Stuff Goes Bad:
Erlang in Anger

Fred Hébert 著、elixir.jp 訳

2018 年 6 月 21 日
Git commit ID: e1a27f5

FRED HEBERT

STUFF GOES BAD: ERLANG IN ANGER

次の皆様のサポート、レビュー、そして編集に感謝します。

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

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

v1.1.0 *2016-04-06*

目次

図目次

はじめに

ソフトウェアを実行するにあたって

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

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

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

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

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

Erlang というのは人体の免疫システムと同様の手法が取られているプログラミング環境です。一方で、他のたいていの言語は体内に病原菌が一切入らないようにするような衛生についてだけを考えています。どちらも私にとって極めて重要なプログラミング環境ものです。ほぼすべての環境でそれぞれに衛生状況が異なります。実行時のエラーがうまく対処されて、そのまま生き残れるよ

---

[1] 生命に関わるシステムは通常この議論の対象外です。

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

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

うな治癒の仕組みを持っているプログラミング環境は Erlang の他にほとんどありません。

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

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

対象読者

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

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

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

本書の読み進め方

本書は二部構成です。

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

[4] 無料の資料が必要であれば Learn You Some Erlang や通常の Erlang ドキュメントをおすすめします。

[5] 訳注:日本語資料としては、Learn you some Erlang for great good!日本語訳とその書籍版をおすすめします。

[6] Erlang を screen や tmux のセッションで実行する、というのはデプロイ戦略では**ありません**

てシステム設計における過負荷への計画の仕方 (第 3 章) を説明します。

第 II 部では Erlang 衛生兵になって、既存の動作しているシステムに取り組みます。この部では実行中のノードへの接続方法の解説 (第 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`[7]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。

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

第I部

Writing Applications

第1章

コードベースへの飛び込み方

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

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

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

1.1　生の Erlang

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

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

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

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

1.2 OTP アプリケーション

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

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

わずかな違いはあるかもしれませんが、一般的な構造は同じです。

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

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

そして

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

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

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

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

の 2 種類です。

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

1.2.1 ライブラリアプリケーション

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

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

1.2.2 標準アプリケーション

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

1. *appname*
2. *appname*_app

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

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

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

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

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

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

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

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

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

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

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

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

て使われます。

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

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

1.2.3 依存関係

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

こうした依存関係を使って各アプリケーションの短い解説を見ることで、何がどこにあるかの大まかな地図を描くのに役立つでしょう。似たダイ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。[6]似たダイアグラムが `observer`[7]アプリケーションを使うことで得られますが、各監視ツリーのものになります。これらをまとめることで、コードベースの中で何が何をしているかを簡単に見つけられるようになるでしょう。

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

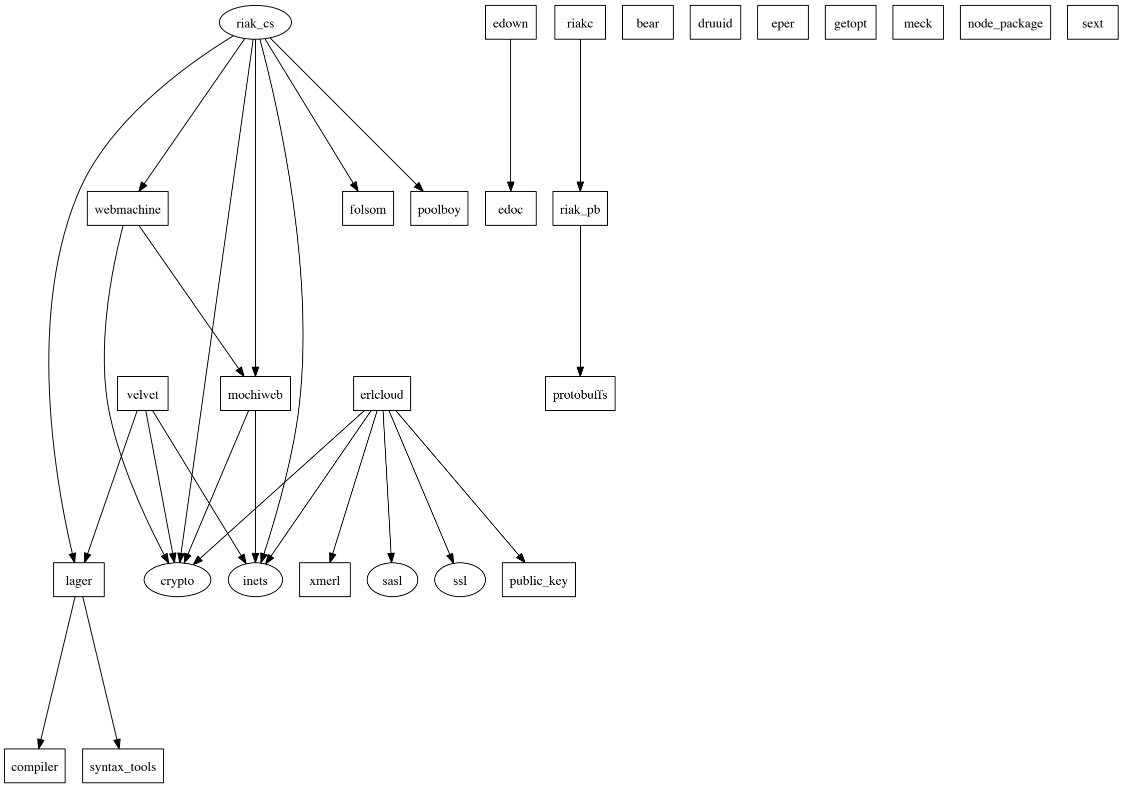


図 1.1　Basho のオープンソースクラウドライブラリである riak_cs の依存関係を表したグラフです。このグラフは kernel や stdlib といった必ず依存するようなものは除いています。楕円はアプリケーションで、四角はライブラリアプリケーションです。

