Stuff Goes Bad:
Erlang in Anger

Fred Hébert 著、elixir.jp 訳

2018 年 6 月 19 日

FRED HEBERT

STUFF GOES BAD: ERLANG IN ANGER

次の皆様のサポート、レビュー、そして編集に感謝します。

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

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

v1.1.0 *2016-04-06*

目次

図目次

はじめに

ソフトウェアを実行するにあたって

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

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

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

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

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

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

[1] 生命に関わるシステムは通常この議論の対象外です。

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

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

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

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

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

対象読者

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

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

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

本書の読み進め方

本書は二部構成です。

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

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

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

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

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

第 II 部では Erlang 衛生兵になって、既存の動作しているシステムに取り組みます。この部では実行中のノードへの接続方法の解説 (第**??**章)、取得できる基本的な実行時のメトリクス (第**??**章) を説明します。またクラッシュダンプを使ったシステムの検死方法 (第**??**章)、メモリリークの検出方法と修正方法 (第**??**章)、そして暴走した CPU 使用率の検出方法 (第**??**章) を説明します。最終章では問題がシステムを落としてしまう前に理解するために、本番環境での Erlang の関数呼び出しを `recon`[7]を使ってトレースする方法を説明します。(第 2 章)

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

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

第I部

Writing Applications

第1章

コードベースへの飛び込み方

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

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

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

1.1 生の Erlang

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

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

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

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

1.2 OTP アプリケーション

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

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

わずかな違いはあるかもしれませんが、一般的な構造は同じです。

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

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

そして

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

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

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

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

の 2 種類です。

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

1.2.1 ライブラリアプリケーション

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

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

1.2.2 標準アプリケーション

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

1. *appname*
2. *appname*_app

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

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

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

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

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

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

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

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

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

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

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

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

て使われます。

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

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

1.2.3 依存関係

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

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

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

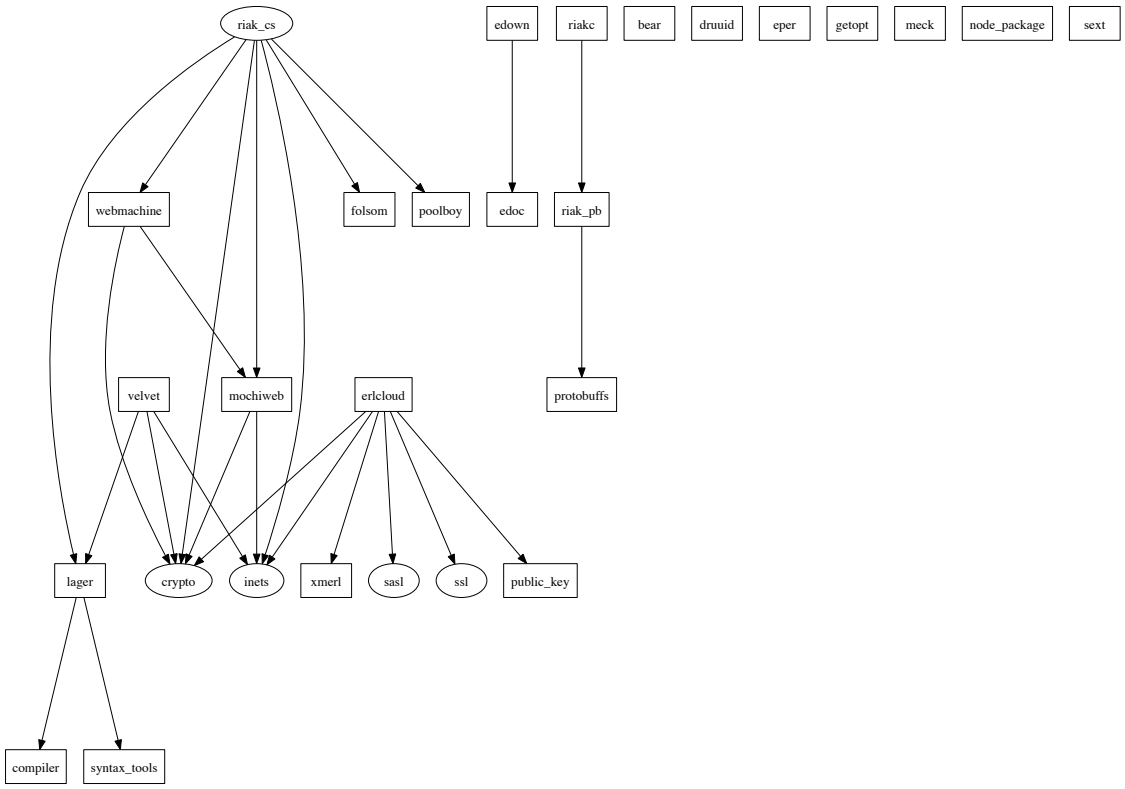

図 1.1　Basho のオープンソースクラウドライブラリである riak_cs の依存関係を表したグラフです。このグラフは kernel や stdlib といった必ず依存するようなものは除いています。楕円はアプリケーションで、四角はライブラリアプリケーションです。

1.3　OTP Releases

OTP releases are not a lot harder to understand than most OTP applications you'll encounter in the wild. A release is a set of OTP applications packaged in a production-ready manner so it boots and shuts down without needing to manually call `application:start/2` for any app. Compiled releases may contain their own copy of the Erlang virtual machine with more or less libraries than the default distribution, and can be ready to run standalone. Of course there's a bit more to releases than that, but generally, the same discovery process used for individual OTP applications will be applicable here.

