Stuff Goes Bad:
Erlang in Anger

Fred Hébert 著、elixir.jp 訳

2020 年 5 月 7 日
Git commit ID: 5e0f179

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。私にとってはどちらのやり方も極めて重要です。ほぼすべての環境でそれぞれに衛生状況が異なります。実行時のエラーがうまく対処されて、そのまま生き残れるような治癒の仕組みを持っているプログラミング環境は Erlang の他にほとんどありません。

---

[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 個のバグが一時的なもの (非決定的で調査するときにはなくなっていて、再実行することで問題が解決するもの) です。

Erlang 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)、

[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**

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

アプリケーションを書く

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"},
  {vsn, "1.0.0"},
  {applications, [kernel, stdlib]},
  {registered, []},
```

[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。

```
  {mod, {dispcount, []}},
  {modules, [dispcount, dispcount_serv, dispcount_sup,
             dispcount_supersup, dispcount_watcher, watchers_sup]}
]}.
```

の 2 種類です。

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

1.2.1　ライブラリアプリケーション

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

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

1.2.2　標準アプリケーション

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

1. `appname`
2. `appname_app`

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

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

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

[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。

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。

[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。

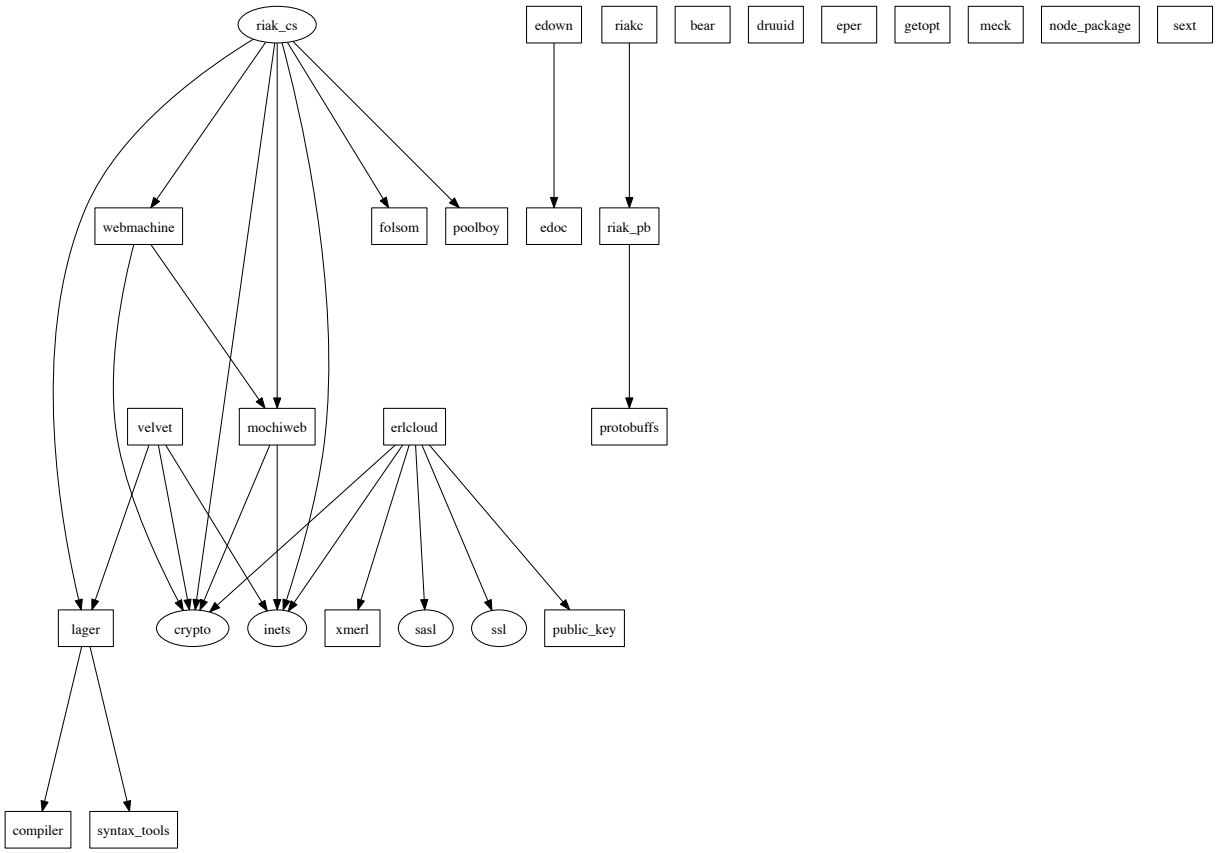


図 1.1　Basho のオープンソースクラウドライブラリである riak_cs の依存関係を表したグラフです。このグラフは kernel や stdlib といった必ず依存するようなものは除いています。楕円はアプリケーションで、四角はライブラリアプリケーションです。

1.2.3　依存関係

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

こうした依存関係を使って各アプリケーションの短い解説を見ることで、何がどこにあるかの大まかな地図を描くのに役立つでしょう。似たダイヤグラムを生成するためには、`recon` の script ディレクトリ内のツールを使って `escript script/app_deps.erl` を実行してみましょう。[6]似たダイヤグラムが

[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。

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。

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を行えるようになっています。コンパイルされたリリースが含むライブラリの数は、デフォルトの配布形式と比べて多少違いますが、自分専用の Erlang VM のコピーを持っていて、単独で起動できるようになっています。もちろん、リリースに関してはまだ話すことはありますが、一般的に OTP アプリケーションのときと同じようなやり方で中身を確認していきます。

OTP リリース内には通常、`relx.config` または `rebar.config` ファイル内の `relx` タプルがあります。ここに、どのトップレベルアプリケーションがリリースに含まれているかとパッケージ化に関するオプションが書かれています。relx を使ったリリースはプロジェクトの Wiki ページ[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。

---

[7] http://www.erlang.org/doc/apps/observer/observer_ug.html
[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

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. README によると、このアプリケーションは非決定的な振る舞いをするらしい。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。

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。

2.1 プロジェクト構造

OTP アプリケーションと OTP リリースではプロジェクトの構造が異なります。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。アプリケーションの主な構成方法はこのように 2 種類あります。

[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。

2.1.1　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成果物を置きます。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]代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の中に保存します。

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。

```
{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"}}}
 ]}.
```

git（または hg）を使ってソースコードを、あるいは hex.pm からパッケージを、幅優先探索で取得し、依存しているソフトウェアを集めます。続いて、ソースコードをコンパイルします。このとき、設定ファイ

[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。

ルの{erl_opts, List}. 項目で追加のコンパイルオプションを指定できます。[4]

rebar3 compile を呼んで、すべての依存しているソフトウェアをダウンロードし、あなたのアプリとダウンロードしたソフトウェアとをまとめてビルドします。

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**含めずに**配布しましょう。他の開発者のアプリケーションとあなたのアプリケーションが共通の依存先を持つことはよくあり、何度もそれを配布するのは意味がありません。その場にあるビルドシステム（この場合は rebar3）が重複した項目を見つけ出して、必要なものは 1 度だけしか取得しないようにしてくれるでしょう。

2.1.2 OTP リリース

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

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

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

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 Reltool 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。

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

```
1 {relx, [
2   {release, {demo, "1.0.0"},
```

[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

[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

```
    [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、その成果物は`_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、一時的なバグが発生したとき処理を再実行すれば、システムの **Mean Times Between Failures（障害間隔平均時間）**(MTBF) は 4 倍良くなると引用していました。そして、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。

この点が人々が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

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

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。最適化の一環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、スーバーバイザーは再起動を行いますが、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。

[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

```
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

何があろうとも、ノードで障害が連続して起きることはノードへの死刑宣告ではありません。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、`application:start(Name, Type)` と手動で起動するときに引数で指定するか、リリース内の設定ファイルに書くことで選択可能です。

- `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。これは自分のスーバ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。

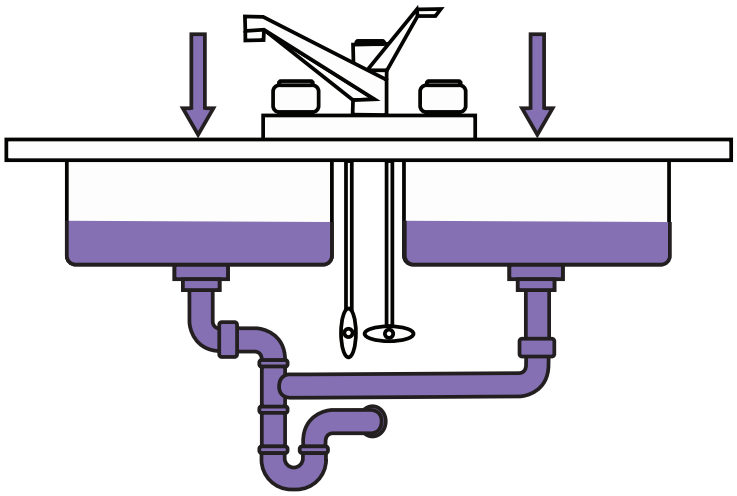
1. リリースにホストされている main アプリケーションを抜き出して独立したアプリケーションにして、他のプロジェクトにインクルード可能にしてください。
2. アプリケーションをどこかにホストしてください。(GitHub、Bitbukect、ローカルサーバーなど) そしてそのアプリケーションに依存するリリースをビルドしてください。
3. main アプリケーションのワーカー (`council_member`) はサーバーの起動とそこへの接続を自身の `init/1` 関数で行っています。この接続を init 関数の外で行なえますか。このアプリの用途においてそうすることの利点はあるでしょうか。

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、誰の責任か解明することが重要です。誰かがシンクに水を入れすぎましたか？ 下水道が渋滞していますか？ あまりにも小さなパイプを設計しましたか？

どのキューが爆発したかを判断することは必ずしも困難ではありません。これはクラッシュダン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、

[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。

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 (圧縮、より良いアルゴリズムとデータ表現、キャッシングなどを使って) 軽量化しようとします。

それでも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こと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

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

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[3]。

`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。私達のテストケースでは概ねうまく動いていて、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。[4]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。

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ったことでした。この 10 ミリ秒は接続試行が断念されるまでの時間です。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自体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[5]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

[3] 訳注: 現在 lager はコミュニティーにより https://github.com/erlang-lager/lager として開発、メンテナンスされています
[4] 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。
[5] 本番環境で事実に基づいて過負荷になるもの

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。[6]

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す。あなたが十分に幸運であれば、監視ツール[7]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[8]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　入力を制限する

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

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

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

[6] `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。

[7] 5.1 の節を見ましょう

[8] 5.2.1 の節を見ましょう

ク越しのサービスでしょう。

これはつまり同期的な動作をシステムの深層部に導入すると、おそらく各階層ごとにバックプレッシャーに対応し、システムの周辺部までたどり着いてユーザーに「もう少しゆっくりしてください」と言えるようになるまで対応する必要が出て来るということです。このパターンを理解した開発者はしばしばユーザーごとにシステムのエントリーポイントに API 制限を設けようとします。[9]特に基本的なシステムに対するサービスのクオリティ（QoS）を保証でき、リソースを公平に（あるいは不公平に）望んだとおりにリソースを割り当てられるので、これは正当なやり方です。

3.2.1　タイムアウトはどれくらいの長さであるべきか

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

この問題を最もうまく表現しようと思うと、最初のタイマーがシステムの周辺部で開始されるけれども、致命的な処理はシステムの深層部で起きているというような具合です。つまり、システムの周辺部にあるタイマーはシステムの深層部にあるタイマーよりも長い時間待つ必要があるでしょう。ただし、深層部で処理が成功していたとしても周辺部では操作はタイムアウトしたということにしたいのであれば別です。

この状況を脱する簡単な方法はタイムアウトを無期限にすることです。Pat Helland[10]がこれに対して興味深い回答をしています。

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

つまり、究極的にはケースバイケースな問題です。多くの場合、フロー制御には異なる機構を使うほうがより実用的でしょう。[11]

3.2.2　許可を求める

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

---

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

[10] Idempotence is Not a Medical Condition 2012 年 4 月 14 日発行

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

あって、メモリ、CPU、全体の負荷、呼び出し回数の上限、並行処理、応答時間、これらの組み合わせ、などです。

SafetyValve[12]アプリケーションはシステム全体に及ぶフレームワークで、バックプレッシャーが必要だとわかっているときに使えます。

サービスやシステムの障害により関係しそうなユースーケースには、多くのサーキットブレーカーアプリケーションがあります。たとえば `breaky`[13]、`fuse`[14]、Klarna の `circuit_breaker`[15]といった具合です。

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

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

3.2.3　ユーザーが見るもの

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

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

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

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

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

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

[12] https://github.com/jlouis/safetyvalve
[13] https://github.com/mmzeeman/breaky
[14] https://github.com/jlouis/fuse
[15] https://github.com/klarna/circuit_breaker

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

3.3 データの破棄

Erlang システム以外から流入速度を落とせず、かつスケールアップもできない場合、データをドロップさせるか、クラッシュ（結局それでも大抵の場合は、処理中のデータをより乱暴にドロップ）させる必要があります。

これは全ての人が目を背けたくなる悲しい現実です。プログラマーや、ソフトウェアエンジニア、そしてコンピュータサイエンティストは、不要なデータは消去し、有用なデータはすべて保持するよう学んできています。地道な最適化をすることにより、Erlang システムは入力データを問題なく処理できる状態になります。

しかし、Erlang システム自身が全てのデータを十分な速度で処理できたとしても、Erlang システムの**後ろ**に控えるコンポーネントがブロックし、出力よりもデータ入力の方が速くなることもあります。

受信するデータ量を制限する選択肢がない場合は、クラッシュを避けるためにメッセージをドロップする必要があります。

3.3.1 ランダムドロップ

ランダムにメッセージをドロップすることは、これらの対処を行う最も簡単な方法であり、シンプルがゆえに最も堅牢な実装でもあります。

その中身は、0.0 から 1.0 の間のある閾値を定義し、その範囲の乱数を取り出すという仕組みになっています。