1.3　OTP リリース

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

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

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

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

1.4 演習

復習問題

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

ハンズオン

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

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

---

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

第2章

オープンソースのErlang製ソフトウェアをビルドする

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

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

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

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

---

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

2.1 プロジェクト構造

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

2.1.1 OTP アプリケーション

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

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

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

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

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

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

[3] まだビルドされていないパッケージのローカルキャッシュは除きます。

```
{deps, [
  %% Hex.pm Packages
  myapp,
  {myapp,"1.0.0"},
  %% source dependencies
  {myapp, {git, "git://github.com/user/myapp.git", {ref, "aef728"}}},
  {myapp, {git, "https://github.com/user/myapp.git", {branch, "master"}}},
  {myapp, {hg, "https://othersite.com/user/myapp", {tag, "3.0.0"}}}
 ]}.
```

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

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

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

2.1.2 OTP リリース

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

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

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

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

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

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

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

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

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

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

2.2 スーパーバイザーと start_link セマンティクス

複雑な本番システムでは、多くの障害やエラーは一時的なもので、処理を再実行するのは良いことです—Jim Gray の論文[6]では、一時的なバグを扱う場合、システムの *Mean Times Between Failures*（**障害間隔平均時間**）(MTBF) で見た時の 4 回に 1 回程度行うのが良いと引用していました。スー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、さもなければ頻繁に失敗することになります。

---

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

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

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

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

2.2.1 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。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、コネクションに失敗する可能性があります。うまく行かない[7]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

[7] 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

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う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でもあります。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。

```
init(Args) ->
    Opts = parse_args(Args),
    {ok, Port} = connect(Opts),
    {ok, #state{sock=Port, opts=Opts}}.

[...]

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

```
    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。

```
init(Args) ->
    Opts = parse_args(Args),
    %% いずれにせよここでベストエフォートで接続を試みて
    %% 接続が出来なかった場合には備えましょう。
    self() ! reconnect,
    {ok, #state{sock=undefined, opts=Opts}}.

[...]

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

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

私が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か。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か。
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 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。

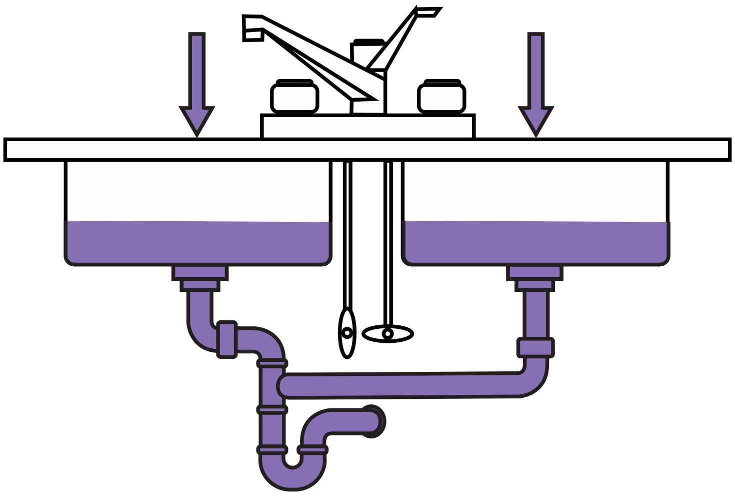
2. アプリケーションをどこかにホストしてください。(GitHub、Bitbukect、ローカルサーバーなど) そしてそのアプリケーションに依存するリリースをビルドしてください。
3. main アプリケーションのワーカー (`council_member`) はサーバーの起動とそこへの接続を自身の `init/1` 関数で行っています。この接続を init 関数の外で行なえますか。このアプリの用途においてそうすることの利点はあるでしょうか。

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。[1] これに対処する方法はたくさんありますが、自身が開発しているシステムを適切に理解することで、どう対処するかを決めることができます。

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



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

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

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

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

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

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

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

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

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

3.1 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`[2]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。

私が経験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。[3]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。

[2] http://www.erlang.org/doc/man/error_logger.html で定義されています。

[3] 10 ミリ秒は非常に短いですが、リアルタイムビッディングに使わ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[4]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。[5]

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

[4] 本番環境で事実に基づいて過負荷になるもの

[5] `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[6]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[7]どのプロセスに問題があるかわかるでしょう。そのあと、メッセージを必要なように処理することで問題を修正できます。

3.2 Restricting Input

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

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

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

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

3.2.1 How Long Should a Time Out Be

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

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

[6] 5.1 の節を見ましょう
[7] 5.2.1 の節を見ましょう
[8] There's a tradeoff between slowing down all requests equally or using rate-limiting, both of which are valid. Rate-limiting per user would mean you'd still need to increase capacity or lower the limits of all users when more new users hammer your system, whereas a synchronous system that indiscriminately blocks should adapt to any load with more ease, but possibly unfairly.

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

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

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

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

3.2.2 Asking For Permission

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

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

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

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

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

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

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

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

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

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

right can be granted or not.

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

3.2.3 What Users See

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

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

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

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

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

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

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

3.3 Discarding Data

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

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

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

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

3.3.1 Random Drop

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

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

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

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

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

If you aim to keep 95% of the messages you send, the authorization could be written by a call to `case drop:random(0.95) of true -> send(); false -> drop() end`, or a shorter `drop:random(0.95) andalso send()` if you don't need to do anything specific when dropping a message.

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

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

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

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

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

3.3.2 Queue Buffers

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

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

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

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

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

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

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

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

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

3.3.3 Stack Buffers

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

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

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

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

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

---

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

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

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

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

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

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

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

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

3.3.4 Time-Sensitive Buffers

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

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

3.3.5 Dealing With Constant Overload

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

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

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

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

ETS tables will be required for both approaches.

Generally speaking, the first approach could work well with the regular process registry: you take `N` processes to divide up the load, give them all a known name, and pick one of them to send the message to. Given you're pretty much going to assume you'll be overloaded, randomly picking a process with an even distribution tends to be reliable: no state communication is required, work will be shared in a roughly equal manner, and it's rather insensitive to failure.

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

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

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

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

3.3.6 How Do You Drop

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

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

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

[22] The lhttpc_lb module in this library implements it.
[23] By using `ets:update_counter/3`.
[24] https://github.com/ferd/dispcount
[25] Old papers such as Hints for Computer System Designs by Butler W. Lampson recommend dropping messages: "Shed load to control demand, rather than allowing the system to become overloaded." The paper also mentions that "A system cannot be expected to function well if the demand for any resource exceeds two-thirds of the capacity, unless the load can be characterized extremely well." adding that "The only systems in which cleverness has worked are those with very well-known loads."

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

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

3.4 Exercises

Review Questions

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

Open-ended Questions

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

第II部

Diagnosing Applications

第4章

Connecting to Remote Nodes

Interacting with a running server program is traditionally done in one of two ways. One is to do it through an interactive shell kept available by using a `screen` or `tmux` session that runs in the background and letting someone connect to it. The other is to program management functions or comprehensive configuration files that can be dynamically reloaded.

The interactive session approach is usually okay for software that runs in a strict Read-Eval-Print-Loop (REPL). The programmed management and configuration approach requires careful planning in whatever tasks you think you'll need to do, and hopefully getting it right. Pretty much all systems can try that approach, so I'll skip it given I'm somewhat more interested in the cases where stuff is already bad and no function exists for it.

Erlang uses something closer to an "interactor" than a REPL. Basically, a regular Erlang virtual machine does not need a REPL, and will happily run byte code and stick with that, no shell needed. However, because of how it works with concurrency and multiprocessing, and good support for distribution, it is possible to have in-software REPLs that run as arbitrary Erlang processes.

This means that, unlike a single screen session with a single shell, it's possible to have as many Erlang shells connected and interacting with one virtual machine as you want at a time[1].

Most common usages will depend on a cookie being present on the two nodes you want to connect together[2], but there are ways to do it that do not include it. Most usages will also require the use of named nodes, and all of them will require *a priori* measures to make sure you can contact the node.

[1] More details on the mechanisms at http://ferd.ca/repl-a-bit-more-and-less-than-that.html

[2] More details at http://learnyousomeerlang.com/distribunomicon#cookies or http://www.erlang.org/doc/reference_manual/distributed.html#id83619

4.1 Job Control Mode

The Job Control Mode (JCL mode) is the menu you get when you press `^G` in the Erlang shell. From that menu, there is an option allowing you to connect to a remote shell:

```
(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>
```

When that happens, the local shell runs all the line editing and job management locally, but the evaluation is actually done remotely. All output coming from said remote evaluation will be forwarded to the local shell.

To quit the shell, go back in the JCL mode with `^G`. This job management is, as I said, done locally, and it is thus safe to quit with `^G q`:

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

You may choose to start the initial shell in hidden mode (with the argument `-hidden`) to avoid connecting to an entire cluster automatically.

4.2 Remsh

There's a mechanism entirely similar to the one available through the JCL mode, although invoked in a different manner. The entire JCL mode sequence can by bypassed by starting the shell as follows for long names:

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

And as follows for short names:

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

All other Erlang arguments (such as `-hidden` and `-setcookie $COOKIE`) are also valid. The underlying mechanisms are the same as when using JCL mode, but the initial shell is started remotely instead of locally (JCL is still local). `^G` remains the safest way to exit the remote shell.

4.3 SSH Daemon

Erlang/OTP comes shipped with an SSH implementation that can both act as a server and a client. Part of it is a demo application providing a remote shell working in Erlang.

To get this to work, you usually need to have your keys to have access to SSH stuff remotely in place already, but for quick test purposes, you can get things working by doing:

```
$ 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>}
```

I've only set a few options here, namely `system_dir`, which is where the host files are, and `user_dir`, which contains SSH configuration files. There are plenty of other options available to allow for specific passwords, customize handling of public keys, and so on[3].

To connect to the daemon, any SSH client will do:

[3] Complete instructions with all options to get this set up are available at http://www.erlang.org/doc/man/ssh.html#daemon-3.

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

And with this you can interact with an Erlang installation without having it installed on the current machine. Just disconnecting from the SSH session (closing the terminal) will be enough to leave. *Do not run* functions such as `q()` or `init:stop()`, which will terminate the remote host.[4]

If you have trouble connecting, you can add the `-oLogLevel=DEBUG` option to `ssh` to get debug output.

4.4 Named Pipes

A little known way to connect with an Erlang node that requires no explicit distribution is through named pipes. This can be done by starting Erlang with `run_erl`, which wraps Erlang in a named pipe[5]:

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

The first argument is the name of the file that will act as the named pipe. The second one is where logs will be saved[6].

To connect to the node, you use the `to_erl` program:

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

1>
```

And the shell is connected. Closing stdio (with `^D`) will disconnect from the shell while leaving it running.

[4] This is true for all methods of interacting with a remote Erlang node.

[5] `"erl"` is the command being run. Additional arguments can be added after it. For example `"erl +K true"` will turn kernel polling on.

