},
{io, [{input, 0}, {output, 0}]},
{memory_used, [{memory, 40}, {queue_size, 0}]},
{type, [{statistics, [{recv_oct, 0},
                      {recv_cnt, 0},
                      {recv_max, 0},
                      {recv_avg, 0},
                      {recv_dvi, ...},
                      {...}|...]}],
        {peername, {{50, 19, 218, 110}, 80}},
        {sockname, {{97, 107, 140, 172}, 39337}},
        {options, [{active, true},
                   {broadcast, false},
                   {buffer, 1460},
                   {delay_send, ...},
                   {...}|...}]}}]

```

---

On top of this, functions to find out specific problematic ports exist the way they do for processes. The gotcha is that so far, recon only supports them for inet ports and with restricted attributes: the number of octets (bytes) sent, received, or both (`send_oct`, `recv_oct`, `oct`, respectively), or the number of packets sent, received, or both (`send_cnt`, `recv_cnt`, `cnt`, respectively).

So for the cumulative total, which can help find out who is slowly but surely eating up all your bandwidth:

---

```

2> recon:inet_count(oct, 3).
[#{#Port<0.6821166>,15828716661,
  [{recv_oct,15828716661},{send_oct,0}]},
 #{#Port<0.6757848>,15762095249,
  [{recv_oct,15762095249},{send_oct,0}]},
 #{#Port<0.6718690>,15630954707,
  [{recv_oct,15630954707},{send_oct,0}]}]

```

---

Which suggest some ports are doing only input and eating lots of bytes. You can then use `recon:port_info("#Port<0.6821166>")` to dig in and find who owns that socket, and what is going on with it.

Or in any other case, we can look at what is sending the most data within any time window<sup>27</sup> with the `recon:inet_window(Attribute, Count, Milliseconds)` function:

---

<sup>27</sup> See the explanations for the `recon:proc_window/3` in the preceding subsection

---

```
3> recon:inet_window(send_oct, 3, 5000).  
[{{#Port<0.11976746>,2986216, [{send_oct,4421857688}]},  
  {{#Port<0.11704865>,1881957, [{send_oct,1476456967}]},  
  {{#Port<0.12518151>,1214051, [{send_oct,600070031}]}}]
```

---

For this one, the value in the middle of the tuple is what `send_oct` was worth (or any chosen attribute for each call) during the specific time interval chosen (5 seconds here).

There is still some manual work involved into properly linking a misbehaving port to a process (and then possibly to a specific user or customer), but all the tools are in place.

## 5.3 演習

### 復習問題

1. Erlang のメモリではどのような値が報告されますか。
2. グローバルビュー向けのプロセス関連で有益なメトリクスはなんでしょう。
3. ポートとはどんなもので、グローバルではどのように監視されているでしょう。
4. Erlang システムにおいてなぜ `top` や `htop` が信頼できないのでしょうか。代替手段はなんでしょう。
5. プロセス用に得られる 2 種類のシグナル関係の情報を挙げてください。
6. 特定のプロセスがどのコードを動かしているかをどうやって見つけられるでしょう。
7. 特定のプロセスのメモリ関連の情報にはどのような種類があるでしょう。
8. プロセスが大量の処理をしているかどうかをどうやって見極めますか。
9. 本番システム内のプロセスを調査する際に取得すると危険な値を 2 つ、3 つ挙げてください。
10. `sys` モジュール経由で OTP プロセスに提供されるいくつかの機能はなんでしょう。
11. `inet` ポートを調査しているときに得られる値にはどんなものがあるでしょう。
12. ポートの種類（ファイル、TCP、UDP）はどのように見つけられるでしょう。

### 自由回答問題

1. グローバルメトリクス内で利用できる長時間のウィンドウが欲しくなる理由はなんでしょう。
2. 次の問題を見つげるときに使うべき関数は `recon:proc_count/2` と `recon:proc_window/3` のどちらでしょう。
3. (a) リダクション  
(b) メッセージキューの長さ

(c) メモリ

4. あるプロセスのスーパーバイザーがどれかに関する情報はどうやって見つけますか。
5. `recon:inet_count/2` や `recon:inet_window/3` はそれぞれいつ使うべきでしょう。
6. OS から報告されたメモリと Erlang の `memory` 関数から報告されるメモリの違いを説明してください。
7. ときどき Erlang が実際にはさほど稼働していないのに、非常に稼働率が上がっているように見えるのはなぜでしょうか。
8. あるノード上のプロセスの何割が実行可能だがすぐにはスケジュールできない状況になっているかを調べる方法を教えてください。

## ハンズオン

次のコードを使って回答してください。 [https://github.com/ferd/recon\\_demo](https://github.com/ferd/recon_demo):

1. システムメモリとはなんですか。
2. ノードは多くの CPU リソースを使っていますか。
3. メールボックスが溢れているプロセスはありますか。
4. どの chatty プロセス (`council_member`) が一番メモリを使っていますか。
5. どの chatty プロセスが一番 CPU を使用していますか。
6. どの chatty プロセスが一番帯域を消費していますか。
7. どの chatty プロセスが TCP で一番メッセージを送っていますか。また一番メッセージを送っていないものも答えてください。
8. ノード上のあるプロセスが複数の接続やファイルディスクリプタを同時に保持しがちかどうか、どのように判断しますか。
9. 今現在あるノード上でほとんどのプロセスから同時に呼ばれている関数を見つけられますか。

## 第 6 章

# クラッシュダンプを読む

Erlang ノードはクラッシュすると、クラッシュダンプを出力します。<sup>1</sup>

フォーマットについては Erlang のオフィシャルドキュメントにほとんど書かれており<sup>2</sup>、深く掘り下げたい人は誰でもそのドキュメントを見ることで、データが意味していることを把握することができます。特定のデータについては更に VM の一部についても理解していないと理解することが難しいものがありますが、このドキュメントに載せるには複雑過ぎます。

クラッシュダンプはデフォルトで `erl_crash.dump` という名前で、Erlang プロセスが動いている場所へ出力されます。この挙動 (ファイル名も含めて) は `ERL_CRASH_DUMP` 環境変数<sup>3</sup>を指定することで上書きできます。

### 6.1 一般的な見方

クラッシュダンプを読むことは**事後**にノードが死んだ可能性のある理由を理解するのに役立つことがあります。それらを素早く見るために `recon` の `erl_crashdump_analyzer.sh`<sup>4</sup>を使う方法があり、それをクラッシュダンプに対して実行します。