```
-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.
```

送信するメッセージの 95 %を保持することを目指す場合、`case drop:random(0.95) of true -> send(); false`

の呼び出しによって承認できます。メッセージをドロップする際に特別な処理を行う必要がない場合、より短く `drop:random(0.95) andalso send()` で書けます。[16]

`maybe_seed()` 関数は、`now()` (グローバルロックを要求する単調関数) を高頻度で呼び出すことを避けるため、過去に呼び出されていない場合、有効なシードがプロセスディクショナリに存在していることをチェックし、それを有用なものとして使用します。

しかし、この方法には 1 つの「落とし穴」があります。ランダムドロップは、キュー（レシーバ）側ではなくプロデューサ側で行うのが理想的なのです。キューの過負荷を避ける最善の方法は、そもそもデータを送信しないことです。Erlang には制限付きのメールボックスがないため、受信プロセスのメッセージドロップだけが唯一、プロセスの高速動作、メッセージ除去への試み、処理を行うためにスケジューラに追いつくこと、などを保証します。

その一方で、プロデューサ側でのメッセージドロップは、すべてのプロセス間で均等に分散されることを保証します。

このメッセージドロップの方法で、処理するプロセスまたは特定のモニタプロセス[17]が、ETS テーブルまたは `application:set_env/3` の値を使用して、乱数とともに使用されるしきい値を動的に増減させる、という興味深い最適化をすることができます。これにより、負荷に基づいてドロップされるメッセージの数を制御したり、`application:get_env/2` を使用すれば、どんなプロセスでも効率的に構成データを取得できます。

同様の技術を使用すれば、処理する側ですべてを選別するのではなく、メッセージに別々の優先順位をつけて異なるドロップ比率を実装することもできます。

3.3.2　キューバッファ

キューバッファはメッセージを捨てる時に、ランダムではなく、より制御された方法で行いたい時の代替手段となります。ある程度の量を捨てる必要がある (量の変動が少ない) ストリームではなく、負荷がバーストすることが予想されているときには、特にそうです。

プロセスの通常のメールボックスには一定量のキューがありますが、一般的には出来るだけ早く **全ての** メッセージを取り出す必要があるでしょう。安全のためには、キューバッファには 2 種類のプロセスが必要です。

- 通常のプロセス (例えば `gen_server`)
- メッセージをバッファする以外は何もしない新しいプロセス。外部からのメッセージはこのプロセスに送ります。

バッファするプロセスが、自身のメールボックスから取り出せる全てのメッセージを取り除き、自身で管理できるキュー用のデータ構造[18]の中に入れることで、これを実現できます。サーバがさらに処理を行うこ

[16] 訳注: random module は OTP18 で代替の rand module が追加され OTP19 以降では非推奨になっているため、OTP19 以降は rand を使った方が良いでしょう。

[17] `process_info(Pid, message_queue_len)` のような、ヒューリスティックスを使用して特定のプロセスの負荷をチェックするプロセスは、どんなものでもモニタにすることができます。

[18] Erlang の `queue` モジュールは、このような用途のバッファに適した、完全に関数型のキューデータ構造です

とができるときには、バッファプロセスに対していつでも、処理可能な量だけのメッセージを送るよう要求することができます。バッファプロセスはキューからメッセージを取り出してサーバに送り、またメッセージを集める処理に戻ります。

キューが一定のサイズ[19]よりも大きくなり新しいメッセージを受け取ったときにはいつでも、最も古いメッセージを取り出し、新しいメッセージをキューに入れ、古いメッセージを消していくことができます[20]。

これにより、受信したメッセージの総数を適切なサイズに保ち、ある程度高負荷にも耐えるリングバッファに似たような機能を提供できます。

PO Box[21]ライブラリはこのようなキューバッファを提供しています。

3.3.3 スタックバッファ

キューバッファのような制御も欲しいが、低レイテンシのための重要な要件があるときには、スタックバッファが理想的です。

スタックをバッファとして利用するには、キューバッファの時のように二つのプロセスが必要となりますが、キュー構造の代わりにリスト[22]が使われます。

低レイテンシのためにスタックバッファが特に優れている理由は、バッファブロート[23]に似た問題と関係しています。キューバッファにあるメッセージの処理が遅れると、それがほんの少量のメッセージだったとしても、キューの中にあるすべてのメッセージにミリ秒単位の待ち時間が発生します。しだいにメッセージは古くなりすぎて、すべてのバッファを削除する必要が出てくるでしょう。

その一方で、スタックはある一定数の要素だけが待たされて、新しい方からタイムリーにサーバで処理されます。

スタックが一定サイズを超えて成長した場合や、その中の要素が QoS 要求を満たすには遅すぎると気づいた時は、ただスタックの残りを破棄して、続きを始めれば良いです。**PO Box** もこのようなバッファ実装を行っていますね。

スタックバッファの主な欠点はメッセージが必ずしも投稿された順番どおりに行われないことです — 独立したタスクには向きますが、イベントが順番通りだと期待するなら、一日を無駄にすることでしょう。

3.3.4 時間に敏感なバッファ

もし、古いイベントに対して、古くなりすぎる**前に**反応する必要があるのなら、物事はより複雑になります。イベントが古いかどうかは、スタックの奥深くまで毎回確認しなければわかりませんし、スタック

[19] キューのサイズを計算する場合、キューを毎回イテレートするよりも、メッセージの送受信の度に加算・減算されるカウンターを使うことが望ましいです。多少のメモリが必要になりますが、カウントの負荷を分散させられますし、バッファのメールボックスが突然あふれるのを防いだりあるいは予測するのに役立ちます

[20] この他にも、古いメッセージが重要だと思えるときには、最も新しいメッセージを取り出して、最も古いメッセージをキューに残すようなキューを作成することができます

[21] https://github.com/ferd/pobox にあります。このライブラリは Heroku の大規模な本番環境で長い間使われてきており、成熟していると考えられています

[22] どうでも良いことですが、Erlang ではリスト**が**スタックです。Erlang リストは O(1) の計算量で push と pop 操作を提供していて、非常に速いです。

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

を底からものをイベントを取り出すのは普通、非効率です。この場合、バケツ — それぞれが担当時間帯を持っている、複数のスタック — を使った面白いアプローチが使えます。リクエストが QoS 制約として古くなりすぎたら、その時間帯のバケツ一つをまるごと削除します。バッファすべては削除しません。

多数の利益のために一部リクエストを非常に悪くする— 中央値は良いが、下位 1% がひどいことになっている — というのは直感に反するかもしれませんが、これはとにかくメッセージを捨てたいという状態で起こるもので、低レイテンシがどうしても求められる場合には、望ましい方法です。

3.3.5　定量負荷の取り扱い

定量負荷に対しては、新しい方法が必要でしょう。キューもバッファも、時々 (たとえそれが、かなりの長い時間続くものだったとしても) 発生する負荷に対しては良い方法ですが、どちらも、入力速度が徐々に落ち着き処理が追いつくことが期待できる場合に信頼できる方法です。

一つのプロセスに大量のメッセージを送ろうとすると、過負荷になってしまうことがよく問題になります。一般的に、この場合には二つのアプローチがあります。

- バッファとして機能するプロセスをたくさん持ち、それらの間で負荷分散を行う（スケールアウトさせる方法)
- ロックとカウンタの仕組みとして、ETS テーブルを使う（入力を減らす方法）

一般的に ETS テーブルは、１秒間にプロセスよりかなりの多くのリクエストを処理できますが、サポートする操作はより基本的なものだけです。あなたが期待できる操作はたいてい、単純な読み込み、カウンタへのアトミックな追加と削除くらいでしょう。

どちらのアプローチを取るにしても、ETS テーブルは必要です。

ひとつめのアプローチは、一般論で言えば、プロセスレジストリを使うだけでも動作します。負荷を分散するのに `N` 個のプロセスを用意し、プロセスそれぞれに known name をつけ、メッセージを送る際はその中の一つを選択すれば良いのです。高負荷がかかることを十分に想定するなら、一様ランダムにプロセスを選択するようにすれば頼もしいですね。プロセス同士で状態を教え合う必要はありませんし、負荷はおおよそ等しく分配され、失敗に対しても鈍感です。

実際にはしかし、プロセス名のために atom を動的に生成するのは避けたいので、`read_concurrency` を `true` に設定した ETS テーブルにワーカーを保存することをおすすめします。ひと手間かかりますが、後々ワーカー数の更新が柔軟に行えるでしょう。

これに似たアプローチは、本書で以前に出てきた `lhttpc`[24] ライブラリでも、ドメインごとの負荷を分けるために使われています。

ふたつめのアプローチ、つまりはカウンタとロックを使う方法でも、たくさんの選択肢から一つを選んでメッセージを送るという基本構造は変わりません。しかし、実際にメッセージを送る前に、ETS カウンタをアトミックに更新[25]しなければなりません。全クライアントで共有できる量は限界があり（スーパバイザを介しても、config や ETS の値を使うにしても、です)、プロセスに対するリクエストはまずはじめに

[24] このライブラリの中で実装されている lhttpc_lb モジュール

[25] `ets:update_counter/3` を使いましょう

クリアする必要があります。

このアプローチは `dispcount`[26]モジュールで使われていて、メッセージキューを避け、任意のメッセージに対して (処理されないかもしれませんが) 低レイテンシを保証することで、リクエストが拒否されたことを知るまでに待たずに済むようになっています。他のワーカをつかって処理を続けるか、可能な限りすぐに諦めるかは、利用する側にまかせています。

3.3.6 どのようにドロップするか

ここで提示された解決策のほとんどはメッセージの数によって動作していましたが、予測できるのであればメッセージのサイズや複雑さに応じた解決策も出来ます。エントリーを数えるかわりにキューやスタックバッファを使う場合は、そのサイズを測るか、最初から制約として決められた容量を与えれば良いでしょう。

私が実務経験の中で発見したことは、どのような種類のメッセージであろうともドロップはかなりうまくいきますが、アプリケーションごとに受け入れられるドロップするメッセージの量の妥協点は異なります。[27]

また、データが—システム全体が非同期パイプラインの一部になっていて—「ファイア・アンド・フォーゲット」の形で送信される場合もあります。そして、この場合、エンドユーザーに対してあるリクエストがドロップされたり取り逃がされている理由をフィードバックを送ることが難しくなります。ドロップしたレスポンスを集めてユーザーに「`N` 個のメッセージが `X` という理由でドロップされました」というような特別なメッセージを送れるようにしておくと、それだけで、ユーザーがずっと納得しやすい妥協点を提供できるでしょう。これは Heroku の logplex というログルーティングシステムで取った選択で、このシステムでは、L10 エラーを出して、ユーザーにシステムの一部が対応出来てない旨を警告します。

結局、負荷に対応するために何が受け入れられるかは、そのシステムを使う人間に依存しがちです。しばしば新しい技術を開発するよりも要求を曲げるほうが簡単ですが、新しい技術を開発することが避けられないこともあります。

3.4 演習

復習問題

1. Erlang システム内での過負荷のよくある原因を挙げてください
2. 過負荷の対応策の主な戦略を 2 つ答えてください
3. 長時間稼働する処理はどのようにして安全にできますか
4. 同期処理を行うとき、タイムアウトはどのように設定しますか

---

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

[27] Butler W. Lampson による Hints for Computer System Designs のような古い論文ではメッセージのドロップを推奨しています。たとえば「システムが過負荷になってしまう状況を許可するのではなく、管理できる程度に負荷を落としましょう。」というコメントがあります。それ以外にも「システムはあらゆるリソースへの要求がキャパシティの三分の二を超えると、負荷を極めてうまく対応しないと、期待した機能を発揮できないでしょう。」というコメントもあります。それに付け加えて「うまく対応できるようなシステムは、よく知られた負荷がかかっているシステムだけです。」というコメントもあります。

5. タイムアウト以外の代替案はなんでしょうか
6. どんなときにスタックバッファの前にキューバッファを選びますか

自由回答問題

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

第II部

アプリケーションを診断する

第4章 リモートノードへの接続

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

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

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

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

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

4.1 ジョブ制御モード

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

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

[2] 詳細は http://learnyousomeerlang.com/distribunomicon#cookies か http://www.erlang.org/doc/reference_manual/distributed.html#id83619 参照

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

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

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

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

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

4.2 Remsh

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

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

ショートネームを使う場合は、以下のようになります。

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

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

4.3　SSH デーモン

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

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

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

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

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

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

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

もし接続する際に問題が発生したら、ssh にオプションとして-oLogLevel=DEBUG をつければ、デバッ

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

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

グ出力が見られます。

4.4 名前付きパイプ

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

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

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

ノードに接続するには、`to_erl` プログラムを使います。

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

1>
```