[6] Using this method ends up calling fsync for each piece of output, which may give quite a performance hit if a lot of IO is taking place over standard output

4.5 Exercises

Review Questions

1. What are the 4 ways to connect to a remote node?
2. Can you connect to a node that wasn't given a name?
3. What's the command to go into the Job Control Mode (JCL)?
4. Which method(s) of connecting to a remote shell should you avoid for a system that outputs a lot of data to standard output?
5. What instances of remote connections shouldn't be disconnected using `^G`?
6. What command(s) should never be used to disconnect from a session?
7. Can all of the methods mentioned support having multiple users connected onto the same Erlang node without issue?

第5章

Runtime Metrics

One of the best selling points of the Erlang VM for production use is how transparent it can be for all kinds of introspection, debugging, profiling, and analysis at run time.

The advantage of having these runtime metrics accessible programmatically is that building tools relying on them is easy, and building automation for some tasks or watchdogs is equally simple[1]. Then, in times of need, it's also possible to bypass the tools and go direct to the VM for information.

A practical approach to growing a system and keeping it healthy in production is to make sure all angles are observable: in the large, and in the small. There's no generic recipe to tell in advance what is going to be normal or not. You want to keep a lot of data and to look at it from time to time to form an idea about what your system looks like under normal circumstances. The day something goes awry, you will have all these angles you've grown to know, and it will be simpler to find what is off and needs fixing.

For this chapter (and most of those that follow), most of the concepts or features to be shown are accessible through code in the standard library, part of the regular OTP distribution.

However, these features aren't all in one place, and can make it too easy to shoot yourself in the foot within a production system. They also tend to be closer to building blocks than usable tools.

Therefore, to make the text lighter and to be more usable, common operations have been regrouped in the `recon`[2] library, and are generally production-safe.

---

[1] Making sure your automated processes don't run away and go overboard with whatever corrective actions they take is more complex

[2] http://ferd.github.io/recon/

5.1 Global View

For a view of the VM in the large, it's useful to track statistics and metrics general to the VM, regardless of the code running on it. Moreover, you should aim for a solution that allows long-term views of each metric — some problems show up as a very long accumulation over weeks that couldn't be detected over small time windows.

Good examples for issues exposed by a long-term view include memory or process leaks, but also could be regular or irregular spikes in activities relative to the time of the day or week, which can often require having months of data to be sure about it.

For these cases, using existing Erlang metrics applications is useful. Common options are:

- `folsom`[3] to store metrics in memory within the VM, whether global or app-specific..
- `vmstats`[4] and `statsderl`[5], sending node metrics over to graphite through `statsd`[6].
- `exometer`[7], a fancy-pants metrics system that can integrate with `folsom` (among other things), and a variety of back-ends (graphite, collectd, `statsd`, Riak, SNMP, etc.). It's the newest player in town
- `ehmon`[8] for output done directly to standard output, to be grabbed later through specific agents, splunk, and so on.
- custom hand-rolled solutions, generally using ETS tables and processes periodically dumping the data.[9]
- or if you have nothing and are in trouble, a function printing stuff in a loop in a shell[10].

It is generally a good idea to explore them a bit, pick one, and get a persistence layer that will let you look through your metrics over time.

5.1.1 Memory

The memory reported by the Erlang VM in most tools will be a variant of what is reported by `erlang:memory()`:

---

[3] https://github.com/boundary/folsom
[4] https://github.com/ferd/vmstats
[5] https://github.com/lpgauth/statsderl
[6] https://github.com/etsy/statsd/
[7] https://github.com/Feuerlabs/exometer
[8] https://github.com/heroku/ehmon
[9] Common patterns may fit the `ectr` application, at https://github.com/heroku/ectr
[10] The `recon` application has the function `recon:node_stats_print/2` to do this if you're in a pinch

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

This requires some explaining.

First of all, all the values returned are in bytes, and they represent memory *allocated* (memory actively used by the Erlang VM, not the memory set aside by the operating system for the Erlang VM). It will sooner or later look much smaller than what the operating system reports.

The `total` field contains the sum of the memory used for `processes` and `system` (which is incomplete, unless the VM is instrumented!). `processes` is the memory used by Erlang processes, their stacks and heaps. `system` is the rest: memory used by ETS tables, atoms in the VM, refc binaries[11], and some of the hidden data I mentioned was missing.

If you want the total amount of memory owned by the virtual machine, as in the amount that will trip system limits (`ulimit`), this value is more difficult to get from within the VM. If you want the data without calling `top` or `htop`, you have to dig down into the VM's memory allocators to find things out.[12]

Fortunately, recon has the function `recon_alloc:memory/1` to figure it out, where the argument is:

- `used` reports the memory that is actively used for allocated Erlang data;
- `allocated` reports the memory that is reserved by the VM. It includes the memory used, but also the memory yet-to-be-used but still given by the OS. This is the amount you want if you're dealing with `ulimit` and OS-reported values.
- `unused` reports the amount of memory reserved by the VM that is not being allocated. Equivalent to `allocated - used`.
- `usage` returns a percentage (0.0 .. 1.0) of used over allocated memory ratios.

[11] See Section 7.2
[12] See Section 7.3.2

There are additional options available, but you'll likely only need them when investigating memory leaks in chapter 7

5.1.2 CPU

Unfortunately for Erlang developers, CPU is very hard to profile. There are a few reasons for this:

- The VM does a lot of work unrelated to processes when it comes to scheduling — high scheduling work and high amounts of work done by the Erlang processes are hard to characterize.
- The VM internally uses a model based on *reductions*, which represent an arbitrary number of work actions. Every function call, including BIFs, will increment a process reduction counter. After a given number of reductions, the process gets descheduled.
- To avoid going to sleep when work is low, the threads that control the Erlang schedulers will do busy looping. This ensures the lowest latency possible for sudden load spikes. The VM flag `+sbwt none|very_short|short|medium|long|very_long` can be used to change this value.