```
$ ./recon/script/erl_crashdump_analyzer.sh erl_crash.dump
analyzing erl_crash.dump, generated on: Thu Apr 17 18:34:53 2014
```

<sup>1</sup> ダンプ中に `ulimits` の制限を超えて OS に殺されたり、セグメンテーションフォールトしない限りは

<sup>2</sup> [http://www.erlang.org/doc/apps/erts/crash\\_dump.html](http://www.erlang.org/doc/apps/erts/crash_dump.html)

<sup>3</sup> Heroku のルーティングとテレメトリチームは [heroku\\_crashdumps](#) アプリケーションを使ってクラッシュダンプのパスと名前を設定しています。これをプロジェクトに追加して、起動時にダンプの名前をつけ、それらを事前に指定した場所に置くことができます。

<sup>4</sup> [https://github.com/ferd/recon/blob/master/script/erl\\_crashdump\\_analyzer.sh](https://github.com/ferd/recon/blob/master/script/erl_crashdump_analyzer.sh)

Slogan: eheap\_alloc: Cannot allocate 2733560184 bytes of memory  
(of type "old\_heap").

Memory:

===

processes: 2912 Mb  
processes\_used: 2912 Mb  
system: 8167 Mb  
atom: 0 Mb  
atom\_used: 0 Mb  
binary: 3243 Mb  
code: 11 Mb  
ets: 4755 Mb

---

total: 11079 Mb

Different message queue lengths (5 largest different):

===

1 5010932  
2 159  
5 158  
49 157  
4 156

Error logger queue length:

===

0

File descriptors open:

===

UDP: 0  
TCP: 19951  
Files: 2  
---  
Total: 19953

Number of processes:

===

36496

Processes Heap+Stack memory sizes (words) used in the VM (5 largest different):

===

1 284745853

1 5157867

1 4298223

2 196650

12 121536

Processes OldHeap memory sizes (words) used in the VM (5 largest different):

===

3 318187

9 196650

14 121536

64 75113

15 46422

Process States when crashing (sum):

===

1 Garbing

74 Scheduled

36421 Waiting

---

このデータダンプはあなたの目の前にある問題を直接指摘してはくれませんが、どこを見れば良いのかの良い手がかりとなります。例えば、ここではこのノードはメモリ不足となりましたが、15GB あるうちの 11079MB を使用していました (我々が使っていた最大のインスタンスサイズだったので私はこれを覚えています!)。これは以下の事柄に対するひとつの症状となり得ます。

- メモリフラグメンテーション;
- C コードまたはドライバーのメモリリーク;



- クラッシュダンプを生成する前にガベージコレクトされてしまった大量のメモリ<sup>5</sup>.

より一般的にメモリに関して驚くべきものを探すには、それをプロセスの数とメールボックスのサイズに相関させます。どちらか一方が他方を説明してくれるかもしれません。

この特定のダンプでは、1つのプロセスのメールボックスに5百万のメッセージが格納されていたと示されています。これは取得できるすべてのメッセージがパターンマッチしないか、過負荷になっています。そこには数百のメッセージをキューに溜めた数十のプロセスも存在しています。—これは過負荷または競合を指すことがあります。あなたの一般的なクラッシュダンプについて一般的なアドバイスをすることは難しいですが、これらを理解するのに役立つ方法はまだいくつかあります。

## 6.2 Full Mailboxes

For loaded mailboxes, looking at large counters is the best way to do it. If there is one large mailbox, go investigate the process in the crash dump. Figure out if it's happening because it's not matching on some message, or overload. If you have a similar node running, you can log on it and go inspect it. If you find out many mailboxes are loaded, you may want to use recon's `queue_fun.awk` to figure out what function they're running at the time of the crash:

---

```
1 $ awk -v threshold=10000 -f queue_fun.awk /path/to/erl_crash.dump
2 MESSAGE QUEUE LENGTH: CURRENT FUNCTION
3 =====
4 10641: io:wait_io_mon_reply/2
5 12646: io:wait_io_mon_reply/2
6 32991: io:wait_io_mon_reply/2
7 2183837: io:wait_io_mon_reply/2
8 730790: io:wait_io_mon_reply/2
9 80194: io:wait_io_mon_reply/2
10 ...
```

---

This one will run over the crash dump and output all of the functions scheduled to run for processes with at least 10000 messages in their mailbox. In the case of this run, the script showed that the entire node was locking up waiting on IO for `io:format/2` calls, for example.

---

<sup>5</sup> 特にここでは参照カウンタされたバイナリメモリです。これはグローバルのヒープ領域に格納されますが、クラッシュダンプを生成する前にガベージコレクトされ消えます。従ってそのバイナリメモリは過小にレポートされます。より詳しくは [7](#) を参照

## 6.3 非常に多い（もしくは非常に少ない）プロセス

正常かどうかを判別するためにプロセス数を数えることは、ノードの通常時のプロセス数を把握している場合にはとても有用です<sup>6</sup>。

アプリケーションによりますが、通常よりも多い場合には何かがリークしているか、過負荷かもしれません。

プロセス数が通常時と比較して極めて少ない場合、ノードが下記のような出力をして（プロセスを）終了していないかを見てください。