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

4.5 演習

復習問題

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

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

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

第5章 ランタイムメトリクス

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

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

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

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

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

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

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

5.1 グローバルビュー

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

[1] 自動化プロセスが何かしら是正措置を行おうとして、暴走したり、やり過ぎたりしないか保証するほうが、より一層複雑です
[2] http://ferd.github.io/recon/

知するのは不可能です。

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

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

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

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

5.1.1 メモリ

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

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

[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] よくあるパターンでは、`ectr` アプリケーションが適しています。https://github.com/heroku/ectr を参照してください。
[10] あなたがピンチの時にも諸々の出力ができるよう、`recon` アプリケーションは関数 `recon:node_stats_print/2` を備えています

```
{ets,305920}]
```

これは説明が必要ですね。

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

`total` の項には、`processes` と `system` で利用してるメモリ量の和が入ります (instrument モジュールを有効にして[11]VM を実行しないかぎり、完璧な和にはなりません！)。`processes` の項は、スタックやヒープとして Erlang プロセスが使用するメモリ量です。`system` の項は残りです。つまり、ETS テーブルや、VM の中で使われるアトムや、refc バイナリ[12]。先述の通り、instrument モジュールを使わない限り一部データは失われます。

システム限界 (`ulimit`) を突破する時などに仮想マシンが保有する総メモリ量を知りたいなら、VM 内部から取得するのはより困難です。もし `top` や `htop` を使うことなしにこのデータが欲しければ、あなたは VM のメモリアロケータを深堀して見つけ出すしかありません[13]。

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

- `used`。使用中の Erlang データが割り当てられているメモリの総量
- `allocated`。VM により予約されているメモリ量。使用中のメモリ量だけでなく、OS から与えられた未使用な領域も含む。あなたが `ulimit` や OS の示す値を取り扱うなら、この値がほしい値でしょう。
- `unused`。VM により予約されているがまだ割り当てられていない容量。`allocated - used` と等価です。
- `usage`。割当てられたメモリ量に対する使用中のメモリ量の割合 (0.0 から 1.0)。

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

5.1.2　CPU

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

- VM はスケジューリングの際にプロセスとは無関係のタスクをたくさん行っています。高度なスケジューリングタスクと Erlang プロセスによる大量のタスクは、特徴付けすることが難しいのです。
- VM は内部で **リダクション** に基づくモデルを使っています。このモデルは任意の数のタスクで表せ

[11] 訳註：例えば `erl -instr` などで有効化できます。

[12] リファレンスカウントされたバイナリ (reference-counted binary)。プロセスヒープ外に保持するバイナリオブジェクト (実データ及び参照カウンタ) と、プロセスヒープ内に保持する ProcBin オブジェクト (バイナリオブジェクトを参照するオブジェクト) から構成されるバイナリのこと。7.2 節参照

[13] 詳しくは 7.3.2 節を見てください

ます。すべての (BIF を含む) 関数呼び出しでプロセスリダクション数がインクリメントされ、所与のリダクション数に到達すると、プロセスがスケジューラ上の実行中から外れます。

- 負荷の低いスケジューラスレッドがすぐスリープ状態にならないように、Erlang スケジューラの扱うスレッドはしばらくビジーウェイト状態になります。これは、突然負荷が上がるケースでも遅延が大きくなり過ぎないための配慮です。この値を変更するには、VM フラグ (`+sbwt none|very_short|short|medium|long|very_long`) が使えます。

これらの要素が組み合わさっているため、実行中の Erlang コードによる CPU 使用量をしっかりと測る方法を見つけるのは大変です。Erlang ノードは実運用において、ほどほどのタスクでも CPU を大量に使用するようによく見えますが、これは負荷が上がっても残りの CPU リソースで多くのタスクをこなせるよう適合させているのです。

このデータを最も適切に表現するのは、スケジューラの総経過時間です。これは標準では無効のメトリクスで、ノードごとに手動で有効に設定して一定間隔で取得させる必要があります。取得される値からわかるのは、スケジューラがプロセス、普通の Erlang コード、NIF、BIF、ガベージコレクションなどを走らせる時間と、スケジューラがプロセススケジュールを試みているか、アイドル状態になっている時間の割合です。

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

基本的な使い方は Erlang/OTP のリファレンスマニュアル[14]に記載されていますが、recon を使ってもこの値は取得できます。

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

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

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

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

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

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

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

5.1.3　プロセス

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

5.1 の章で説明するツールのほとんどが、何らかの方法でプロセス数を監視していますが、自前でプロセスカウントが必要であれば、以下のようにすれば十分です。

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

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

5.1.4　ポート

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

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

```
1> recon:port_types().
[{"tcp_inet",21480},
 {"efile",2},
 {"udp_inet",2},
```

```
{"0/1",1},
{"2/2",1},
{"inet_gethost 4 ",1}]
```

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

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

その他のタイプのほとんどは、ポートがやり取りしているドライバ名が入ります。例としては **port programs**[15]や **port drivers**[16]などです。

繰り返しになりますが、これらの値を監視することはシステム使用率や負荷を見たり、リークを見たり、その他いろいろなことに役立ちます。

5.2 内部分析

大規模なビュー (または、ロギング) が問題の潜在的な原因を示す時は大抵、目的を持って内部状態を分析する事が面白くなってきます。プロセスの状態はおかしいですか？ 多分トレーシングが必要になるでしょう[17]! 特定の関数の呼び出しや入出力を監視する時はいつでも、トレーシングが有効ですが、多くの場合、監視前に多くの内部分析が必要になります。

Chaptere 7 で議論する特定のテクニックが必要なメモリリークを除けば、最も共通するタスクはプロセスとポート (ファイルディスクリプタとソケット) に関連しています。

5.2.1 プロセス

どんな事があろうとも、プロセスは実行中の Erlang システムの重要な部分です。そして、プロセスは動作中のすべてのものの中心にあるので、プロセスについて知りたい事が沢山あります。幸いにも、VM は多くの情報を提供しており、そのうちのいくつかは安全に使用できますが、一部は安全ではありません (シェルプロセスにコピーされ表示するのに使用するメモリ量が、ノードを強制終了させる程大きいデータセットを返す事ができるからです)。

全ての値は、`process_info(Pid, Key)` か `process_info(Pid, [Keys])` を呼び出して取得する事ができます[18]。よく使われるキーはこちらになります[19]:

Meta

[15] http://www.erlang.org/doc/tutorial/c_port.html
[16] http://www.erlang.org/doc/tutorial/c_portdriver.html
[17] Chapter 9 参照
[18] プロセスに機密情報が含まれている場合、`process_flag(sensitive, true)` を呼び出す事でデータを非公開にする事ができます
[19] **全ての**オプションは http://www.erlang.org/doc/man/erlang.html#process_info-2 を参照してください

`dictionary`　プロセス辞書の全てのエントリを返します[20]。ギガバイトもする任意のデータをそこに格納する事はまず無いので、一般的には安全です。

`group_leader`　プロセスのグループリーダーは IO(ファイル、`io:format/1-3` の出力) をどこに向けるかを定義します。[21]

`registered_name`　もしプロセスが名前 (`erlang:register/2` で登録) を持っているなら、それを取得します。

`status`　スケジューラーによって参照できるプロセスの特性。取りうる値は以下となります。

`exiting`　プロセスは終了していますが、まだクリアされていません

`waiting`　プロセスは `receive ... end` によって待機中です

`running`　実行中

`runnable`　実行の準備は整っていますが、他のプロセスが実行中の為、まだスケジューリングされていません

`garbage_collecting`　GC 中

`suspended`　BIF、またソケットやポートのバッファがいっぱいになる事によるバックプレッシャーのメカニズムによって停止された時はいつでも。ポートがビジーでなくなったら、プロセスは再び runnable になります。

Signals

`links`　他のプロセスやポート (ソケット、ファイルディスクリプタ) に対する全てのリンクのリストを表示します。

`monitored_by`　(`erlang:monitor/2` を使って) 現プロセスをモニタしているプロセスのリストを表示します。

`monitors`　`monitored_by` の反対。モニタされている全てのプロセスのリストを取得します。

`trap_exit`　プロセスが終了 (exits) メッセージを補足する場合は `true`、そうでない場合は `false` となります。

Location

`current_function`　`{Mod, Fun, Arity}`の形式のタプルで、現在実行中の関数を表示します。

`current_location`　`{Mod, Fun, Arity, [{File, FileName}, {line, Num}]}`形式のタプルで、モジュールの現在のロケーションを表示します。

`current_stacktrace`　前述のオプションよりも詳細な形式、'current location' のリストで現在のスタックトレースを表示します。

`initial_call`　プロセス生成時に実行されていた関数を`{Mod, Fun, Arity}`形式で表示します。これは現在実行中のプロセスが何かというより、プロセスがどう生成されたかの特定に役立ちます。

Memory Used

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

[21] 詳細は http://learnyousomeerlang.com/building-otp-applications#the-application-behaviour と http://erlang.org/doc/apps/stdlib/io_protocol.html を参照。

`binary` Refc バイナリ[22]への参照をそのサイズと一緒に表示します。プロセスへの割り当て数が大きい場合、使用するのは危険です。

`garbage_collection` プロセスのガベージコレクションに関する情報が含まれています。その情報は' 予告なしに変更がある' 内容となるでしょう。この情報は、プロセスが実行したガベージコレクションの回数、フルスワップガベージコレクションのオプション、ヒープサイズといったエントリが含みます。

`heap_size` 典型的な Erlang プロセスは、' 古い' ヒープと' 新しい' ヒープを含み、世代別ガベージコレクションが行われます。このエントリは最新世代のプロセスのヒープサイズを示し、通常はスタックサイズが含まれます。値は **words** 単位で返します。

`memory` コールスタック、ヒープ、プロセスの一部である VM によって使用される内部構造、これらを含むプロセスのサイズを **bytes** 単位で返します。

`message_queue_len` プロセスのメールボックスで待機中のメッセージ数を返します。

`messages` プロセスのメールボックス内の全てのメッセージを返します。もしロック状態になっているプロセスをデバッグしているのなら、メールボックスは何百万ものメッセージを保持できるので、この属性を本番環境で取得するのは**非常に**危険です。安全に使用できる事を確認する為、**常に** `message_queue_len` を最初に呼び出してください。

`total_heap_size` `heap_size` と似ていますが、最新世代のヒープだけでなく、それ以外の全ての断片、つまり旧世代のヒープを含みます。値は **words** 単位で返されます。

Work

`reductions` ErlangVM は、スケジューリング実装の移植性を高める任意の作業単位である **reductions** に基づいてスケジューリングを行います (時間ベースのスケジューリングは通常、Erlang が動作する多くの OS で効率的に実行させるのが難しいです)。reductions が大きいほど、CPU と関数呼び出しの面で、より多くの仕事をこなしています。

幸いなことに、全ての安全な共通の属性を取得するのに、recon が `recon:info/1` 関数を提供しています。

```
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,
```

[22] Section 7.2 参照

```
                                  [{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}]}]
```