These factors combine to make it fairly hard to find a good absolute measure of how busy your CPU is actually running Erlang code. It will be common for Erlang nodes in production to do a moderate amount of work and use a lot of CPU, but to actually fit a lot of work in the remaining place when the workload gets higher.

The most accurate representation for this data is the scheduler wall time. It's an optional metric that needs to be turned on by hand on a node, and polled at regular intervals. It will reveal the time percentage a scheduler has been running processes and normal Erlang code, NIFs, BIFs, garbage collection, and so on, versus the amount of time it has spent idling or trying to schedule processes.

The value here represents *scheduler utilization* rather than CPU utilization. The higher the ratio, the higher the workload.

While the basic usage is explained in the Erlang/OTP reference manual[13], the value can be obtained by calling recon:

```
1> recon:scheduler_usage(1000).
[{1,0.9919596133421669},
 {2,0.9369579039389054},
```

[13] http://www.erlang.org/doc/man/erlang.html#statistics_scheduler_wall_time

```
 {3,1.9294092120138725e-5},
 {4,1.2087551402238991e-5}]
```

The function `recon:scheduler_usage(N)` will poll for `N` milliseconds (here, 1 second) and output the value of each scheduler. In this case, the VM has two very loaded schedulers (at 99.2% and 93.7% repectively), and two mostly unused ones at far below 1%. Yet, a tool like `htop` would report something closer to this for each core:

```
[|||||||||||||||||||||||||     70.4%]
[|||||||                       20.6%]
[|||||||||||||||||||||||||||||100.0%]
[||||||||||||||||              40.2%]
```

The result being that there is a decent chunk of CPU usage that would be mostly free for scheduling actual Erlang work (assuming the schedulers are busy waiting more than trying to select tasks to run), but is being reported as busy by the OS.

Another interesting behaviour possible is that the scheduler usage may show a higher rate (1.0) than what the OS will report. Schedulers waiting for os resources are considered utilized as they cannot handle more work. If the OS itself is holding up on non-CPU tasks it is still possible for Erlang's schedulers not to be able to do more work and report a full ratio.

These behaviours may especially be important to consider when doing capacity planning, and can be better indicators of headroom than looking at CPU usage or load.

5.1.3 Processes

Trying to get a global view of processes is helpful when trying to assess how much work is being done in the VM in terms of *tasks*. A general good practice in Erlang is to use processes for truly concurrent activities — on web servers, you will usually get one process per request or connection, and on stateful systems, you may add one process per-user — and therefore the number of processes on a node can be used as a metric for load.

Most tools mentioned in section 5.1 will track them in one way or another, but if the process count needs to be done manually, calling the following expression is enough:

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

Tracking this value over time can be extremely helpful to try and characterize load or detect process leaks, along with other metrics you may have around.

5.1.4 Ports

In a manner similar to processes, *Ports* should be considered. Ports are a datatype that encompasses all kinds of connections and sockets opened to the outside world: TCP sockets, UDP sockets, SCTP sockets, file descriptors, and so on.

There is a general function (again, similar to processes) to count them: `length(erlang:ports())`. However, this function merges in all types of ports into a single entity. Instead, one can use `recon` to get them sorted by type:

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

This list contains the types and the count for each type of port. The type name is a string and is defined by the Erlang VM itself.

All the `*_inet` ports are usually sockets, where the prefix is the protocol used (TCP, UDP, SCTP). The `efile` type is for files, while `"0/1"` and `"2/2"` are file descriptors for standard I/O channels (*stdin* and *stdout*) and standard error channels (*stderr*), respectively.

Most other types will be given names of the driver they're talking to, and will be examples of *port programs*[14] or *port drivers*[15].

Again, tracking these can be useful to assess load or usage of a system, detect leaks, and so on.

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

[14] http://www.erlang.org/doc/tutorial/c_port.html
[15] http://www.erlang.org/doc/tutorial/c_portdriver.html

a process in a weird state? Maybe it needs tracing[16]! 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])`[17]. Here are the commonly used keys[18]:

Meta

- `dictionary` returns all the entries in the process dictionary[19]. 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.[20]
- `registered_name` if the process has a name (as registered with `erlang:register/2`), it is given here.
- `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;

[16] See Chapter 9

[17] In cases where processes contain sensitive information, data can be forced to be kept private by calling `process_flag(sensitive, true)`

[18] For *all* options, look at http://www.erlang.org/doc/man/erlang.html#process_info-2

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

[20] See http://learnyousomeerlang.com/building-otp-applications#the-application-behaviour and http://erlang.org/doc/apps/stdlib/io_protocol.html for more details.

`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[21] 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 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*.

---

[21] See Section 7.2

`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.[22]

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}}]},
 {<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}}]}]
```

[22] 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.

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[23] 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.

`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

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

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[24]. This return the size of this queue, in bytes.

Type-Specific

Inet Ports Returns inet-specific data, including statistics[25], the local address and port number for the socket (`sockname`), and the inet options used[26]

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>]},
           {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,...},
```

[24] 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.

[25] http://www.erlang.org/doc/man/inet.html#getstat-1

[26] http://www.erlang.org/doc/man/inet.html#setopts-2

```
                 {...}|...]},
          {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[27] with the `recon:inet_window(Attribute, Count, Milliseconds)` function:

```
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

[27] See the explanations for the `recon:proc_window/3` in the preceding subsection

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 Exercises

Review Questions

1. What kind of values are reported for Erlang's memory?
2. What's a valuable process-related metric for a global view?
3. What's a port, and how should it be monitored globally?
4. Why can't you trust `top` or `htop` for CPU usage with Erlang systems? What's the alternative?
5. Name two types of signal-related information available for processes
6. How can you find what code a specific process is running?
7. What are the different kinds of memory information available for a specific process?
8. How can you know if a process is doing a lot of work?
9. Name a few of the values that are dangerous to fetch when inspecting processes in a production system.
10. What are some features provided to OTP processes through the sys module?
11. What kind of values are available when inspecting inet ports?
12. How can you find the type of a port (Files, TCP, UDP)?