```
Kernel pid terminated (application_controller)
({application_terminated, <AppName>, shutdown})
```

このような場合、そのアプリケーション (AppName) がスーパーバイザ内で再起動制限に引っかかったため、ノードがシャットダウンを行ったことが原因です。一連の問題の原因を詳しく調べるにはエラーログが役に立ちます。

## 6.4 大量のポート数

プロセス数のカウントと同様に、ポート数も通常時の数を把握している時にはシンプルでとても有用です<sup>7</sup>。

ポート数が多い場合、DoS 攻撃や使わなくなったリソースのリークなどに、過負荷になっている可能性があります。リークしているポートの種類（TCP, UTP, ファイル）をみてみることで、リソースの競合やそれらのリソースを使っているコードが間違っているかどうか分かる可能性があります。

## 6.5 メモリ割り当てができない

これらは、おそらくあなたが非常によく見る類のクラッシュです。カバーすべきことが多くあるため、7 章ではその内容の理解と、稼働中のシステムでのデバッグで必要となることについてまとめています。

いずれにせよ、クラッシュダンプは何が問題であったかを事後に把握するために役立ちます。プロセスのメッセージボックスと個々のヒープは通常、問題に対する良い指標です。メールボックスにメールが大量にあるわけではないのにメモリが不足している時には、recon スクリプトにより返されるプロセスヒープとスタックサイズを見てみてください。

---

<sup>6</sup> 詳細は 5.1.3 を参照のこと

<sup>7</sup> 詳細は 5.1.4 を参照のこと

(サイズが) 上に大きく外れている場合には、いくつかのプロセスによりノードのほとんどのメモリが食いつぶされているかもしれないとわかります。同等の量の場合、(スクリプトから) 返されたメモリの量が多くないかを確認してください。

多い (少ない) ようであれば、ダンプの「メモリ」セクションでタイプ (ETS やバイナリーなど) が非常に大きくないかをチェックしてください。想定外のリソースのリークが見つかるかもしれません。

## 6.6 Exercises

### Review Questions

1. How can you choose where a crash dump will be generated?
2. What are common avenues to explore if the crash dump shows that the node ran out of memory?
3. What should you look for if the process count is suspiciously low?
4. If you find the node died with a process having a lot of memory, what could you do to find out which one it was?

### Hands-On

Using the analysis of a crash dump in Section 6.1:

1. What are specific outliers that could point to an issue?
2. Does it look like repeated errors are the issue? If not, what could it be?

There are truckloads of ways for an Erlang node to bleed memory. They go from extremely simple to astonishingly hard to figure out (fortunately, the latter type is also rarer), and it's possible you'll never encounter any problem with them.

You will find out about memory leaks in two ways:

1. A crash dump (see Chapter 6);
2. By finding a worrisome trend in the data you are monitoring.

This chapter will mostly focus on the latter kind of leak, because they're easier to investigate and see grow in real time. We will focus on finding what is growing on the node and common remediation options, handling binary leaks (they're a special case), and detecting memory fragmentation.

## 7.1 Common Sources of Leaks

Whenever someone calls for help saying "oh no, my nodes are crashing", the first step is always to ask for data. Interesting questions to ask and pieces of data to consider are:

- Do you have a crash dump and is it complaining about memory specifically? If not, the issue may be unrelated. If so, go dig into it, it's full of data.
- Are the crashes cyclical? How predictable are they? What else tends to happen at around the same time and could it be related?
- Do crashes coincide with peaks in load on your systems, or do they seem to happen at more or less any time? Crashes that happen especially **during** peak times are often due to bad overload management (see Chapter 3). Crashes that happen at any time,

even when load goes down following a peak are more likely to be actual memory issues.

If all of this seems to point towards a memory leak, install one of the metrics libraries mentioned in Chapter 5 and/or `recon` and get ready to dive in.<sup>1</sup>

The first thing to look at in any of these cases is trends. Check for all types of memory using `erlang:memory()` or some variant of it you have in a library or metrics system. Check for the following points:

- Is any type of memory growing faster than others?
- Is there any type of memory that's taking the majority of the space available?
- Is there any type of memory that never seems to go down, and always up (other than atoms)?

Many options are available depending on the type of memory that's growing.

### 7.1.1 Atom

**Don't use dynamic atoms!** Atoms go in a global table and are cached forever. Look for places where you call `erlang:binary_to_term/1` and `erlang:list_to_atom/1`, and consider switching to safer variants (`erlang:binary_to_term(Bin, [safe])` and `erlang:list_to_existing_atom/1`).

If you use the `xmerl` library that ships with Erlang, consider open source alternatives<sup>2</sup> or figuring the way to add your own SAX parser that can be safe<sup>3</sup>.

If you do none of this, consider what you do to interact with the node. One specific case that bit me in production was that some of our common tools used random names to connect to nodes remotely, or generated nodes with random names that connected to each other from a central server.<sup>4</sup> Erlang node names are converted to atoms, so just having this was enough to slowly but surely exhaust space on atom tables. Make sure you generate them from a fixed set, or slowly enough that it won't be a problem in the long run.

### 7.1.2 Binary

See Section 7.2.

---

<sup>1</sup> See Chapter 4 if you need help to connect to a running node

<sup>2</sup> I don't dislike `exml` or `erlsom`

<sup>3</sup> See Ulf Wiger at <http://erlang.org/pipermail/erlang-questions/2013-July/074901.html>

<sup>4</sup> This is a common approach to figuring out how to connect nodes together: have one or two central nodes with fixed names, and have every other one log to them. Connections will then propagate automatically.

### 7.1.3 Code

The code on an Erlang node is loaded in memory in its own area, and sits there until it is garbage collected. Only two copies of a module can coexist at one time, so looking for very large modules should be easy-ish.

If none of them stand out, look for code compiled with HiPE<sup>5</sup>. HiPE code, unlike regular BEAM code, is native code and cannot be garbage collected from the VM when new versions are loaded. Memory can accumulate, usually very slowly, if many or large modules are native-compiled and loaded at run time.

Alternatively, you may look for weird modules you didn't load yourself on the node and panic if someone got access to your system!

### 7.1.4 ETS

ETS tables are never garbage collected, and will maintain their memory usage as long as records will be left undeleted in a table. Only removing records manually (or deleting the table) will reclaim memory.

In the rare cases you're actually leaking ETS data, call the undocumented `ets:i()` function in the shell. It will print out information regarding number of entries (`size`) and how much memory they take (`mem`). Figure out if anything is bad.

It's entirely possible all the data there is legit, and you're facing the difficult problem of needing to shard your data set and distribute it over many nodes. This is out of scope for this book, so best of luck to you. You can look into compression of your tables if you need to buy time, however.<sup>6</sup>

### 7.1.5 Processes

There are a lot of different ways in which process memory can grow. Most interesting cases will be related to a few common cases: process leaks (as in, you're leaking processes), specific processes leaking their memory, and so on. It's possible there's more than one cause, so multiple metrics are worth investigating. Note that the process count itself is skipped and has been covered before.

---

<sup>5</sup> [http://www.erlang.org/doc/man/HiPE\\_app.html](http://www.erlang.org/doc/man/HiPE_app.html)

<sup>6</sup> See the `compressed` option for `ets:new/2`

## Links and Monitors

Is the global process count indicative of a leak? If so, you may need to investigate unlinked processes, or peek inside supervisors' children lists to see what may be weird-looking.

Finding unlinked (and unmonitored) processes is easy to do with a few basic commands:

---

```
1> [P || P <- processes(),  
    [{_,Ls},{_,Ms}] <- [process_info(P, [links,monitors])],  
    []==Ls, []==Ms].
```

---

This will return a list of processes with neither. For supervisors, just fetching `supervisor:count_children(SupervisorPidOrName)` and seeing what looks normal can be a good pointer.

## Memory Used

The per-process memory model is briefly described in Subsection 7.3.2, but generally speaking, you can find which individual processes use the most memory by looking for their `memory` attribute. You can look things up either as absolute terms or as a sliding window.

For memory leaks, unless you're in a predictable fast increase, absolute values are usually those worth digging into first:

---