使用しやすいよう、recon:info/1 はプロセス id、文字列"<0.12.0>"、名前として登録されたアトム、グローバルネーム{global, Atom}、サードパーティーのレジストリで登録された名前 (例えば gproc による{via, gproc, Name})、タプル{0,12,0}といった pid ライクな最初の引数を受け取ってそれを処理します。このプロセスはデバッグ対象のノードにある必要があります。

カテゴリの情報だけ欲しいのであれば、カテゴリを直接指定できます。

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

また、process_info/2 と全く同じ方法で使用できます。

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

後者の方式は、安全でない情報を取得する方法です。

これらの全てのデータを使って、システムをデバッグする為に必要なものを全て見つける事ができます。難しいのは、プロセス単位のデータとグローバルなプロセス間で、どのプロセスをターゲットにするかを把握する事です。

メモリ使用量が高い箇所を探している時、例えばノードの全てのプロセスをリストアップして上位 N 個の使用量が高いプロセスを見つける事ができるのは興味深いです。この情報と recon:proc_count(Attribute, N) 関数を使って、次の結果を得る事ができます。

```
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}}]}]
```

前述の属性のいくつかが作用して、問題を引き起こす可能性のある長時間生存しているプロセスを持つノードだった場合、これはかなり有用な関数です。

しかし、ほとんどのプロセスが短命で、通常は他のツールで調べるには短すぎる場合や、(例えば、**今この瞬間**、どのプロセスがメモリを積んでいるのか、またコードを実行中なのかといった) 必要な情報が検査対象の区間から移動している場合、問題があります。

このようなケースの為、Recon は `recon:proc_window(Attribute, Num, Milliseconds)` 関数を持っています。

この関数をスライドする区間上のスナップショットとして見る事が重要です。サンプリング中のプログラムのタイムラインは以下の様になります。

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

この関数はミリ秒単位で指定する間隔で 2 つのサンプルをとります。

いくつかのプロセスは `w` から `x` の間で、いくつかは `y` から `z` の間で、またいくつかは `x` から `y` の間で生存してから死にます。これらのサンプルは (どちらか、または両方が欠けていて) 不完全なので、重要ではなくなるでしょう。

プロセスの大半が (絶対値で)`x` から `y` の間隔の中で実行されるのであれば、それらのプロセスの寿命は `w` から `z` の時間に相当するので、サンプリング時間をこれより短くする必要があります。でなければ、結果を歪めてしまいます。データを蓄積する時間 (reductions とも言います) の 10 倍の長寿命のプロセスは、大きなボトルネックのように見えます。[23]

この関数が実行されると、次の様な結果が得られます。

```
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}}]}]
```

[23] 注意: この関数は、2 つのスナップショットで収集されたデータに依存し、それらを区別するためのエントリを持つ辞書を構築します。数十万ものプロセスがある時、短い時間にメモリに大きな負担をかける可能性があります。

これらの 2 つの関数によって、問題を引き起こしたり、振る舞いのおかしい特定のプロセスに対処する事ができるようになります。

5.2.2　OTP プロセス

対象のプロセスが OTP プロセスの場合（本番環境のほとんどのプロセスはおそらく OTP プロセスです）、それらのプロセスを調査するための多くのツールをすぐに見つけられます。

通常は sys[24]が知りたいツールでしょう。sys モジュールのドキュメントを読めば、なぜそんなに便利なのかがわかるはずです。OTP プロセスに対する下記のような機能を持っています。

- 全てのメッセージと状態遷移を、シェルやファイルまたは参照可能な内部バッファにすらロギングできます
- 統計情報（リダクション、メッセージ数、時間など）
- プロセスの状態（状態などのメタデータ）
- （`#state{}` レコードに）プロセスの状態を取得
- 状態の書き換え
- コールバックとして使えるようにデバッグ関数をカスタマイズ

また、プロセスの実行を中断、再開する機能も提供します。

これらの機能の詳細は割愛しますが、それらの機能が存在しているということは認識しておいてください。

5.2.3　ポート

プロセスと同様に、Erlang のポートは沢山の内省する機会を与えてくれます。ポートの情報へは `erlang:port_info(Port, Key)` を呼ぶことによってアクセスでき、それ以上の情報は `inet` モジュールを通して取得できます。それらのほとんどは、`recon:port_info/1-2` によって再編成され、それはプロセス関連の同等の関数のインターフェースとやや似ているものを使っています。

メタ情報

`id`　ポートの内部インデックス。ポートを区別すること以外に用途はありません。

`name`　ポートのタイプです。例えば、`"tcp_inet"`、`"udp_inet"`、`"efile"`などの名前で表します。

`os_pid`　もしもポートが inet ソケットではなく、プログラムの外のプロセスを表しているのであれば、このポートの値は先述した外部プログラムに関連する OS の pid を含んでいます。

シグナル

`connected`　各ポートは、その責務として controlling process の機能を持っており、このプロセスの pid は `connected` となります。

`links`　ポートは、他のプロセスと同様にプロセスとリンクできます。この値にリンクされたプロセスのリストが含まれます。このプロセスが獲得されてしまったり、手作業によりプロセスとリ

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

ンクされない限り、この値の使用は安全です。

`monitors` 外部プログラムを表すポートは、それらのプログラムの情報を持ち、最終的に Erlang のプロセスをモニターします。これらのプロセスはここで列挙されています。

IO

`input` ポートに読み込まれたバイト数です。

`output` ポートに書き込まれたバイト数です。

メモリ使用量

`memory` これは、ポートのランタイムシステムによって割り当てられたメモリ（バイト単位）です。この数値は小さくなる傾向があり、そのポート自身によって割り当てられたスペースは除外しています。

`queue_size` ポートのプログラムは、ドライバキューと呼ばれる特別なキューを持っています[25]。これはドライバが遅いデバイスを待ったり、エミュレータに処理を返却するのに便利です。

タイプ特有の情報

inet ポート inet に関するデータを返却します。このデータは統計情報[26]、ソケット (sockname) のローカルアドレスとポート番号、そして 使われている inet オプション[27]を含んでいます。

その他 現在のところ、recon では inet ポート以外の形式はサポートされておらず、空のリストが返却されます。

次のようなリストが取得されます:

```
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,...},
                     {...}|...]},
        {peername,{{50,19,218,110},80}},
        {sockname,{{97,107,140,172},39337}},
        {options,[{active,true},
                  {broadcast,false},
                  {buffer,1460},
```

[25] ドライバキューはエミュレータからドライバ（ドライバからエミュレータへのデータは通常の Erlang のメッセージキューにキューイングされます。）へ出力をキューイングするのに使えます。

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

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

```
                {delay_send,...},
                {...}|...]}]}]
```

この例の一番上に、特定の、プロセスに関与している問題のあるポートを見つけ出す関数があります。これまででわかることは、recon は inet ポートと限定された属性についてサポートするのみということです: 送信、受信、そして両者のオクテット（バイト）数（それぞれ send_oct、recv_oct、oct）か、送信、受信、そして両者のパケット数（それぞれ send_cnt、recv_cnt、cnt）です。

ここまでの累積合計量は、どのポートが遅い一方で確かに帯域を食いつぶしているかを見つけるのに役立ちます:

```
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}]}]
```

これは、いくつかのポートがただ受信しているだけで沢山のバイトを食い荒らしていることを示してします。そして recon:port_info("#Port<0.6821166>") を使って深掘り、誰がそのソケットを所有しているか、何が起こっているかを見つけることができます。

もしくはその他のケースで、recon:inet_window(Attribute, Count, Milliseconds) を使って、あるタイムウィンドウの間で何がほとんどのデータを送信しているのか[28]を見ることができます:

```
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}]}]
```

この例では、中ほどのタプルの値は設定した特定の時間の間（ここでは 5 秒間）send_oct どのような値だったか（もしくは呼び出し時に設定した属性に対応するもの）を表しています。

依然、正しくプロセス（と、もしかしたら特定のユーザや顧客）にリンクされた行儀の悪いポートの巻き添えになった手作業がありますが、これらすべてのツールはちゃんとしています。

5.3　演習

復習問題

1. Erlang のメモリではどのような値が報告されますか。

[28] 前小節の recon:proc_window/3 の説明を見てください

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

自由回答問題

1. グローバルメトリクス内で利用できる長時間のウィンドウが欲しくなる理由はなんでしょうか。
2. 次の問題を見つけるときに使うべき関数は `recon:proc_count/2` と `recon:proc_window/3` のどちらでしょうか。
3. （a）リダクション
   （b）メッセージキューの長さ
   （c）メモリ
4. あるプロセスのスーパーバイザーがどれかに関する情報はどうやって見つけますか。
5. `recon:inet_count/2` や `recon:inet_window/3` はそれぞれいつ使うべきでしょう。
6. OS から報告されたメモリと Erlang の memory 関数から報告されるメモリの違いを説明してください。
7. ときどき Erlang が実際にはさほど稼働していないのに、非常に稼働率が上がっているように見えるのはなぜでしょうか。
8. あるノード上のプロセスの何割が実行可能だがすぐにはスケジュールできない状況になっているかを調べる方法を教えてください。

ハンズオン

次のコードを使って回答してください。https://github.com/ferd/recon_demo:

1. システムメモリとはなんですか。
2. ノードは多くの CPU リソースを使っていますか。
3. メールボックスが溢れいているプロセスはありますか。
4. どの chatty プロセス（`council_member`）が一番メモリを使っていますか。

5. どの chatty プロセスが一番 CPU を使用していますか。
6. どの chatty プロセスが一番帯域を消費していますか。
7. どの chatty プロセスが TCP で一番メッセージを送っていますか。また一番メッセージを送っていないものも答えてください。
8. ノード上のあるプロセスが複数の接続やファイルディスクリプタを同時に保持しがちかどうか、どのように判断しますか。
9. 今現在あるノード上でほとんどのプロセスから同時に呼ばれている関数を見つけられますか。

第6章 クラッシュダンプを読む

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

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

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

6.1 一般的な見方

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

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

Slogan: eheap_alloc: Cannot allocate 2733560184 bytes of memory
(of type "old_heap").
```

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

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

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

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

```
Memory:
===
  processes: 2912 Mb
  processes_used: 2912 Mb
  system: 8167 Mb
  atom: 0 Mb
  atom_used: 0 Mb
  binary: 3243 Mb
  code: 11 Mb
  ets: 4755 Mb
  ---
  total: 11079 Mb

Different message queue lengths (5 largest different):
===
      1 5010932
      2 159
      5 158
     49 157
      4 156

Error logger queue length:
===
0

File descriptors open:
===
  UDP:  0
  TCP:  19951
  Files:  2
  ---
  Total:  19953

Number of processes:
===
36496

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

```
different):
===
       1 284745853
       1 5157867
       1 4298223
       2 196650
      12 121536

Processes OldHeap memory sizes (words) used in the VM (5 largest
different):
===
       3 318187
       9 196650
      14 121536
      64 75113
      15 46422

Process States when crashing (sum):
===
       1 Garbing
      74 Scheduled
   36421 Waiting
```

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

- メモリフラグメンテーション;
- C コードまたはドライバーのメモリリーク;
- クラッシュダンプを生成する前にガベージコレクトされてしまった大量のメモリ[5].

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

この特定のダンプでは、1 つのプロセスのメールボックスに 5 百万のメッセージが格納されていたと示されています。これは取得できるすべてのメッセージがパターンマッチしないか、過負荷になっています。そ

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

こには数百のメッセージをキューに溜めた数十のプロセスも存在しています。— これは過負荷または競合を指すことがあります。あなたの一般的なクラッシュダンプについて一般的なアドバイスをすることは難しいですが、これらを理解するのに役立つ方法はまだいくつかあります。

6.2　メールボックスがいっぱい

いっぱいのメールボックスについては、大きなカウンターを見るのが最良の方法です。もし大きなメールボックスが 1 つある場合は、クラッシュダンプのそのプロセスについて調べてください。メッセージがパターンマッチしないか、過負荷のために発生しているかどうかを把握してください。もし同様のノードが動いているのなら、そこに接続して調査することができます。いっぱい溜まっているメールボックスがたくさんあると分かっている場合は、クラッシュ時に何の関数が動いていたか把握するため recon の `queue_fun.awk` を使うことができます。

```
$ 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
...
```

これはクラッシュダンプに対して実行され、メールボックスに少なくとも 10000 メッセージあるプロセスに対して実行がスケジュールされていた関数をすべて出力します。この実行の場合は、すべてのノードが `io:format/2` の呼び出しのために IO 待ちでロックしていたことをスクリプトが示していました。

6.3　非常に多い（もしくは非常に少ない）プロセス

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

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

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

[6] 詳細は 5.1.3 を参照のこと

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

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

6.4 大量のポート数

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

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

6.5 メモリ割り当てができない

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

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

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

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

6.6 演習

復習問題

1. クラッシュダンプが生成される場所をどのように指定しますか?
2. クラッシュダンプがノードがメモリ不足で落ちていることを示している場合、通常どのように探しますか?
3. プロセス数が疑わしいほど少ない場合、どこを見るべきでしょうか?
4. ノードが大量のメモリを確保したプロセスとともに死んだことがわかる場合、それがどれであるか

[7] 詳細は 5.1.4 を参照のこと

見つけるために何をすることができますか?