Open-ended Questions

1. Why do you want a long time window available on global metrics?
2. Which would be more appropriate between `recon:proc_count/2` and `recon:proc_window/3` to find issues with:
   (a) Reductions
   (b) Memory
   (c) Message queue length
3. How can you find information about who is the supervisor of a given process?
4. When should you use `recon:inet_count/2`? `recon:inet_window/3`?
5. What could explain the difference in memory reported by the operating system and the memory functions in Erlang?
6. Why is it that Erlang can sometimes look very busy even when it isn't?

7. How can you find what proportion of processes on a node are ready to run, but can't be scheduled right away?

Hands-On

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

1. What's the system memory?
2. Is the node using a lot of CPU resources?
3. Is any process mailbox overflowing?
4. Which chatty process (`council_member`) takes the most memory?
5. Which chatty process is eating the most CPU?
6. Which chatty process is consuming the most bandwidth?
7. Which chatty process sends the most messages over TCP? The least?
8. Can you find out if a specific process tends to hold multiple connections or file descriptors open at the same time on a node?
9. Can you find out which function is being called by the most processes at once on the node right now?

第6章

Reading Crash Dumps

Whenever an Erlang node crashes, it will generate a crash dump[1].

The format is mostly documented in Erlang's official documentation[2], and anyone willing to dig deeper inside of it will likely be able to figure out what data means by looking at that documentation. There will be specific data that is hard to understand without also understanding the part of the VM they refer to, but that might be too complex for this document.

The crash dump is going to be named `erl_crash.dump` and be located wherever the Erlang process was running by default. This behaviour (and the file name) can be overridden by specifying the `ERL_CRASH_DUMP` environment variable[3].

6.1 General View

Reading the crash dump will be useful to figure out possible reasons for a node to die *a posteriori*. One way to get a quick look at things is to use recon's `erl_crashdump_analyzer.sh`[4] and run it on a crash dump:

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

Slogan: eheap_alloc: Cannot allocate 2733560184 bytes of memory
```

[1] If it isn't killed by the OS for violating ulimits while dumping or didn't segfault.

[2] http://www.erlang.org/doc/apps/erts/crash_dump.html

[3] Heroku's Routing and Telemetry teams use the `heroku_crashdumps` app to set the path and name of the crash dumps. It can be added to a project to name the dumps by boot time and put them in a pre-set location

[4] https://github.com/ferd/recon/blob/master/script/erl_crashdump_analyzer.sh

```
(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
```

This data dump won't point out a problem directly to your face, but will be a good clue as to where to look. For example, the node here ran out of memory and had 11079 Mb out of 15 Gb used (I know this because that's the max instance size we were using!) This can be a symptom of:

- memory fragmentation;
- memory leaks in C code or drivers;

- lots of memory that got to be garbage-collected before generating the crash dump[5].

More generally, look for anything surprising for memory there. Correlate it with the number of processes and the size of mailboxes. One may explain the other.

In this particular dump, one process had 5 million messages in its mailbox. That's telling. Either it doesn't match on all it can get, or it is getting overloaded. There are also dozens of processes with hundreds of messages queued up — this can point towards overload or contention. It's hard to have general advice for your generic crash dump, but there still are a few pointers to help figure things out.

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:

```
$ awk -v threshold=10000 -f queue_fun.awk /path/to/erl_crash.dump
MESSAGE QUEUE LENGTH: CURRENT FUNCTION
======================================
10641: io:wait_io_mon_reply/2
12646: io:wait_io_mon_reply/2
32991: io:wait_io_mon_reply/2
2183837: io:wait_io_mon_reply/2
730790: io:wait_io_mon_reply/2
80194: io:wait_io_mon_reply/2
...
```

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.

[5] Notably here is reference-counted binary memory, which sits in a global heap, but ends up being garbage-collected before generating the crash dump. The binary memory can therefore be underreported. See Chapter 7 for more details

6.3 Too Many (or too few) Processes

The process count is mostly useful when you know your node's usual average count[6], in order to figure if it's abnormal or not.

A count that is higher than normal may reveal a specific leak or overload, depending on applications.

If the process count is extremely low compared to usual, see if the node terminated with a slogan like:

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

In such a case, the issue is that a specific application (`<AppName>`) has reached its maximal restart frequency within its supervisors, and that prompted the node to shut down. Error logs that led to the cascading failure should be combed over to figure things out.

6.4 Too Many Ports

Similarly to the process count, the port count is simple and mostly useful when you know your usual values[7].

A high count may be the result of overload, Denial of Service attacks, or plain old resource leaks. Looking at the type of port leaked (TCP, UDP, or files) can also help reveal if there was contention on specific resources, or if the code using them is just wrong.

6.5 Can't Allocate Memory

These are by far the most common types of crashes you are likely to see. There's so much to cover, that Chapter 7 is dedicated to understanding them and doing the required debugging on live systems.

In any case, the crash dump will help figure out what the problem was after the fact. The process mailboxes and individual heaps are usually good indicators of issues. If you're running out of memory without any mailbox being outrageously large, look at the processes heap and stack sizes as returned by the recon script.

In case of large outliers at the top, you know some restricted set of processes may be eating

[6] See subsection 5.1.3 for details

[7] See subsection 5.1.4 for details

up most of your node's memory. In case they're all more or less equal, see if the amount of memory reported sounds like a lot.

If it looks more or less reasonable, head towards the "Memory" section of the dump and check if a type (ETS or Binary, for example) seems to be fairly large. They may point towards resource leaks you hadn't expected.

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?

第7章

Memory Leaks

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.[1]

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[2] or figuring the way to add your own SAX parser that can be safe[3].

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.[4] 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.

[1] See Chapter 4 if you need help to connect to a running node

[2] I don't dislike exml or erlsom

[3] See Ulf Wiger at http://erlang.org/pipermail/erlang-questions/2013-July/074901.html

[4] 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[5]. 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.[6]

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.

---

[5] http://www.erlang.org/doc/man/HiPE_app.html

[6] 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[7] 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.

[7] 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[8]. 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.

[8] 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[9], 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).
```