```
1> recon:proc_count(memory, 3).  
[<0.175.0>,325276504,  
 [myapp_stats,  
  {current_function,{gen_server,loop,6}},  
  {initial_call,{proc_lib,init_p,5}}]],  
<0.169.0>,73521608,  
 [myapp_giant_sup,  
  {current_function,{gen_server,loop,6}},  
  {initial_call,{proc_lib,init_p,5}}]],  
<0.72.0>,4193496,  
 [gproc,  
  {current_function,{gen_server,loop,6}},  
  {initial_call,{proc_lib,init_p,5}}]]]
```

---

Attributes that may be interesting to check other than `memory` may be any other fields in Subsection 5.2.1, including `message_queue_len`, but `memory` will usually encompass all other types.

## Garbage Collections

It is very well possible that a process uses lots of memory, but only for short periods of time. For long-lived nodes with a large overhead for operations, this is usually not a problem, but whenever memory starts being scarce, such spiky behaviour might be something you want to get rid of.

Monitoring all garbage collections in real-time from the shell would be costly. Instead, setting up Erlang's system monitor<sup>7</sup> might be the best way to go at it.

Erlang's system monitor will allow you to track information such as long garbage collection periods and large process heaps, among other things. A monitor can temporarily be set up as follows:

---

```
1> erlang:system_monitor().
undefined
2> erlang:system_monitor(self(), [{long_gc, 500}]).
undefined
3> flush().
Shell got {monitor,<4683.31798.0>,long_gc,
          [{timeout,515},
           {old_heap_block_size,0},
           {heap_block_size,75113},
           {mbuf_size,0},
           {stack_size,19},
           {old_heap_size,0},
           {heap_size,33878}]}
5> erlang:system_monitor(undefined).
{<0.26706.4961>,[{long_gc,500}]}
6> erlang:system_monitor().
undefined
```

---

The first command checks that nothing (or nobody else) is using a system monitor yet — you don't want to take this away from an existing application or coworker.

The second command will be notified every time a garbage collection takes over 500 milliseconds. The result is flushed in the third command. Feel free to also check for `{large_heap, NumWords}` if you want to monitor such sizes. Be careful to start with large values at first if you're unsure. You don't want to flood your process' mailbox with a bunch of heaps that are 1-word large or more, for example.

---

<sup>7</sup> [http://www.erlang.org/doc/man/erlang.html#system\\_monitor-2](http://www.erlang.org/doc/man/erlang.html#system_monitor-2)



Command 5 unsets the system monitor (exiting or killing the monitor process also frees it up), and command 6 validates that everything worked.

You can then find out if such monitoring messages tend to coincide with the memory increases that seem to result in leaks or overuses, and try to catch culprits before things are too bad. Quickly reacting and digging into the process (possibly with `recon:info/1`) may help find out what's wrong with the application.

### 7.1.6 Nothing in Particular

If nothing seems to stand out in the preceding material, binary leaks (Section 7.2) and memory fragmentation (Section 7.3) may be the culprits. If nothing there fits either, it's possible a C driver, NIF, or even the VM itself is leaking. Of course, a possible scenario is that load on the node and memory usage were proportional, and nothing specifically ended up being leaky or modified. The system just needs more resources or nodes.

## 7.2 Binaries

Erlang's binaries are of two main types: ProcBins and Refc binaries<sup>8</sup>. Binaries up to 64 bytes are allocated directly on the process's heap, and their entire life cycle is spent in there. Binaries bigger than that get allocated in a global heap for binaries only, and each process to use one holds a local reference to it in its local heap. These binaries are reference-counted, and the deallocation will occur only once all references are garbage-collected from all processes that pointed to a specific binary.

In 99% of the cases, this mechanism works entirely fine. In some cases, however, the process will either:

1. do too little work to warrant allocations and garbage collection;
2. eventually grow a large stack or heap with various data structures, collect them, then get to work with a lot of refc binaries. Filling the heap again with binaries (even though a virtual heap is used to account for the refc binaries' real size) may take a lot of time, giving long delays between garbage collections.

---

<sup>8</sup> [http://www.erlang.org/doc/efficiency\\_guide/binaryhandling.html#id65798](http://www.erlang.org/doc/efficiency_guide/binaryhandling.html#id65798)

## 7.2.1 Detecting Leaks

Detecting leaks for reference-counted binaries is easy enough: take a measure of all of each process' list of binary references (using the `binary` attribute), force a global garbage collection, take another snapshot, and calculate the difference.

This can be done directly with `recon:bin_leak(Max)` and looking at the node's total memory before and after the call:

---

```
1> recon:bin_leak(5).
[<0.4612.0>,-5580,
  [{current_function,{gen_fsm,loop,7}},
   {initial_call,{proc_lib,init_p,5}}]],
<0.17479.0>,-3724,
  [{current_function,{gen_fsm,loop,7}},
   {initial_call,{proc_lib,init_p,5}}]],
<0.31798.0>,-3648,
  [{current_function,{gen_fsm,loop,7}},
   {initial_call,{proc_lib,init_p,5}}]],
<0.31797.0>,-3266,
  [{current_function,{gen,do_call,4}},
   {initial_call,{proc_lib,init_p,5}}]],
<0.22711.1>,-2532,
  [{current_function,{gen_fsm,loop,7}},
   {initial_call,{proc_lib,init_p,5}}]]]
```

---

This will show how many individual binaries were held and then freed by each process as a delta. The value `-5580` means there were 5580 fewer refc binaries after the call than before.

It is normal to have a given amount of them stored at any point in time, and not all numbers are a sign that something is bad. If you see the memory used by the VM go down drastically after running this call, you may have had a lot of idling refc binaries.

Similarly, if you instead see some processes hold impressively large numbers of them<sup>9</sup>, that might be a good sign you have a problem.

You can further validate the top consumers in total binary memory by using the special `binary_memory` attribute supported in `recon`:

---

```
1> recon:proc_count(binary_memory, 3).
```

---

<sup>9</sup> We've seen some processes hold hundreds of thousands of them during leak investigations at Heroku!