ハンズオン

6.1 の章にあるクラッシュダンプの分析を使用します。

1. 問題を指し示すことができる特定の異常値は何でしょうか?
2. 繰り返されるエラーは問題のように見えますか?そうではない場合、それは何になるのでしょうか?

第7章 メモリリーク

Erlang ノードがメモリを垂れ流す原因は山ほどあります。それらは非常に簡単に見つけられるものから驚くほど見つけにくいものまであり（幸いにして、後者は稀です)、場合によってはそれらのメモリリークが何の問題も引き起こさないこともあります。

メモリリークについては以下の 2 通りの方法で知ることができます:

1. クラッシュダンプ（6 を参照）
2. モニタリングしているデータにやっかいな傾向を見つける

この章では主に後者のようなリークに焦点を当てます。その理由は、こちらのほうが調査しやすく、増加がリアルタイムで見えるためです。ここでは、ノード上で何の数値が伸びているか、一般的な対処の選択肢、（特別なケースである）バイナリリーク、メモリの断片化の発見方法を扱います。

7.1 よくあるリークの原因

誰かが「助けて！ ノードがクラッシュしているの！」と助けを求めてきたとき、最初にするべきことは常にデータを求めることです。ここで重要な質問やデータには以下のようなものがあります:

- クラッシュダンプはありますか、そしてそれは特にメモリについて文句を言っていますか？ もし違うのならば、この問題はメモリとは関係がない可能性があります。もしそうであるならば、クラッシュダンプを掘り返しましょう。データが山のようにあります。
- クラッシュは周期的に起こりますか？ どのくらい予測可能ですか？ クラッシュと同じくらいの時間にはほかに何が起こる傾向があり、関連していそうですか？
- クラッシュは負荷のピークと同じくして起こりますか、それともほぼ無関係にどんなときでも起こりますか？ 特にピーク**中に**起こるクラッシュはしばしば負荷のマネジメントの失敗から来ます（3 を参照)。どんな状況でも起こるクラッシュ、特にピーク後に負荷が減っても起こるようなクラッシュは本当にメモリに起因する問題である可能性が高いです。

これらの質問ですべてメモリリークを指し示すような回答を得たならば、5 章にて説明されているメトリ

クスライブラリまたは recon をインストールし、データを探索する準備をしましょう。[1]

どのようなケースでも、最初に見るべきはデータの傾向です。erlang:memory() またはライブラリやメモリシステムにある類似のものを使ってすべてのメモリの種類について調べましょう。以下の点を調べるとよいです:

- ある種類のメモリ使用量が他のタイプのメモリ使用量よりも速く伸びていますか？
- ある種類のメモリが利用可能なメモリ空間の大部分を占めていませんか？
- ある種類（アトムを除く）のメモリ使用量が全く減りそうになく、常に増えていませんか？

増えているメモリ使用の種類に応じて使える選択肢は様々です。

7.1.1　アトム

動的にアトムを生成してはいけません！ アトムはグローバルなテーブルに保存され永久にキャッシュされるためです。erlang:binary_to_term/1 や erlang:list_to_atom/1 を呼んでいる場所を探し、erlang:binary_to_term(Bin, [safe]) や erlang:list_to_existing_atom/1 のような同様の機能でより安全なものを用いるようにしましょう[2]。

もし Erlang に同梱されている xmerl ライブラリを使っているのならば、オープンソースの代替品を使うか[3]、自分で安全な SAX パーサーを追加するか[4] を検討してみてください。

これらのことをしていない場合、ノードとどのようにインタラクションを行っているかを調査してみてください。実運用で問題を起こした一つの例として、我々の使っていたいくつかの共通ツールが外部のノードに接続するときにランダムな名前を使っていたり、中央サーバーを介してお互いにつながっている[5]ランダムな名前のノードを生成していました。Erlang ノード名はアトムに変換されるため、このような仕組みがあるだけでゆっくりとしかし確実にアトムテーブルの空間を埋め尽くしていくのでした。固定された範囲の中で生成するか、長期的に問題にならないくらい遅いペースで生成するように気をつけましょう。

7.1.2　バイナリ

7.2 章を参照。

7.1.3　コード

Erlang ノードのコードはメモリ上の独自の領域に読み込まれ、ガベージコレクションが行われるまでその領域を使い続けます。同じモジュールの異なるバージョンは同時に 2 つまでしか存在できないので、巨大なモジュールを探すことによって問題のある箇所を見つけるのは簡単なのではないでしょうか。

---

[1] 動作しているノードへの接続に手助けが必要な場合は 4 章を参照。

[2] 訳注: erlang:binary_to_atom/2 も同様に注意が必要です。これは binary_to_existing_atom/2 で代用できます。

[3] exml や erlsom などは悪くないでしょう

[4] Ulf Wiger の以下の記事を参照: http://erlang.org/pipermail/erlang-questions/2013-July/074901.html

[5] これはノード同士をどのようにつなげるかという問題を解決するためによく使われるアプローチです。一つか二つの名前の固定された中央ノードを用意し、他のノードはそれらに接続するようにします。そうすると、自ずと接続は伝搬します。

もしどのモジュールも特に目立って大きくないならば、HiPE[6]でコンパイルされたコードを探してみましょう。通常の BEAM コードとは違い、HiPE コードはネイティヴコードで、新しいバージョンが読み込まれたときでも VM からガベージコレクションされないという性質があります。たくさんの、または巨大なモジュールが実行時にネイティヴコンパイルされて読み込まれた場合、メモリ使用量は徐々に蓄積されていきます。

または、自分でロードしたものでない怪しげなモジュールを探して、誰かがシステムへ侵入したとパニックすることもあるでしょう！

7.1.4 ETS

ETS テーブルはガベージコレクションの対象にならず、レコードがテーブル上から削除されない限りメモリを使い続けます。レコードを手動で削除するかテーブルを削除することでしか空きメモリを取り戻すことができません。

本当に ETS データでメモリリークをしているという稀なケースでは、ドキュメント化されていない `ets:i()` 関数をシェルで呼んでみてください。これはエントリの数 (`size`) とどのくらいメモリを使っているか (`mem`) についての情報を出力します。これを使って何がおかしいかを探ってみましょう。

ETS 上のすべてのデータが正当なもので、データセットを複数のノードにシャーディングして分配するという難しい問題に直面している、という可能性も全くもってありえます。これはこの本の範囲外なので、あなたの幸運を祈ります。テーブルの圧縮をすることで時間を稼ぐという選択肢もあります[7]。

7.1.5 プロセス

プロセスメモリが増えるには多くの原因があります。最も興味深いケースはいくつかの頻出ケースに関連します。例えば、プロセス自身をリークしている、特定のプロセスがメモリをリークしている、といったケースです。複数の原因に由来している可能性はあるので、複数のメトリクスを調べる価値は高いでしょう。なお、プロセスのカウントについては既出なのでここでは扱いません。

リンクとモニタ

グローバルでのプロセス数がプロセスのリークの兆候を示していますか？ もしそうであれば、リンクされていないプロセスを調査するか、スーパーバイザの子プロセスリストを見て何かおかしいところがないかを探りましょう。

リンクされていない、またはモニタされていないプロセスを探すのは簡単ないくつかのコマンドでできます:

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

[6] http://www.erlang.org/doc/man/HiPE_app.html
[7] `ets:new/2` の `compressed` オプション を見てみてください

これはリンクもモニタもされていないプロセスのリストを返します。スーパーバイザについては、`supervisor:count_children(SupervisorPidOrName)` を見て正常な状態がどんなものかを見てみるのがよいでしょう。

メモリ使用

プロセスごとのメモリモデルについては 7.3.2 節で簡潔に紹介していますが、一般的には、どの個別のプロセスが一番メモリを使っているかはプロセスの `memory` 属性を見ることでわかります。この値は、メモリ使用量の絶対値として、または区間を指定してその中での変化率として見ることができます。

メモリリークについていえば、急激な上昇が予測可能でない限りにおいて、絶対値から順に見ていくのがよいでしょう:

```
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}}]}]
```

`message_queue_len` のような `memory` 以外の示唆的な属性は 5.2.1 節にて紹介していますが、だいたいの場合 `memory` でカバーできるでしょう。

ガベージコレクション

プロセスがものすごく短期間だけ大量のメモリを使うということは大いにありえます。操作のオーバーヘッドの大きい長命のノードではこれらはだいたいの場合問題ではありませんが、メモリが少なくなってきたときにはこのようなスパイク状の挙動こそが本当に対処したいものかもしれません。

リアルタイムですべてのガベージコレクションを監視するのはコストが高いです。代わりに、Erlang のシステムモニタ[8]を使うのが一番よい方法かもしれません。

Erlang のシステムモニタを使えば例えば長いガベージコレクションの期間や巨大なプロセスヒープなどの情報を追跡できます。モニタは以下のようにして設定できます:

```
1> erlang:system_monitor().
undefined
```

[8] http://www.erlang.org/doc/man/erlang.html#system_monitor-2

```
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
```

最初のコマンドは既にシステムモニタを使っていないかどうかをチェックします—使用中の他のアプリケーションや同僚からシステムモニタを奪いたくはないでしょう。

二つ目のコマンドはガベージコレクションに 500 ミリ秒以上かかったときに通知されます。その結果は三つ目のコマンドでフラッシュされます。ヒープサイズを監視したい場合は `{large_heap, NumWords}` を使うとよいでしょう。自信がない場合は大きな値でモニタリングを開始しましょう。例えば、1 ワード以上のサイズを持つヒープの情報でプロセスのメールボックスを溢れさせたくはないですよね？

五つ目のコマンドはシステムモニタを解除し（モニタプロセスが終了または kill されるとシステムモニタは同様に解除されます）、その次のコマンドでちゃんと解除されたかどうかを確認できます。

これを使って、監視メッセージがリークやメモリの過剰使用に由来していそうなメモリ使用量上昇と重なっていないかを見ることができ、事態が悪化する前に犯人をつきとめることができます。すばやく対応してプロセスの中身を調べる（`recon:info/1` などを使います）ことでアプリケーションの問題を発見できるでしょう。

7.1.6 特に何もない場合

これまでに書かれた内容があてはまらない場合、バイナリリーク (7.2 章) やメモリのフラグメンテーション (7.3 章) が原因である可能性があります。もしそのいずれでもなければ、C ドライバ、NIF、または VM 自身がメモリリークを起こしている可能性があります。もちろん、ノードの負荷とメモリ使用が比例していて特に何もメモリリークを起こしていないという可能性もあります。単にシステムはもっとリソースやノードを必要としているだけなのかもしれません。

7.2　バイナリ

Erlang のバイナリは 2 つの主な型から成ります：ProcBin と refc バイナリです[9]。64 バイトまでのバイナリはプロセスのヒープ上に直接割り当てられ、それらの全ライフサイクルはそのヒープ上で費やされます。それより大きいバイナリは、バイナリのためだけのグローバルヒープ上に割り当てられ、バイナリを使うプロセスは、ローカルヒープ内にローカル参照を保持します。これらのバイナリは参照をカウントされており、割り当ての解除は、ある特定のバイナリを参照しているすべてのプロセスのすべての参照がガベージコレクトされたときに 1 回だけ発生します。

99% のケースでは、このメカニズムは正常に動作します。しかしながら、一部のケースでは、このプロセスはうまくいきません:

1. メモリの割り当てとガベージコレクションを保証することについてほとんど機能しません。
2. 最終的には様々なデータ構造の大きなスタックやヒープを広げ、それらを集め、そして沢山の refc バイナリと動きます。バイナリでヒープを満たすことは（たとえ仮想ヒープが、その refc バイナリの実際のサイズに対するカウントに使われていても）、長い時間がかかってしまうかもしれません。それは、ガベージコレクションの間に長い遅延を発生させます。

7.2.1　リークを検出する

参照カウントされたバイナリによるリークを検出することは十分に簡単です。すべての各プロセスのバイナリ参照のリストを基準とし（`binary` 属性を使って）、ガベージコレクションを強制的に実行し、別のスナップショットを取得し、その差を計算します。