You'll usually have a file named `relx.config` or a `relx` tuple in a `rebar.config` file, which will state which top-level applications are part of the release and some options regarding their

packaging. Relx-based releases can be understood by reading the project's wiki[8], or their documentation on the documentation sites of `rebar3`[9] or `erlang.mk`[10].

Other systems may depend on the configuration files used by `systools` or `reltool`, which will state all applications part of the release and a few[11] options regarding their packaging. To understand them, I recommend reading existing documentation on them.

1.4 Exercises

Review Questions

1. How do you know if a code base is an application? A release?
2. What differentiates an application from a library application?
3. What can be said of processes under a `one_for_all` scheme for supervision?
4. Why would someone use a `gen_fsm` behaviour over a `gen_server`?

Hands-On

Download the code at https://github.com/ferd/recon_demo. This will be used as a test bed for exercises throughout the book. Given you are not familiar with the code base yet, let's see if you can use the tips and tricks mentioned in this chapter to get an understanding of it.

1. Is this application meant to be used as a library? A standalone system?
2. What does it do?
3. Does it have any dependencies? What are they?
4. The app's `README` mentions being non-deterministic. Can you prove if this is true? How?
5. Can you express the dependency chain of applications in there? Generate a diagram of them?
6. Can you add more processes to the main application than those described in the `README`?

[8] https://github.com/erlware/relx/wiki
[9] https://www.rebar3.org/docs/releases
[10] http://erlang.mk/guide/relx.html
[11] A lot

第II部

Diagnosing Applications

第2章

トレース

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

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

トレースを行ういくつかの Erlang プログラムがあります。

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

---

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

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

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

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

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

2.1 Tracing Principles

The Erlang Trace BIFs allow to trace any Erlang code at all[8]. They work in two parts: *pid specifications*, and *trace patterns*.

Pid specifications lets the user decide which processes to target. They can be specific pids, `all` pids, `existing` pids, or `new` pids (those not spawned at the time of the function call).

The trace patterns represent functions. Functions can be specified in two parts: specifying the modules, functions, and arity, and then with Erlang match specifications[9] to add constraints to arguments.

What defines whether a specific function call gets traced or not is the intersection of both, as seen in Figure 2.1.

If either the pid specification excludes a process or a trace pattern excludes a given call, no trace will be received.

Tools like `dbg` (and trace BIFs) force you to work with this Venn diagram in mind. You specify sets of matching pids and sets of trace patterns independently, and whatever happens to be at the intersection of both sets gets to be displayed.

Tools like `redbug` and `recon_trace`, on the other hand, abstract this away.

---

[4] https://github.com/massemanet/eper/blob/master/doc/redbug.txt
[5] https://github.com/massemanet/eper
[6] http://ferd.github.io/recon/recon_trace.html
[7] メッセージのトレース機能は将来のバージョンでサポートされるかもしれません。ライブラリの著者は OTP を使っている時には必要性を感じておらず、またビヘイビアと特定の引数へのマッチングにより、ユーザはおよそ同じことを実現できます
[8] In cases where processes contain sensitive information, data can be forced to be kept private by calling `process_flag(sensitive, true)`
[9] http://www.erlang.org/doc/apps/erts/match_spec.html

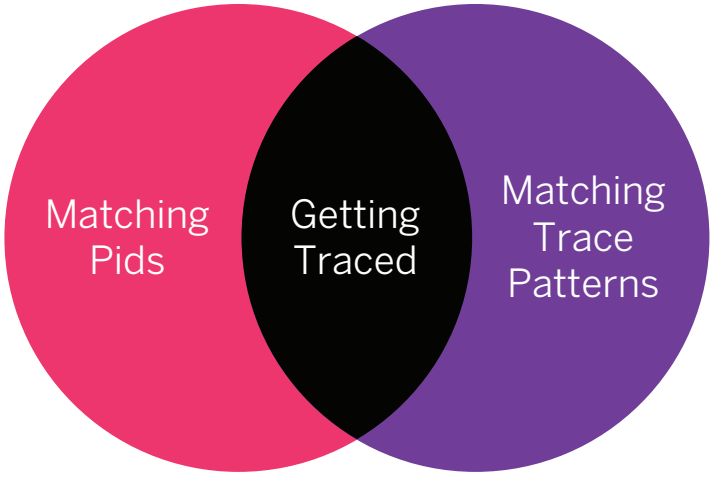

図 2.1　What gets traced is the result of the intersection between the matching pids and the matching trace patterns

2.2　Tracing with Recon

Recon, by default, will match all processes. This will often be good enough for a lot of debugging cases. The interesting part you'll want to play with most of the time is specification of trace patterns. Recon support a few basic ways to declare them.