```
[{<0.169.0>,77301349,
 [app_sup,
  {current_function,{gen_server,loop,6}},
  {initial_call,{proc_lib,init_p,5}}]},
 {<0.21928.1>,9733935,
  [{current_function,{erlang,hibernate,3}},
   {initial_call,{proc_lib,init_p,5}}]},
 {<0.12386.1172>,7208179,
  [{current_function,{erlang,hibernate,3}},
   {initial_call,{proc_lib,init_p,5}}]}]
```

---

This will return the N top processes sorted by the amount of memory the refc binaries reference to hold, and can help point to specific processes that hold a few large binaries, instead of their raw amount. You may want to try running this function **before** `recon:bin_leak/1`, given the latter garbage collects the entire node first.

## 7.2.2 Fixing Leaks

Once you've established you've got a binary memory leak using `recon:bin_leak(Max)`, it should be simple enough to look at the top processes and see what they are and what kind of work they do.

Generally, refc binaries memory leaks can be solved in a few different ways, depending on the source:

- call garbage collection manually at given intervals (icky, but somewhat efficient);
- stop using binaries (often not desirable);
- use `binary:copy/1-2`<sup>10</sup> if keeping only a small fragment (usually less than 64 bytes) of a larger binary;<sup>11</sup>
- move work that involves larger binaries to temporary one-off processes that will die when they're done (a lesser form of manual GC!);
- or add hibernation calls when appropriate (possibly the cleanest solution for inactive processes).

The first two options are frankly not agreeable and should not be attempted before all else failed. The last three options are usually the best ones to be used.

---

<sup>10</sup> <http://www.erlang.org/doc/man/binary.html#copy-1>

<sup>11</sup> It might be worth copying even a larger fragment of a refc binary. For example, copying 10 megabytes off a 2 gigabytes binary should be worth the short-term overhead if it allows the 2 gigabytes binary to be garbage-collected while keeping the smaller fragment longer.

## Routing Binaries

There's a specific solution for a specific use case some Erlang users have reported. The problematic use case is usually having a middleman process routing binaries from one process to another one. That middleman process will therefore acquire a reference to every binary passing through it and risks being a common major source of refc binaries leaks.

The solution to this pattern is to have the router process return the pid to route to and let the original caller move the binary around. This will make it so that only processes that do **need** to touch the binaries will do so.

A fix for this can be implemented transparently in the router's API functions, without any visible change required by the callers.

## 7.3 Memory Fragmentation

Memory fragmentation issues are intimately related to Erlang's memory model, as described in Section 7.3.2. It is by far one of the trickiest issues of running long-lived Erlang nodes (often when individual node uptime reaches many months), and will show up relatively rarely.

The general symptoms of memory fragmentation are large amounts of memory being allocated during peak load, and that memory not going away after the fact. The damning factor will be that the node will internally report much lower usage (through `erlang:memory()`) than what is reported by the operating system.

### 7.3.1 Finding Fragmentation

The `recon_alloc` module was developed specifically to detect and help point towards the resolution of such issues.

Given how rare this type of issue has been so far over the community (or happened without the developers knowing what it was), only broad steps to detect things are defined. They're all vague and require the operator's judgement.

#### Check Allocated Memory

Calling `recon_alloc:memory/1` will report various memory metrics with more flexibility than `erlang:memory/0`. Here are the possibly relevant arguments:

1. call `recon_alloc:memory(usage)`. This will return a value between 0 and 1 representing a percentage of memory that is being actively used by Erlang terms versus the memory that the Erlang VM has obtained from the OS for such purposes. If the usage

is close to 100%, you likely do not have memory fragmentation issues. You're just using a lot of it.

2. check if `recon_alloc:memory(allocated)` matches what the OS reports.<sup>12</sup> It should match it fairly closely if the problem is really about fragmentation or a memory leak from Erlang terms.

That should confirm if memory seems to be fragmented or not.

### Find the Guilty Allocator

Call `recon_alloc:memory(allocated_types)` to see which type of util allocator (see Section 7.3.2) is allocating the most memory. See if one looks like an obvious culprit when you compare the results with `erlang:memory()`.

Try `recon_alloc:fragmentation(current)`. The resulting data dump will show different allocators on the node with various usage ratios.<sup>13</sup>

If you see very low ratios, check if they differ when calling `recon_alloc:fragmentation(max)`, which should show what the usage patterns were like under your max memory load.

If there is a big difference, you are likely having issues with memory fragmentation for a few specific allocator types following usage spikes.

## 7.3.2 Erlang's Memory Model

### The Global Level

To understand where memory goes, one must first understand the many allocators being used. Erlang's memory model, for the entire virtual machine, is hierarchical. As shown in Figure 7.1, there are two main allocators, and a bunch of sub-allocators (numbered 1-9). The sub-allocators are the specific allocators used directly by Erlang code and the VM for most data types:<sup>14</sup>

1. `temp_alloc`: does temporary allocations for short use cases (such as data living within a single C function call).
2. `ehheap_alloc`: heap data, used for things such as the Erlang processes' heaps.
3. `binary_alloc`: the allocator used for reference counted binaries (what their 'global heap' is). Reference counted binaries stored in an ETS table remain in this allocator.

---

<sup>12</sup> You can call `recon_alloc:set_unit(Type)` to set the values reported by `recon_alloc` in bytes, kilobytes, megabytes, or gigabytes

<sup>13</sup> More information is available at [http://ferd.github.io/recon/recon\\_alloc.html](http://ferd.github.io/recon/recon_alloc.html)

<sup>14</sup> The complete list of where each data type lives can be found in [erts/emulator/beam/erl\\_alloc.types](erts/emulator/beam/erl_alloc.types)



Figure 7.1 Erlang’s Memory allocators and their hierarchy. Not shown is the special **super carrier**, optionally allowing to pre-allocate (and limit) all memory available to the Erlang VM since R16B03.

4. **ets\_alloc**: ETS tables store their data in an isolated part of memory that isn’t garbage collected, but allocated and deallocated as long as terms are being stored in tables.
5. **driver\_alloc**: used to store driver data in particular, which doesn’t keep drivers that generate Erlang terms from using other allocators. The driver data allocated here contains locks/mutexes, options, Erlang ports, etc.
6. **sl\_alloc**: short-lived memory blocks will be stored there, and include items such as some of the VM’s scheduling information or small buffers used for some data types’ handling.