この作業は `recon:bin_leak(Max)` を使い、その前後でノードの全体メモリ使用量を見ることで端的に実行することができます。

```
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,
```

[9] http://www.erlang.org/doc/efficiency_guide/binaryhandling.html#id65798

```
  [{current_function,{gen_fsm,loop,7}},
   {initial_call,{proc_lib,init_p,5}}]}]
```

この例では、どれくらいの独立したバイナリが保持された後に開放されているかを差分として表しています。この -5580 は、この関数が実行される前と後で 5580 少ない refc バイナリが存在したことを表しています。

これが常時一定量あることを示しているのは普通のことですし、すべての数字が何かが悪いことを示しているわけではありません。もしあなたが、VM が使用しているメモリがこの関数の呼び出しよりも後に大幅に減っていることを確認したら、それはアイドル状態の refc バイナリがあるということかもしれません。

同様に、もしそうではなく、あなたがいくつかのプロセスがびっくりするほど大量の refc バイナリを保持していることを見つけるのであれば[10]、それは問題があるというよい証拠となるかもしれません。

さらに、あなたは recon でサポートされている特別な binary_memory 属性を使うことによって、バイナリメモリの合計中最もメモリを消費しているものを検証することができます:

```
1> recon:proc_count(binary_memory, 3).
[{<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}}]}]
```

この関数は N の、refc バイナリの参照が保持しているメモリ量によって並び替えられた上位のプロセスを返し、バイナリの実際の量を返す代わりに、いくつかの大きなバイナリを保持する特定のプロセスを示してくれます。

7.2.2 リークを修正する

recon:bin_leak(Max) を使ってバイナリメモリリークを見つけることができたなら、そのトップのプロセスを見てそれがなにか、そしてどんな種類の仕事をしているものかを見つけることは十分にシンプルでしょう。

一般的に、refc バイナリのメモリリークはいくつかの手段によって解決でき、それは原因によって異なります:

- ガベージコレクションを一定の間隔で手動で実行します（これはうずうずしますが、ある程度効果

[10] 筆者は Heroku でのメモリリークの調査中に、いくつかのプロセスが 10 万件ほどの refc バイナリを保持しているのを見つけました！

的です）
- バイナリを使うことをやめる（望ましいということはめったにありません）
- もし大きなバイナリのうち小さな一部分（たいていの場合 64 バイト未満）のみを保持しているのであれば[11]、`binary:copy/1-2`[12]を使います。
- 大きなバイナリを含むものを一時的な一度限りのプロセスに移します。このプロセスは実行を終えると死にます (手動の GC の簡単な方法です！)。
- もしくは適切なタイミングでハイバネーション呼び出しを追加します（アクティブではないプロセスに対する最もきれいな解決法かもしれません)。

最初の 2 つの操作は正直に言って同意できるものではなく、その他の方法が失敗する前に試すべきではありません。後の 3 つの選択肢は通常、実施するのに最もよいものでしょう。

ルーティングバイナリ

何人かの Erlang ユーザが報告した、ある特定のユースケースに対する解決方法があります。問題のあるユースケースは、たいてい、あるプロセスから他のプロセスへバイナリをルーティングする中間プロセスを持っているというものです。したがって、この中間プロセスはそれを通るすべてのバイナリに対する参照を獲得し、refc バイナリリークのよくある主要な原因になる危険があります。

このパターンに対する解決方法は、ルータプロセスにルーティング先の pid を返却させ、呼び出し元にバイナリを移動させることです。バイナリに触る必要があるプロセスのみがバイナリに触ることになるため、これは効果があるでしょう。

この問題に対する修正はルータの API 関数において、呼び出し側の目に見える変更が必要とされることなしにわかりやすく実装することができるでしょう。

7.3　メモリフラグメンテーション

メモリフラグメンテーションの問題は 7.3.2 の章で記述されているように Erlang のメモリモデルと密接に関連しています。これは長時間動いている Erlang ノードの最も難しい問題 (個々のノードの稼働時間が数ヶ月にも達する時によく起こる) の一つで、比較的まれに見られます。

一般的なメモリフラグメンテーションの症状は、ピーク時に大量のメモリが割り当てられ、その後に解放されないというものです。ノードが内部的にレポートするメモリ使用量 (`erlang:memory()` を通した) が OS がレポートするものに比べてとても少ない場合は、明らかにこれが要因です。

7.3.1　フラグメンテーションを見つける

`recon_alloc` モジュールはこのような問題を見つけたり、解決の助けとなるように開発されました。

この種の問題がコミュニティにとって稀であった (または開発者がそれが何であるか知らずに起こった)

[11] refc バイナリよりも大きな領域をコピーするのに値するかもしれません。例えば、2 ギガバイトのバイナリがその小さな一部分を保持し続けている間にガベージコレクトされるのであれば、2 ギガバイトのバイナリのうち 10 メガバイトをコピーすることは小さなオーバーヘッドに値します。

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

と考えると、手がかりを見つけるための様々な手順が定義されているだけです。これらはすべて曖昧で運用者の判断が必要です。

割り当て済みのメモリを確認する

`recon_alloc:memory/1` を呼び出すことで様々なメモリのメトリクスを `erlang:memory/0` より柔軟にレポートします。これらは関連する引数です。

1. `recon_alloc:memory(usage)` を呼び出す。これは Erlang VM が OS から取得したメモリに対して、Erlang 項としてアクティブに使われているメモリの割合を 0 から 1 の値として返します。使用率が 100% に近い場合は、おそらくメモリフラグメンテーションの問題ではありません。たくさんメモリを使っているだけです。
2. `recon_alloc:memory(allocated)` と OS がレポートするものがマッチするか確認してください。[13] 問題が本当にフラグメンテーションか Erlang 項のメモリリークの場合には、それはかなり近く一致する必要があります。

これでメモリがフラグメントしているかどうかを確認できるはずです。

問題のアロケータを見つける

どの種別のアロケータ (7.3.2 を参照) がほとんどのメモリを確保しているか見るために `recon_alloc:memory(allocated_types)` を呼び出してください。結果を `erlang:memory()` と比較して、明らかに犯人のように見えるものがあるか確認してください。

`recon_alloc:fragmentation(current)` を試してください。データのダンプ結果はノード上のそれぞれ異なったアロケータを様々な使用比率とともに表示します。[14]

とても少ない比率の場合、それらが `recon_alloc:fragmentation(max)` を呼び出した結果と異なるか確認してください。これは最大のメモリ負荷の状況下での使用量のパターンを表示するはずです。

大きく異なる場合は、使用率にスパイクのある特定の種別のアロケータについてメモリフラグメンテーションの問題がある可能性があります。

7.3.2 Erlang のメモリモデル

グローバルレベル

メモリがどこへ行くのか理解するために、たくさんのメモリアロケータが使われていることを最初に理解しなければなりません。VM 全体のための Erlang のメモリモデルは階層的になっています。7.1 の図にあるように、2 つのメインとなるアロケータがあり、他にサブアロケータ (1 – 9 までの番号が付けられた) があります。サブアロケータは Erlang コードと VM から使われるほとんどのデータ型用の特定のアロケータです。[15]

---

[13] `recon_alloc` でレポートされた値をバイト (bytes)、キロバイト (kilobytes)、メガバイト (megabytes)、ギガバイト (gigabytes) 表記にするために `recon_alloc:set_unit(Type)` を呼び出すことができます。

[14] http://ferd.github.io/recon/recon_alloc.html に詳細な情報があります。

[15] 各データ型の完全なリストはこちらで見つけることができます。erts/emulator/beam/erl_alloc.types

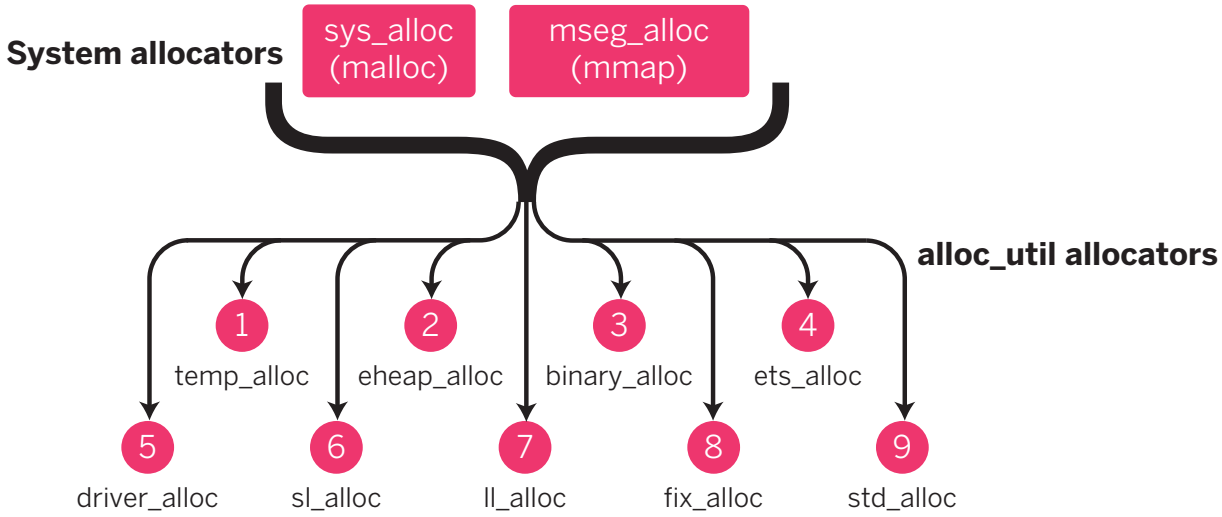


図 7.1　Erlang のメモリアロケータとそれらの階層。R16B03 からオプションで Erlang VM のために利用可能なすべてのメモリを事前に確保すること (加えて制限も) が許されている特別な**スーパーキャリア**は図示していません。

1. `temp_alloc`: 短期間のユースケース用 (1 つの C の関数呼び出しで使われるデータのような) の一時的なアロケーション。
2. `eheap_alloc`: Erlang プロセスのヒープなどに使われるヒープデータ。
3. `binary_alloc`: リファレンスカウントされたバイナリ (Erlang プロセスで共有される「グローバルなヒープ」そのもの) に使われるアロケータ。ETS テーブルに格納されたリファレンスカウントされたバイナリはこのアロケータに残ります。
4. `ets_alloc`: ETS テーブルはガベージコレクトされないメモリの独立した部分へデータを格納しますが、項がテーブルに格納されている限り、割り当ておよび割り当て解除されます。
5. `driver_alloc`: とりわけドライバーデータを格納するために使われ、他のアロケータを使って Erlang 項を生成したドライバーは保持しません。ここで割り当てられたドライバーデータはロック/ミューテックス、オプション、Erlang ポートなどを含みます。
6. `sl_alloc`: 短期間生存するメモリブロックがここに保存されます。VM のスケジューリング情報の一部や、いくつかのデータ型の処理に使う小さなバッファを含みます。
7. `ll_alloc`: 長期間生存するアロケーションはここにあります。例えば Erlang コード自身やアトムのテーブルが存在します。
8. `fix_alloc`: 頻繁に使われる固定サイズのメモリブロックのために使われるアロケータ。ここで使われるデータの 1 つとしては VM で内部的に使われているプロセスの C 構造体があります。
9. `std_alloc`: 上記カテゴリに適合しないものを全てキャッチするアロケータ。名前付きのプロセス用のプロセスレジストリがここにあります。

デフォルトでは、スケジューラごとに各アロケータのインスタンスが 1 つずつ (そしてコアごとに 1 つのスケジューラを持つはずです)、加えて非同期のスレッドを使う linked-in drivers で使われるインスタンス