[9] 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`[10] if keeping only a small fragment (usually less than 64 bytes) of a larger binary;[11]
- 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.

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

[11] 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.[12] 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.[13]

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:[14]

1. `temp_alloc`: does temporary allocations for short use cases (such as data living within a single C function call).
2. `eheap_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.

---

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

[13] More information is available at http://ferd.github.io/recon/recon_alloc.html

[14] The complete list of where each data type lives can be found in erts/emulator/beam/erl_alloc.types

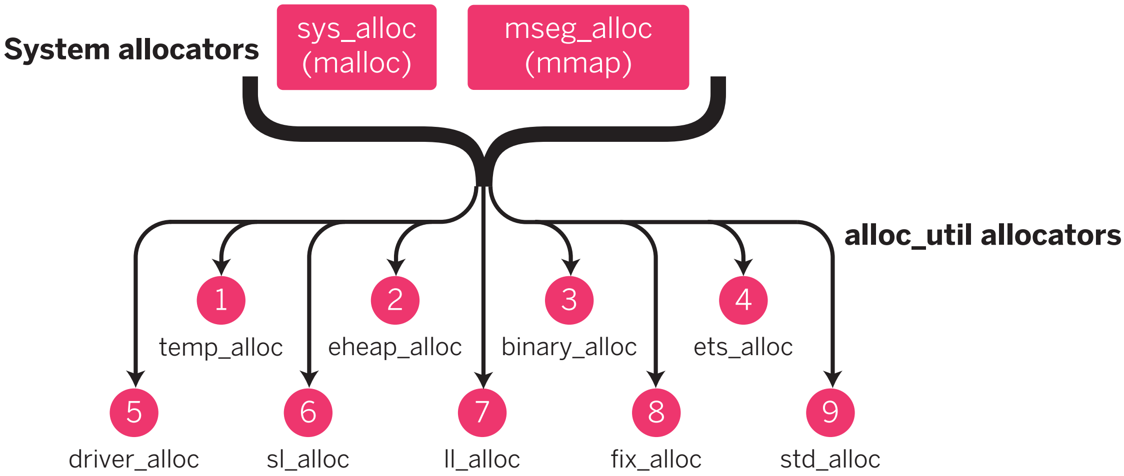


図 7.1 Erlang's Memory allocators and their hierarchy. Not shown is the special *super carrier*, optionally allowing to pre-allocate (and limit) all memory available to the Erlang VM since R16B03.

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.

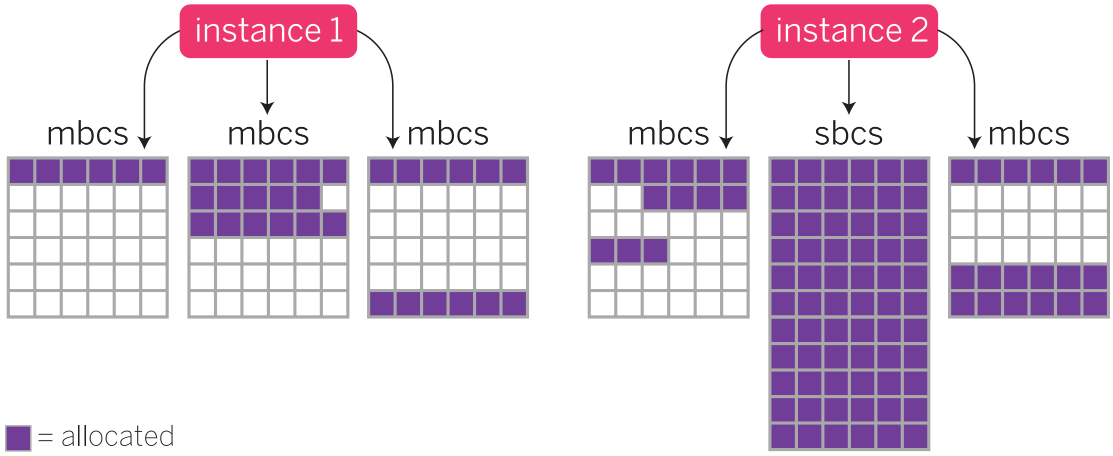


図 7.2　Example memory allocated in a specific sub-allocator

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`)[15], 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`[16] 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`[17] 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`)

[15] http://erlang.org/doc/man/erts_alloc.html#M_sbct
[16] http://erlang.org/doc/man/erts_alloc.html#M_mmsbc
[17] http://erlang.org/doc/man/erts_alloc.html#M_mmsbc