7. **ll\_alloc**: long-lived allocations will be in there. Examples include Erlang code itself and the atom table, which stay there.
8. **fix\_alloc**: allocator used for frequently used fixed-size blocks of memory. One example of data used there is the internal processes’ C struct, used internally by the VM.
9. **std\_alloc**: catch-all allocator for whatever didn’t fit the previous categories. The process registry for named process is there.

By default, there will be one instance of each allocator per scheduler (and you should have one scheduler per core), plus one instance to be used by linked-in drivers using async threads. This ends up giving you a structure a bit like in Figure 7.1, but split it in  $N$  parts at each leaf.

Each of these sub-allocators will request memory from **mseg\_alloc** and **sys\_alloc** depending on the use case, and in two possible ways. The first way is to act as a multiblock carrier (**mbcs**), which will fetch chunks of memory that will be used for many Erlang terms at once.



图 7.2 Example memory allocated in a specific sub-allocator

For each `mbc`, the VM will set aside a given amount of memory (about 8MB by default in our case, which can be configured by tweaking VM options), and each term allocated will be free to go look into the many multiblock carriers to find some decent space in which to reside.

Whenever the item to be allocated is greater than the single block carrier threshold (`sbct`)<sup>15</sup>, the allocator switches this allocation into a single block carrier (`sbcs`). A single block carrier will request memory directly from `mseg_alloc` for the first `mmsbc`<sup>16</sup> entries, and then switch over to `sys_alloc` and store the term there until it's deallocated.

So looking at something such as the binary allocator, we may end up with something similar to Figure 7.2

Whenever a multiblock carrier (or the first `mmsbc`<sup>17</sup> single block carriers) can be reclaimed, `mseg_alloc` will try to keep it in memory for a while so that the next allocation spike that hits your VM can use pre-allocated memory rather than needing to ask the system for more each time.

You then need to know the different memory allocation strategies of the Erlang virtual machine:

1. Best fit (`bf`)
2. Address order best fit (`aobf`)
3. Address order first fit (`aoff`)

<sup>15</sup> [http://erlang.org/doc/man/erts\\_alloc.html#M\\_sbct](http://erlang.org/doc/man/erts_alloc.html#M_sbct)

<sup>16</sup> [http://erlang.org/doc/man/erts\\_alloc.html#M\\_mmsbc](http://erlang.org/doc/man/erts_alloc.html#M_mmsbc)

<sup>17</sup> [http://erlang.org/doc/man/erts\\_alloc.html#M\\_mmsbc](http://erlang.org/doc/man/erts_alloc.html#M_mmsbc)



图 7.3 Example memory allocated in a specific sub-allocator

4. Address order first fit carrier best fit (**aoffcbf**)
5. Address order first fit carrier address order best fit (**aoffcaobf**)
6. Good fit (**gf**)
7. A fit (**af**)

Each of these strategies can be configured individually for each `alloc_util` allocator<sup>18</sup>

For **best fit** (**bf**), the VM builds a balanced binary tree of all the free blocks' sizes, and will try to find the smallest one that will accommodate the piece of data and allocate it there. In Figure 7.3, having a piece of data that requires three blocks would likely end in area 3.

**Address order best fit** (**aobf**) will work similarly, but the tree instead is based on the addresses of the blocks. So the VM will look for the smallest block available that can accommodate the data, but if many of the same size exist, it will favor picking one that has a lower address. If I have a piece of data that requires three blocks, I'll still likely end up in area 3, but if I need two blocks, this strategy will favor the first **mbcs** in Figure 7.3 with area 1 (instead of area 5). This could make the VM have a tendency to favor the same carriers for many allocations.

**Address order first fit** (**aoff**) will favor the address order for its search, and as soon as a block fits, **aoff** uses it. Where **aobf** and **bf** would both have picked area 3 to allocate four blocks in Figure 7.3, this one will get area 2 as a first priority given its address is lowest. In Figure 7.4, if we were to allocate four blocks, we'd favor block 1 to block 3 because its address

<sup>18</sup> [http://erlang.org/doc/man/erts\\_alloc.html#M\\_as](http://erlang.org/doc/man/erts_alloc.html#M_as)





Figure 7.4 Example memory allocated in a specific sub-allocator

is lower, whereas **bf** would have picked either 3 or 4, and **aobf** would have picked 3.

**Address order first fit carrier best fit** (**aoffcbf**) is a strategy that will first favor a carrier that can accommodate the size and then look for the best fit within that one. So if we were to allocate two blocks in Figure 7.4, **bf** and **aobf** would both favor block 5, **aoff** would pick block 1. **aoffcbf** would pick area 2, because the first **mbcs** can accommodate it fine, and area 2 fits it better than area 1.

**Address order first fit carrier address order best fit** (**aoffcaobf**) will be similar to **aoffcbf**, but if multiple areas within a carrier have the same size, it will favor the one with the smallest address between the two rather than leaving it unspecified.

**Good fit** (**gf**) is a different kind of allocator; it will try to work like best fit (**bf**), but will only search for a limited amount of time. If it doesn't find a perfect fit there and then, it will pick the best one encountered so far. The value is configurable through the **mbsd**<sup>19</sup> VM argument.

**A fit** (**af**), finally, is an allocator behaviour for temporary data that looks for a single existing memory block, and if the data can fit, **af** uses it. If the data can't fit, **af** allocates a new one.

Each of these strategies can be applied individually to every kind of allocator, so that the heap allocator and the binary allocator do not necessarily share the same strategy.

Finally, starting with Erlang version 17.0, each **alloc\_util** allocator on each scheduler has what is called a **mbcs pool**. The **mbcs** pool is a feature used to fight against memory

<sup>19</sup> [http://www.erlang.org/doc/man/erts\\_alloc.html#M\\_mbsd](http://www.erlang.org/doc/man/erts_alloc.html#M_mbsd)

fragmentation on the VM. When an allocator gets to have one of its multiblock carriers become mostly empty,<sup>20</sup> the carrier becomes **abandoned**.

This abandoned carrier will stop being used for new allocations, until new multiblock carriers start being required. When this happens, the carrier will be fetched from the `mbcs` pool. This can be done across multiple `alloc_util` allocators of the same type across schedulers. This allows the VM to cache mostly-empty carriers without forcing deallocation of their memory.<sup>21</sup> It also enables the migration of carriers across schedulers when they contain little data, according to their needs.

### The Process Level

On a smaller scale, for each Erlang process, the layout still is a bit different. It basically has this piece of memory that can be imagined as one box:

---