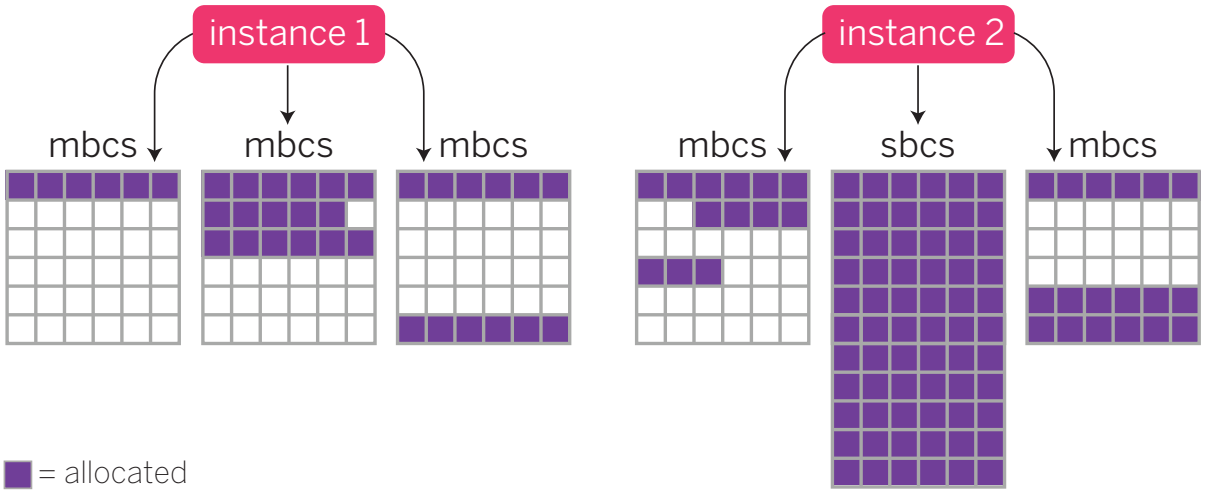


図 7.2 特定のサブアロケータに割り当てられたメモリの例

が一つあります。これは 7.1 の図にあるような構造をあなたに与えますが、それぞれの葉で N 個の部分に分かれます。

これらのサブアロケータはユースケースに応じて 2 つの可能な方法で mseg_alloc と sys_alloc からメモリを要求します。最初の方法はマルチブロックキャリア (mbcs) として行動します。それは多数の Erlang 項のために使われるメモリの塊を一度に取得します。それぞれの mbc のために、VM は与えられたメモリ量 (我々の場合はデフォルトでは 8MB ですが、VM オプションを調整して設定できます) を確保し、それぞれの割り当てられた項は格納されるための十分な空間を探すために、自由にたくさんのマルチブロックキャリアの中を見に行けます。

割り当てられるアイテムがシングルブロックキャリア (sbct)[16]の閾値より大きい場合はいつでも、アロケータはこのアロケーションをシングルブロックキャリア (sbcs) に切り替えます。シングルブロックキャリアは最初の mmsbc[17]のエントリに対して mseg_alloc から直接メモリをリクエストし、その時に sys_alloc へ切り替え、割り当てが解除されるまで項をそこへ格納します。

バイナリアロケータの例も見てみると、7.2 の図のようなものになるでしょう。

マルチブロックキャリア (または最初の mmsbc[18]シングルブロックキャリア) が再利用できる時はいつでも、mseg_alloc は VM に届く次のアロケーションスパイクが、毎回より多くのメモリのためシステムに問い合わせる必要はなく、事前に割り当てたメモリを使うことができるように、しばらくの間それをメモリ内に留めようとします。

次に Erlang VM の色々なメモリアロケーション戦略を知る必要があります。

1. Best fit (bf)
2. Address order best fit (aobf)

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

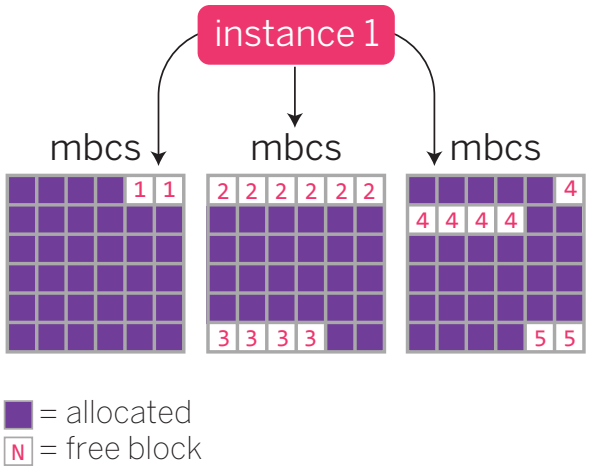


図 7.3　特定のサブアロケータに割り当てられたメモリの例

3. Address order first fit (`aoff`)
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`)

これらの戦略はそれぞれの `alloc_util` アロケータ[19]ごとに個別に設定できます。

best fit(`bf`) 用に、VM は全てのフリーブロックのサイズで平衡バイナリ木を構築し、データが格納できる最も小さいものを探そうとし、それを割り当てます。7.3 の図では、3 つのブロックを必要とするデータがあるとするとエリア 3 になるでしょう。

Address order best fit(`aobf`) は似たように働きますが、代わりに木はブロックのアドレスに基づきます。VM はデータが格納できる利用可能な最も小さなブロックを探しますが、もし同じサイズのものがたくさんある場合は、ブロックのアドレスがより低いものを選びます。3 つのブロックを必要とするデータがある場合、またエリア 3 となりますが、もし 2 つのブロックが必要な場合は、この戦略では 7.3 の図にある最初の `mbcs` のエリア 1(エリア 5 ではなく) になります。これにより VM は多くの割り当てに対し、同じキャリアを優先する傾向を持つようになります。

Address order first fit(`aoff`) は検索時にアドレス順を優先し、ブロックに収まるならすぐに `aoff` がそれを使います。7.3 の図で 4 つのブロックを割りあてるために `aobf` と bf の両方共がエリア 3 を選びますが、これはエリア 2 をアドレスが最も低いために最優先して取得します。7.4 の図で、もし 4 つのブロックを割り当てるなら、この戦略ではアドレスが低いためにブロック 3 ではなくブロック 1 を選び、一方 `bf` は 3 か 4 を選び、`aobf` は 3 を選ぶでしょう。

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

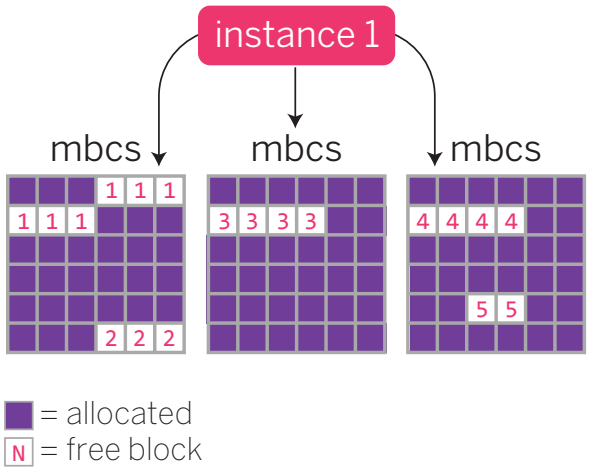


図 7.4 特定のサブアロケータに割り当てられたメモリの例

Address order first fit carrier best fit(aoffcbf) はサイズを格納できる最初のキャリアを選び、その中で最も適合するものを探すという戦略です。7.4 の図で 2 つブロックを割り当てたい場合、bf と aobf は共にブロック 5 となるが、aoff はブロック 1 を選ぶでしょう。aoffcbf は最初の mbcs に格納できて、エリア 2 よりエリア 1 の方がより良く収まるのでエリア 2 を選びます。

Address order first fit carrier address order best fit(aoffcaobf) は aoffcbf と似ていますが、もしキャリアにある複数のエリアが同じサイズである場合に、それを不特定にしたままではなく、その 2 つの中でアドレスが最も小さいものを選びます。

Good fit(gf) は異なる種類のアロケータで、best fit(bf) と同様の働きを試みますが、限られた時間だけ探します。もし完全に収まるところを探せなかったら、これまでに探せたものの中で最も良いものを選びます。この値 (時間) は VM の mbsd[20]引数を通して設定可能です。

最後に **A fit**(af) は一時的なデータのためのアロケータ挙動で、一つの存在しているメモリブロックを探し、データが収まるなら af はそれを使います。もしデータが収まらなければ、af は新しいものを割り当てます。

これらの戦略はそれぞれ個別に全ての種類のアロケータに適用することができ、ヒープアロケータとバイナリアロケータで同じ戦略を共有する必要はありません。

Erlang のバージョン 17.0 からついにそれぞれのスケジューラの alloc_util アロケータごとに **mbcs プール**と呼ばれるものを持つようになりました。mbcs プールは VM のメモリフラグメンテーションと戦うために使われる機能です。アロケータの持つマルチブロックキャリアの 1 つがほとんど空になると[21]、そのキャリアは**放棄された**状態になります。

この放棄されたキャリアは新しいマルチブロックキャリアが必要とされるまで、新しい割り当てのために使われなくなります。もしそれが起こると (新しいマルチブロックキャリアが必要になる)、キャリアは mbcs プールから取得されます。これはスケジューラを超えて同じ種類の複数の alloc_util アロケータに

[20] http://www.erlang.org/doc/man/erts_alloc.html#M_mbsd
[21] http://www.erlang.org/doc/man/erts_alloc.html#M_acul を通して閾値を設定できます

渡って行われます。これは VM にメモリの割り当て解除を強制することなしに、ほとんど空のキャリアをキャッシュすることを許可します[22]。それはまたキャリアが少ないデータを保持していて、それらが必要とされる場合に、スケジューラをまたがったキャリアのマイグレーションを有効にできます。

プロセスレベル

より小さなスケールでは、それぞれの Erlang プロセスごとにレイアウトはまだ少し異なっています。それは基本的に 1 つの箱として想像できるメモリを持っています。

```
1 [                    ]
```

片側の端にはヒープを、その反対側にはスタックを持ちます。

```
1 [ヒープ |      | スタック]
```

実際にはそこにはもっとたくさんのデータがあります (世代別 GC のための古いヒープ、新しいヒープに、プロセスから使われていない特定のサブアロケータにあるリファレンスカウントされたバイナリの領域を考慮した仮想のバイナリヒープも持ちます — `binary_alloc` 対 `eheap_alloc`)。

```
1 [ヒープ   ||      スタック]
```

その領域はスタックかヒープが何も格納できなくなるまでもっと割り当てられます。これはマイナー GC を引き起こします。マイナー GC はオールド領域のヒープに保持できるデータを動かします。そして残りを回収し、さらなる領域を再割り当てするかもしれません。

与えられた回数のマイナー GC や再割り当ての後で、新しいヒープと古いヒープを検査し、より多くの領域を解放するなどフルスイープ GC が行われます。プロセスが死んだ時、スタックとヒープの両方は一度に解放されます。リファレンスカウントされたバイナリはカウントが減り、カウンターが 0 になるとそれらは消滅します。

それが起こると時間の 80% 以上はメモリはサブアロケータで利用可能とマークされ、新しいプロセスまたはリサイズが必要な他のプロセスによって取り戻せるようになります。このメモリが使われていない — マルチブロックキャリアもまた使われていない — 状態になった後でのみ、それは `mseg_alloc` または `sys_alloc` へ返され、しばらくの間それが保持されたりされなかったりします。

7.3.3　異なるアロケーション戦略でメモリフラグメンテーションを修正する

メモリアロケーションに関する VM のオプションを調整することが助けになるかもしれません。

[22] これはあまりにもたくさんのメモリを消費する場合がありますが、`+MBacul 0` オプションによってこの機能を無効にできます。

あなたはメモリ負荷とメモリ使用量の種類が何かについてよく理解し、大量の徹底的なテストを覚悟する必要があります。recon_alloc モジュールはガイダンスを提供するためのいくつかの役立つ機能を含んでおり、この時点でモジュールのドキュメント[23]を読むべきです。

あなたは平均的なデータサイズがどれくらいかや、アロケーションとアロケーション解除の頻度、データが mbcs か sbcs に収まるかどうかについて把握する必要があります。その上で recon_alloc のたくさんのオプションを試し、別の戦略を試し、それらをデプロイし、改善したかマイナスの影響があったかを見る必要があります。

これは近道のないとても長いプロセスで、しかも問題はノードごとに数ヶ月ごとにしか起こりませんので、あなたは長期間それに携わることになるでしょう。

7.4 演習

復習問題

1. Erlang のプログラムにおいてよくあるリークの原因をいくつか挙げてください。
2. Erlang での主要な 2 種類のバイナリはなんでしょうか。
3. どの特定のデータ型もリークの原因でないように思われる場合、何が原因になりうるでしょうか。
4. 多くのメモリを保持したプロセスとともにノードが死んだとき、どのプロセスが死んだかをどうやって見つけられるでしょうか。
5. コードはどうやってリークを起こせるでしょうか。
6. ガベージコレクションの実行に時間がかかりすぎているかどうかをどうやって見つけられるでしょうか。