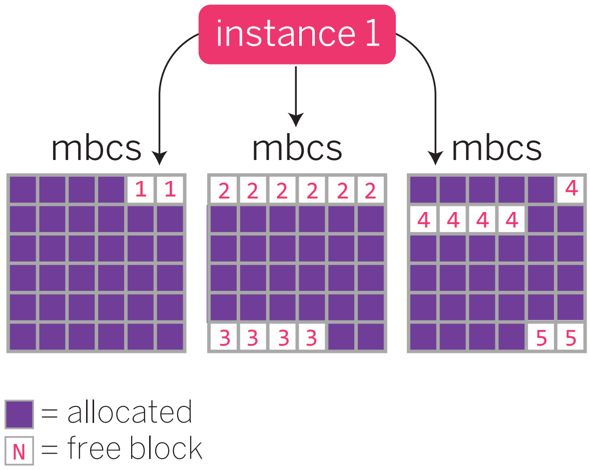


図 7.3　Example memory allocated in a specific sub-allocator

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[18]

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

[18] http://erlang.org/doc/man/erts_alloc.html#M_as

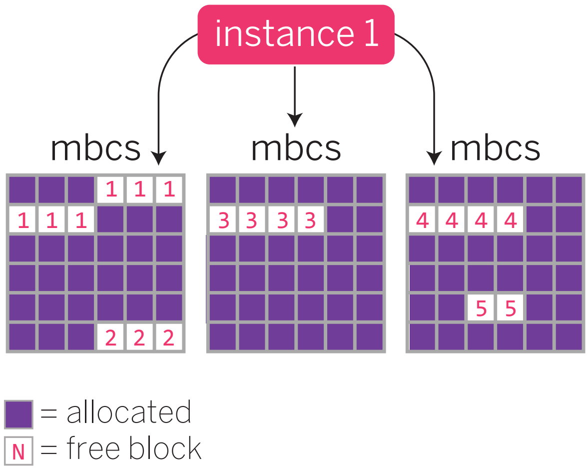


図 7.4 Example memory allocated in a specific sub-allocator

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`[19] 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 fragmentation on the VM. When an allocator gets to have one of its multiblock carriers

[19] http://www.erlang.org/doc/man/erts_alloc.html#M_mbsd

become mostly empty,[20] 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.[21] 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:

```
[                    ]
```

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

```
[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`):

```
[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.

When that happens, over 80% of the time, the only thing that happens is that the memory is

[20] The threshold is configurable through http://www.erlang.org/doc/man/erts_alloc.html#M_acul

[21] In cases this consumes too much memory, the feature can be disabled with the options `+MBacul 0`.

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[22] 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 `sbcs`, 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?

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?

[22] http://ferd.github.io/recon/recon_alloc.html

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 is 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?

第8章

CPU and Scheduler Hogs

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.[1] 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:[2]

[1] 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.

[2] 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`,[3] the oldest Erlang profiler around. It will give general percentage values and will mostly report in terms of time taken.
- `fprof`,[4] 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`,[5] 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,[6] 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[7] and current functions being run, you should be able to identify what kind of code hogs all your schedulers.

---

[3] http://www.erlang.org/doc/man/eprof.html
[4] http://www.erlang.org/doc/man/fprof.html
[5] https://github.com/proger/eflame
[6] See Subsection 5.1.2
[7] 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.[8]

We've seen how to set such a system monitor In Garbage Collection in 7.1.5, but here's a different pattern[9] 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

[8] 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.

[9] 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。[1] Erlang では開発中あるいは稼働中の本番システムの診断であっても、システムのライフサイクルのすべての場所において、トレースは有効です。

トレースを行ういくつかの Erlang プログラムがあります。

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

---

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

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

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

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

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

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[8]。BIF は *pid* **指定**と**トレースパターン**に分かれています。

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

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

特定の関数呼び出しがトレースされるかどうかを定義している箇所は、9.1 にあるように、両者の共通部分です。

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

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

---

[4] https://github.com/massemanet/eper/blob/master/doc/redbug.txt

[5] https://github.com/massemanet/eper

[6] http://ferd.github.io/recon/recon_trace.html

[7] メッセージのトレース機能は将来のバージョンでサポートされるかもしれません。ライブラリの著者は OTP を使っている時には必要性を感じておらず、またビヘイビアと特定の引数へのマッチングにより、ユーザはおよそ同じことを実現できます

[8] プロセスに機密情報が含まれている場合、`process_flag(sensitive, true)` を呼ぶことで、データを非公開にすることを強制できます

[9] http://www.erlang.org/doc/apps/erts/match_spec.html

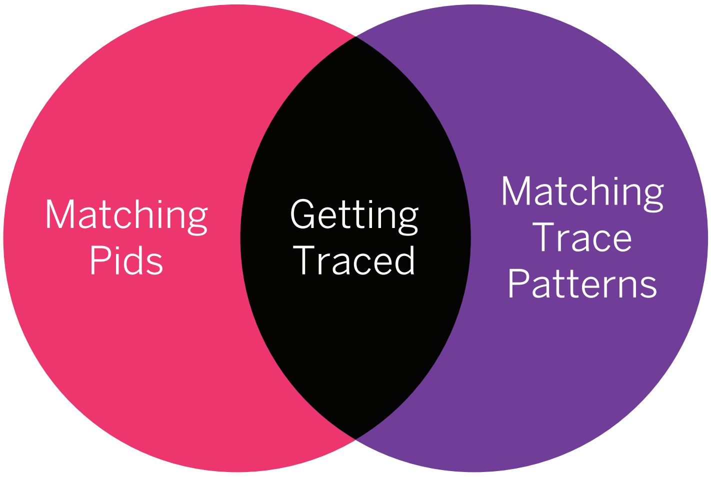


図 9.1　トレースされるのは、pid 指定とトレースパターンの交差した箇所です

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

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、あなたが遊びたいと思う面白い部分は、トレースするパターンの指定です。Recon ではいくつかの方法をサポートしています。

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

組み合わせることが可能なオプションはもう少しあります。

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

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

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

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

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

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

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

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、また `recon_trace` はそれに対し

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

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

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 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。