```
1 [          ]
```

---

On one end you have the heap, and on the other, you have the stack:

---

```
1 [heap |      | stack]
```

---

In practice there's more data (you have an old heap and a new heap, for generational GC, and also a virtual binary heap, to account for the space of reference-counted binaries on a specific sub-allocator not used by the process — `binary_alloc` vs. `eheap_alloc`):

---

```
1 [heap    ||      stack]
```

---

The space is allocated more and more up until either the stack or the heap can't fit in anymore. This triggers a minor GC. The minor GC moves the data that can be kept into the old heap. It then collects the rest, and may end up reallocating more space.

After a given number of minor GCs and/or reallocations, a full-sweep GC is performed, which inspects both the new and old heaps, frees up more space, and so on. When a process dies, both the stack and heap are taken out at once. reference-counted binaries are decreased, and if the counter is at 0, they vanish.

---

<sup>20</sup> The threshold is configurable through [http://www.erlang.org/doc/man/erts\\_alloc.html#M\\_acul](http://www.erlang.org/doc/man/erts_alloc.html#M_acul)

<sup>21</sup> In cases this consumes too much memory, the feature can be disabled with the options `+MBacul 0`.

When that happens, over 80% of the time, the only thing that happens is that the memory is marked as available in the sub-allocator and can be taken back by new processes or other ones that may need to be resized. Only after having this memory unused — and the multiblock carrier unused also — is it returned to `mseg_alloc` or `sys_alloc`, which may or may not keep it for a while longer.

### 7.3.3 Fixing Memory Fragmentation with a Different Allocation Strategy

Tweaking your VM's options for memory allocation may help.

You will likely need to have a good understanding of what your type of memory load and usage is, and be ready to do a lot of in-depth testing. The `recon_alloc` module contains a few helper functions to provide guidance, and the module's documentation<sup>22</sup> should be read at this point.

You will need to figure out what the average data size is, the frequency of allocation and deallocation, whether the data fits in `mbcs` or `sbc`s, and you will then need to try playing with a bunch of the options mentioned in `recon_alloc`, try the different strategies, deploy them, and see if things improve or if they impact times negatively.

This is a very long process for which there is no shortcut, and if issues happen only every few months per node, you may be in for the long haul.

## 7.4 Exercises

### Review Questions

1. Name some of the common sources of leaks in Erlang programs.
2. What are the two main types of binaries in Erlang?
3. What could be to blame if no specific data type seems to be the source of a leak?
4. If you find the node died with a process having a lot of memory, what could you do to find out which one it was?
5. How could code itself cause a leak?
6. How can you find out if garbage collections are taking too long to run?

---

<sup>22</sup> [http://ferd.github.io/recon/recon\\_alloc.html](http://ferd.github.io/recon/recon_alloc.html)

## Open-ended Questions

1. How could you verify if a leak is caused by forgetting to kill processes, or by processes using too much memory on their own?
2. A process opens a 150MB log file in binary mode to go extract a piece of information from it, and then stores that information in an ETS table. After figuring out you have a binary memory leak, what should be done to minimize binary memory usage on the node?
3. What could you use to find out if ETS tables are growing too fast?
4. What steps should you go through to find out that a node is likely suffering from fragmentation? How could you disprove the idea that it could be due to a NIF or driver leaking memory?
5. How could you find out if a process with a large mailbox (from reading `message_queue_len`) seems to be leaking data from there, or never handling new messages?
6. A process with a large memory footprint seems to be rarely running garbage collections. What could explain this?
7. When should you alter the allocation strategies on your nodes? Should you prefer to tweak this, or the way you wrote code?

## Hands-On

1. Using any system you know or have to maintain in Erlang (including toy systems), can you figure out if there are any binary memory leaks on there?

While memory leaks tend to absolutely kill your system, CPU exhaustion tends to act like a bottleneck and limits the maximal work you can get out of a node. Erlang developers will have a tendency to scale horizontally when they face such issues. It is often an easy enough job to scale out the more basic pieces of code out there. Only centralized global state (process registries, ETS tables, and so on) usually need to be modified.<sup>1</sup> Still, if you want to optimize locally before scaling out at first, you need to be able to find your CPU and scheduler hogs.

It is generally difficult to properly analyze the CPU usage of an Erlang node to pin problems to a specific piece of code. With everything concurrent and in a virtual machine, there is no guarantee you will find out if a specific process, driver, your own Erlang code, NIFs you may have installed, or some third-party library is eating up all your processing power.

The existing approaches are often limited to profiling and reduction-counting if it's in your code, and to monitoring the scheduler's work if it might be anywhere else (but also your code).

## 8.1 Profiling and Reduction Counts

To pin issues to specific pieces of Erlang code, as mentioned earlier, there are two main approaches. One will be to do the old standard profiling routine, likely using one of the following applications:<sup>2</sup>

---

<sup>1</sup> Usually this takes the form of sharding or finding a state-replication scheme that's suitable, and little more. It's still a decent piece of work, but nothing compared to finding out most of your program's semantics aren't applicable to distributed systems given Erlang usually forces your hand there in the first place.

<sup>2</sup> All of these profilers work using Erlang tracing functionality with almost no restraint. They will have an impact on the run-time performance of the application, and shouldn't be used in production.

- **eprof**,<sup>3</sup> the oldest Erlang profiler around. It will give general percentage values and will mostly report in terms of time taken.
- **fprof**,<sup>4</sup> a more powerful replacement of eprof. It will support full concurrency and generate in-depth reports. In fact, the reports are so deep that they are usually considered opaque and hard to read.
- **eflame**,<sup>5</sup> the newest kid on the block. It generates flame graphs to show deep call sequences and hot-spots in usage on a given piece of code. It allows one to quickly find issues with a single look at the final result.

It will be left to the reader to thoroughly read each of these application's documentation. The other approach will be to run `recon:proc_window/3` as introduced in Subsection 5.2.1:

---

```
1> recon:proc_window(reductions, 3, 500).
[{<0.46.0>,51728,
  [{current_function,{queue,in,2}},
   {initial_call,{erlang,apply,2}}]},
 {<0.49.0>,5728,
  [{current_function,{dict,new,0}},
   {initial_call,{erlang,apply,2}}]},
 {<0.43.0>,650,
  [{current_function,{timer,sleep,1}},
   {initial_call,{erlang,apply,2}}]}]
```

---

The reduction count has a direct link to function calls in Erlang, and a high count is usually the synonym of a high amount of CPU usage.

What's interesting with this function is to try it while a system is already rather busy,<sup>6</sup> with a relatively short interval. Repeat it many times, and you should hopefully see a pattern emerge where the same processes (or the same **kind** of processes) tend to always come up on top.

Using the code locations<sup>7</sup> and current functions being run, you should be able to identify what kind of code hogs all your schedulers.

---

<sup>3</sup> <http://www.erlang.org/doc/man/eprof.html>

<sup>4</sup> <http://www.erlang.org/doc/man/fprof.html>

<sup>5</sup> <https://github.com/proger/eflame>

<sup>6</sup> See Subsection 5.1.2

<sup>7</sup> Call `recon:info(PidTerm, location)` or `process_info(Pid, current_stacktrace)` to get this information.

## 8.2 System Monitors

If nothing seems to stand out through either profiling or checking reduction counts, it's possible some of your work ends up being done by NIFs, garbage collections, and so on. These kinds of work may not always increment their reductions count correctly, so they won't show up with the previous methods, only through long run times.

To find about such cases, the best way around is to use `erlang:system_monitor/2`, and look for `long_gc` and `long_schedule`. The former will show whenever garbage collection ends up doing a lot of work (it takes time!), and the latter will likely catch issues with busy processes, either through NIFs or some other means, that end up making them hard to de-schedule.<sup>8</sup>

We've seen how to set such a system monitor In Garbage Collection in 7.1.5, but here's a different pattern<sup>9</sup> I've used before to catch long-running items:

---