自由回答問題

1. プロセスの殺し忘れ、あるいはプロセスのメモリの使いすぎによって起こされたリークはどうやって確認できるでしょうか。
2. あるプロセスが 150MB のログファイルをバイナリモード開いてある情報を取り出し、その情報を ETS テーブルに保存しようとしています。バイナリメモリリークがあるとわかった場合に、そのノード上でのバイナリメモリの使用量を最小にするには何をすべきでしょうか。
3. ETS テーブルのサイズの成長が速すぎるのを検出するためには何を使ったら良いでしょうか。
4. ノードがフラグメンテーションで苦労しそうかを検出するためにどのような手順を踏めばよいでしょうか。またそれが NIF やドライバーがメモリリークの原因であろうという意見にどう反論できるでしょうか。
5. （message_queue_len の値を見ることで）メールボックスが大きくなっているプロセスがデータをリークさせている、あるいは新規メッセージをまったく処理できていなさそうかを、どうやって検出できるでしょうか。

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

6. 大きなメモリフットプリントのプロセスがガベージコレクションをほとんど実行していなさそうに見えます。これはどう説明できるでしょうか。
7. ノードに対するアロケーション戦略はいつ変更すべきでしょうか。この設定は手で調整すべきでしょうか、あるいはコードで記述すべきでしょうか。

ハンズオン

1. あなたが知っているあるいは運用しなければならないどんな Erlang システム（トイシステム含む）でもいいので、それを使ってそこにバイナリメモリリークがあるかどうかを判断してください。

第8章 CPUとスケジューラの大量消費

メモリリークはシステムを完全に停止させる傾向にありますが、CPUの枯渇はボトルネックのように振る舞う傾向があり、ノードの最大性能を制限します。Erlang開発者は、そういった問題に直面したとき水平スケールしようとするでしょう。基本的なコードが多ければ多いほど、スケールアウトが簡単なことが多いからです。中央集権的なグローバルの状態（プロセスレジストリ、ETSテーブルなど）だけは、システムの分散化に通常はコードの修正が必要です。[1]それでも、スケールアウトする前にローカルで最適化したい場合は、CPUとスケジューラの大量消費を見つけ出す必要があります。

Erlangノードの CPU使用率を適切に分析し、問題となる特定のコードを見つけ出すのは難しいとされています。あらゆるものが並行して仮想マシンの中にあるため、処理能力を食い尽くしているのが、ある特定のプロセス、ドライバー、自前のErlangコード、組み込まれたNIF、あるいはサードパーティのライブラリーなのかを見つけ出せる保証はありません。

既存のアプローチは大体限られており、コード内にある場合はプロファイリングとリダクションカウント[2]を確認する二つのアプローチ、そして、どこか他の場所（コード内も含む）にある場合はスケジューラの動作をモニタリングするアプローチがあります。

8.1 プロファイリングとリダクションカウント

前述したとおり、Erlangコード内の問題箇所を特定するアプローチは主に二つあります。一つは、次に示すアプリケーションを利用するような、古い標準的なプロファイリングの方法です。[3]

---

[1] 通常、これはシャーディングをするか、状態をレプリケーションする適切な方法を見つける形をとります。依然として大変な問題ではありますが、あなたのプログラムのセマンティクスの大部分を理解することに比べれば、分散システムへの適用は大した問題ではありません。何故なら、最初にErlangは分散システムを強要するからです。

[2] 訳注:プロセスごとにリダクションというカウンタがあり、このカウンターは通常一つの各関数呼び出しでインクリメントされます。このリダクション回数が最大値に達した時、コンテキストスイッチが発生します。http://erlang.org/doc/man/erlang.html#bump_reductions-1

[3] これらのプロファイラは殆ど制限なくErlangのトレース機能を使って動作します。そのため、ランタイム上のアプリケーションのパフォーマンスに影響します。本番環境では使わないほうが良いでしょう。

- `eprof`、[4]一番古い Erlang のプロファイラです。おおよそのパーセンテージ値とその所要時間を結果として出します。
- `fprof`、[5] eprof のよりも強力なプロファイラです。完全な並行性をサポートしており、詳細なレポートを出してくれます。レポート内容が深すぎるため、一般的に不透明で読みにくいとされています。
- `eflame`、[6] 新しいプロファイラの一つです。対象のコードの深い呼び出しシーケンスやホットスポットを可視化する flame graphs を生成します。最終結果を見るだけで問題をすぐに見つけることが出来ます。

各アプリケーションのドキュメントの熟読は読者にお任せします。もう一つのアプローチは、5.2.1 で紹介した `recon:proc_window/3` を実行する方法です。

```
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}}]}]
```

リダクションカウントは Erlang の関数呼び出しに直結していて、カウントが高いことは CPU 使用量が高いこととほぼ同義です。

この関数の興味深いところは、既にシステムがビジー状態のとき、[7]比較的短い間隔で試してみることです。繰り返し何度も実行して、上手くいけば、同じプロセス（あるいは同じ**種類**のプロセス）が常に上位に現れるパターンを確認できるはずです。

実行中のコードの位置[8]や現在の関数を使って、どの種類のコードが全スケジューラのリソースを独占しているか特定できるはずです。

8.2　システムモニター

もしプロファイリングやリダクションカウントの調査から何も目立つものが見つからない場合、NIF やガベージコレクションなどに行き着いている可能性があります。それらは常にリダクションカウントを正確に増加させるとは限らないため、これまでのやり方では表れず、実行時間の長さとしてのみ表れます。

[4] http://www.erlang.org/doc/man/eprof.html
[5] http://www.erlang.org/doc/man/fprof.html
[6] https://github.com/proger/eflame
[7] 5.1.2 を参照してください
[8] コードの位置は `recon:info(PidTerm, location)` または `process_info(Pid, current_stacktrace)` で取得してください。

そういったケースを見つけるために、一番の方法は erlang:system_monitor/2 を使い、long_gc と long_schedule を探すことです。前者はガベージコレクションが沢山の作業（時間がかかります！）を終えるたびに表示され、そして後者は NIF や他の理由のせいでスケジューラをなかなか手放さないビジープロセスの可能性にあるものを捕捉しやすくします。[9]

7.1.5 のガベージコレクションではシステムモニターをどのように設置するかを見てきましたが、私が以前長時間稼動しているアイテムを捉えるのに使った別のパターン[10]があります

```
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}}]
```

long_schedule と long_gc をそこそこ大きめな適切な値にしてください。この例では、1000 ミリ秒にします。あなたは exit(whereis(temp_sys_monitor), kill) を呼ぶ（リンクされているため次にシェルを殺すことになります）か、単にノードから切断する（リンクされているためプロセスを殺すことになります）だけでモニターを殺すことができます。

こういったコードとモニタリングは、長期間のログ保管に向いたストレージへレポートを送るモニタリング用のモジュールへと移せ、パフォーマンス劣化や過負荷の発見のためのカナリアとして使うことができます。

[9] 長いガベージコレクションはスケジュール時間に反映されます。システムによってはガベージコレクションが沢山の長いスケジュールに結びついている可能性は非常に高いです。

[10] もし 17.0 以降の新しいバージョンなら、この関数は名前つきフォームで再帰することで簡潔にできますが、それより古いバージョンの Erlang でも互換性を保てるようそのままにしておきます

8.2.1　一時停止しているポート

これまでのどの文にもあてはまらなかったけれどスケジューリングに関係のある、システムモニターのおもしろいところはポートに関するところです。あるプロセスが沢山のメッセージをポートに送りポートの内部キューが一杯になったとき、スペースに空きがでるまで Erlang のスケジューラーは強制的に送り側のスケジュールを解除します。これは VM から暗黙のバックプレッシャーを予想していなかったいくばくかのユーザーを驚かせることになるかもしれません。

こういったイベントは `busy_port` アトムをシステムモニターへ送ることでモニターできます。特にクラスタ化されたノードで、ノード間の通信がビジーポートによって処理されていた場合、リモートノード上のプロセスと通信しているローカルプロセスがスケジュール解除されるため、それを発見するのに `busy_dist_port` アトムが使われます。

もしそういった問題を抱えていることを発見した場合、クリティカルパスにある送信の関数をポートに送る場合は `erlang:port_command(Port, Data, [nosuspend])`、分散したプロセスの場合は `erlang:send(Pid, Msg, [nosuspend])` へと置き換えてみてください。メッセージを送れないことでスケジュール解除される場合すぐに知らせてくれます。

8.3　演習

復習問題

1. CPU 利用率に関する問題を特定するための主なアプローチ 2 つとは何ですか？
2. プロファイル用のツールの名前をいくつか挙げてください。本番環境で使用する場合、どの方法が好ましいですか？ またそれはなぜですか？
3. ロングスケジュールモニター[11]が、CPU やスケジューラの使いすぎを見つけるのに便利なのはなぜですか？

自由回答

1. ほとんど仕事をしない (リダクションカウンタを増やさない) プロセスが長期間スケジューリングされているのを見つけた場合、そのプロセスもしくは実行しているコードについて何が考えられますか？
2. あなたはシステムモニターを設定して、通常の Erlang コードでそれを起動できますか？ プロセスが平均しておよそどれぐらいの長さスケジューリングされているかを見つけるために、システムモニターを利用できますか？ 既存のシステムにすでにあるものよりもより適切にそれを行えるようなプロセスを、シェルから手動で手当たり次第に起動する必要があるかもしれません。

[11] システムモニター (erlang:system_monitor/2) で監視できる項目に long_schedule があります。これは NIF や driver で bump reductions をせずに CPU ガメるひとを見つけるための監視項目です。http://erlang.org/doc/man/erlang.html#system_monitor-2 も参考になるかと思います

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

トレースを行ういくつかの Erlang プログラムがあります。

- `sys`[2] は OTP に標準で付属されており、利用者はトレース機能のカスタマイズや、あらゆる種類のイベントのロギングなどができます。多くの場合、開発用として完全かつ最適です。一方で、IO をリモートシェルにリダイレクトしないですし、メッセージのトレースのレート制限機能を持たないため、本番環境にはあまり向きません。このモジュールのドキュメントを読むことをお勧めします。
- `dbg`[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十分です。問題点としては、**何をやっているのか知らないといけない**ということです。なぜなら `dbg` はノードのすべてをロギングすることや、2 秒もかからずにノードを落とすこともできるからです。
- **トレース BIF** は `erlang` モジュールの一部として提供されています。このリストの全ての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、抽象度が低いため、利用するのは困難です。
- `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、興味をひくもの

[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

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

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

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 `new` pid（関数呼び出しの時点ではまだ生成されていないプロセス）で指定できます。

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

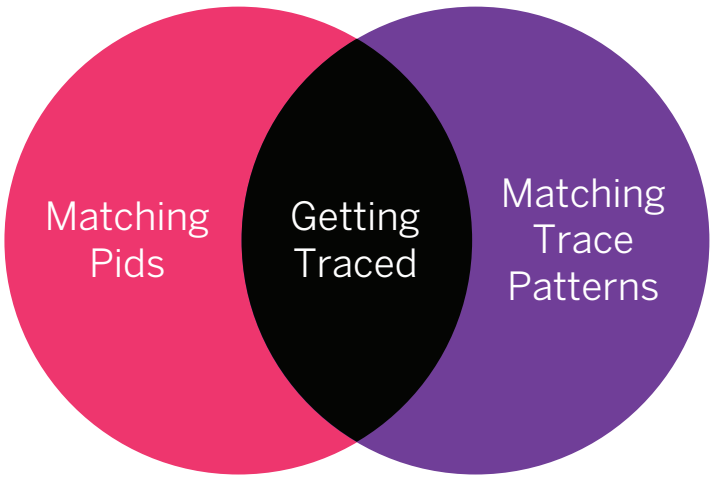


図 9.1　トレースされるのは、pid 指定とトレースパターンの交差した箇所です

[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

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

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

一方で 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、あなたがいじくりたいと思う面白い部分は、トレースするパターンの指定です。Recon ではいくつかの方法をサポートしています。

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

より賢明な方法は、アリティを引数のリストにマッチする関数で置き換えることです。その関数は ETS で利用できるもの[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。

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

より詳細には立ち入らず、ここではいくつかの例と、トレースの方法を見ていきます。

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

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

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

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

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

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

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

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

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

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

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

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

組み合わせることが可能なオプションはもう少しあります。

`{pid, PidSpec}`
: トレースするプロセスの指定です。有効なオプションは all, new, existing, あるいはプロセスディスクリプタ ({A,B,C}, "<A.B.C>", 名前をあらわすアトム、{global, Name}, {via, Registrar, Name}, あるいは pid) のどれかです。リストにすることで、複数指定することも可能です。

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

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

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

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

「うーん、このおかし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（全ての関数や 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、そのノード上のどんなプロセスも処理できないほど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 のトレース機能として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。