```
1> F = fun(F) ->
    receive
        {monitor, Pid, long_schedule, Info} ->
            io:format("monitor=long_schedule pid=~p info=~p~n", [Pid, Info]);
        {monitor, Pid, long_gc, Info} ->
            io:format("monitor=long_gc pid=~p info=~p~n", [Pid, Info])
    end,
    F(F)
end.
2> Setup = fun(Delay) -> fun() ->
    register(temp_sys_monitor, self()),
    erlang:system_monitor(self(), [{long_schedule, Delay}, {long_gc, Delay}]),
    F(F)
end end.
3> spawn_link(Setup(1000)).
<0.1293.0>
monitor=long_schedule pid=<0.54.0> info=[{timeout,1102},
                                         {in,{some_module,some_function,3}},
                                         {out,{some_module,some_function,3}}]
```

---

Be sure to set the `long_schedule` and `long_gc` values to large-ish values that might be

---

<sup>8</sup> Long garbage collections count towards scheduling time. It is very possible that a lot of your long schedules will be tied to garbage collections depending on your system.

<sup>9</sup> If you're on 17.0 or newer versions, the shell functions can be made recursive far more simply by using their named form, but to have the widest compatibility possible with older versions of Erlang, I've let them as is.

reasonable to you. In this example, they're set to 1000 milliseconds. You can either kill the monitor by calling `exit(whereis(temp_sys_monitor), kill)` (which will in turn kill the shell because it's linked), or just disconnect from the node (which will kill the process because it's linked to the shell.)

This kind of code and monitoring can be moved to its own module where it reports to a long-term logging storage, and can be used as a canary for performance degradation or overload detection.

### 8.2.1 Suspended Ports

An interesting part of system monitors that didn't fit anywhere but may have to do with scheduling is regarding ports. When a process sends too many message to a port and the port's internal queue gets full, the Erlang schedulers will forcibly de-schedule the sender until space is freed. This may end up surprising a few users who didn't expect that implicit back-pressure from the VM.

This kind of event can be monitored by passing in the atom `busy_port` to the system monitor. Specifically for clustered nodes, the atom `busy_dist_port` can be used to find when a local process gets de-scheduled when contacting a process on a remote node whose inter-node communication was handled by a busy port.

If you find out you're having problems with these, try replacing your sending functions where in critical paths with `erlang:port_command(Port, Data, [nosuspend])` for ports, and `erlang:send(Pid, Msg, [nosuspend])` for messages to distributed processes. They will then tell you when the message could not be sent and you would therefore have been descheduled.

## 8.3 Exercises

### Review Questions

1. What are the two main approaches to pin issues about CPU usages?
2. Name some of the profiling tools available. What approaches are preferable for production use? Why?
3. Why can long scheduling monitors be useful to find CPU or scheduler over-consumption?



## Open-ended Questions

1. If you find that a process doing very little work with reductions ends up being scheduled for long periods of time, what can you guess about it or the code it runs?
2. Can you set up a system monitor and then trigger it with regular Erlang code? Can you use it to find out for how long processes seem to be scheduled on average? You may need to manually start random processes from the shell that are more aggressive in their work than those provided by the existing system.

## 第 9 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<sup>1</sup>。Erlang では、トレースは開発中あるいは稼働中の本番システムの診断など、システムのライフサイクルのどこでも便利です。

トレースを行ういくつかの Erlang プログラムがあります。

- `sys`<sup>2</sup> は OTP に標準で付属されており、利用者はトレース機能のカスタマイズや、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`<sup>3</sup> 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十分です。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

<sup>1</sup> 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。

<sup>2</sup> <http://www.erlang.org/doc/man/sys.html>

<sup>3</sup> <http://www.erlang.org/doc/man/dbg.html>

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。

- `redbug`<sup>4</sup> は `eper`<sup>5</sup> スイートの一部で、本番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`<sup>6</sup> 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<sup>7</sup>。

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。

## 9.1 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<sup>8</sup>。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 new pid（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関数の代わりになります。関数の指定は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<sup>9</sup>

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、9.1にあるように、両者の共通部分です。

---

<sup>4</sup> <https://github.com/massemannet/eper/blob/master/doc/redbug.txt>

<sup>5</sup> <https://github.com/massemannet/eper>

<sup>6</sup> [http://ferd.github.io/recon/recon\\_trace.html](http://ferd.github.io/recon/recon_trace.html)

<sup>7</sup> 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、ユーザはおおよそ同じことを実現できます

<sup>8</sup> 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

<sup>9</sup> [http://www.erlang.org/doc/apps/erts/match\\_spec.html](http://www.erlang.org/doc/apps/erts/match_spec.html)



図 9.1 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

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。

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

## 9.2 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、あなたがいじくり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。本番環境で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<sup>10</sup>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。

レート制限は静的な値によるカウントか、一定期間内にマッチした数の 2 つの方法で行うことが

<sup>10</sup> <http://www.erlang.org/doc/man/ets.html#fun2ms-1>

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、トレースの方法を見ていきます。

---

```
%% 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)

%% 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)

%% 指定の 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)
```

---

各々の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、このおかしい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。

## 9.3 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す、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はいませんが、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、少しずつ増やして行ってください。

## 9.4 演習

### 復習問題

1. Erlang では通常なぜデバッガの使用が制限されていますか？
2. OTP プロセスをトレースする時に使用できるオプションは？
3. 指定の関数やプロセスがトレースされるかどうかを決めるのは何？
4. recon\_trace あるいはその他のツールで、トレースを止める方法は？
5. エクスポートされていない関数の呼び出しをトレースする方法は？

### 自由回答

1. トレースにタイムスタンプを記録する時に、VM のトレース機能を直接利用するようにしたくなるのはどういう時ですか？ これによる欠点は何ですか？
2. ノードから送信されるトラフィックが SSL 経由の、マルチテナントシステムを想像してみてください。ただし、（顧客からのクレームに対応するため）送信されるデータをバリデートしたいので、平文で中身を参照できる必要があります。ssl ソケット経由で送信されたデータを覗くための方法を考えられますか？ しかも、その他の顧客宛のデータは覗かずにです。

### ハンズオン

[https://github.com/ferd/recon\\_demo](https://github.com/ferd/recon_demo) にあるコードを利用してください（コードの中身をきちんと理解している必要があるかもしれません）

1. メッセージを吐きまくるプロセス (council\_member) は自身にメッセージを送ることができますか？（ヒント：これは登録された名前 (register\_name) で動作しますか？ その吐きまくるプロセスを確認、また、自身にメッセージを送ったかを知る必要はありますか？）



2. 全体で送られるメッセージの頻度を見積もることはできますか？
3. いずれかのトレースツールを使って、ノードをクラッシュさせることはできますか？（ヒント：非常に柔軟性が高いので、dbg を使うと簡単です）

## 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。