The most basic form is `{Mod, Fun, Arity}`, where `Mod` is a literal module, `Fun` is a function name, and `Arity` is the number of arguments of the function to trace. Any of these may also be replaced by wildcards (`'_'`). Recon will forbid forms that match too widely on everything (such as `{'_','_','_'}`), as they could be plain dangerous to run in production.

A fancier form will be to replace the arity by a function to match on lists of arguments. The function is limited to those usable by match specifications similar to what is available in ETS[10]. Finally, multiple patterns can be put into a list to broaden the matching scope.

It will also be possible to rate limit based on two manners: a static count, or a number of matches per time interval.

Rather than going more in details, here's a list of examples and how to trace for them.

```
%% All calls from the queue module, with 10 calls printed at most:
recon_trace:calls({queue, '_', '_'}, 10)
```

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

```
%% All calls to lists:seq(A,B), with 100 calls printed at most:
recon_trace:calls({lists, seq, 2}, 100)

%% All calls to lists:seq(A,B), with 100 calls per second at most:
recon_trace:calls({lists, seq, 2}, {100, 1000})

%% All calls to lists:seq(A,B,2) (all sequences increasing by two) with 100 calls
%% at most:
recon_trace:calls({lists, seq, fun([_,_,2]) -> ok end}, 100)

%% All calls to iolist_to_binary/1 made with a binary as an argument already
%% (a kind of tracking for useless conversions):
recon_trace:calls({erlang, iolist_to_binary,
                   fun([X]) when is_binary(X) -> ok end},
                  10)

%% Calls to the queue module only in a given process Pid, at a rate of 50 per
%% second at most:
recon_trace:calls({queue, '_', '_'}, {50,1000}, [{pid, Pid}])

%% Print the traces with the function arity instead of literal arguments:
recon_trace:calls(TSpec, Max, [{args, arity}])

%% Matching the filter/2 functions of both dict and lists modules, across new
%% processes only:
recon_trace:calls([{dict,filter,2},{lists,filter,2}], 10, [{pid, new}])

%% Tracing the handle_call/3 functions of a given module for all new processes,
%% and those of an existing one registered with gproc:
recon_trace:calls({Mod,handle_call,3}, {1,100}, [{pid, [{via, gproc, Name}, new]}

%% Show the result of a given function call, the important bit being the
%% return_trace() call or the {return_trace} match spec value.
recon_trace:calls({Mod,Fun,fun(_) -> return_trace() end}, Max, Opts)
recon_trace:calls({Mod,Fun,[{'_', [], [{return_trace}]}]}, Max, Opts)
```

Each call made will override the previous one, and all calls can be cancelled with `recon_trace:clear/0`.

There's a few more combination possible, with more options:

`{pid, PidSpec}`

: Which processes to trace. Valid options is any of `all`, `new`, `existing`, or a pro-

cess descriptor (`{A,B,C}`, `"<A.B.C>"`, an atom representing a name, `{global, Name}`, `{via, Registrar, Name}`, or a pid). It's also possible to specify more than one by putting them in a list.

`{timestamp, formatter | trace}`

By default, the formatter process adds timestamps to messages received. If accurate timestamps are required, it's possible to force the usage of timestamps within trace messages by adding the option `{timestamp, trace}`.

`{args, arity | args}`

Whether to print the arity in function calls or their (by default) literal representation.

`{scope, global | local}`

By default, only 'global' (fully qualified function calls) are traced, not calls made internally. To force tracing of local calls, pass in `{scope, local}`. This is useful whenever you want to track the changes of code in a process that isn't called with `Module:Fun(Args)`, but just `Fun(Args)`.

With these options, the multiple ways to pattern match on specific calls for specific functions and whatnot, a lot of development and production issues can more quickly be diagnosed. If the idea ever comes to say "hm, maybe I should add more logging there to see what could cause that funny behaviour", tracing can usually be a very fast shortcut to get the data you need without deploying any code or altering its readability.

2.3 Example Sessions

First let's trace the queue:new functions in any process:

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

The limit was set to 1 trace message at most, and recon let us know when that limit was reached.

Let's instead look for all the `queue:in/2` calls, to see what it is we're inserting in queues:

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

```
Recon tracer rate limit tripped.
```

In order to see the content we want, we should change the trace patterns to use a fun that matches on all arguments in a list (_) and returns `return_trace()`. This last part will generate a second trace for each call that includes the return value:

```
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.
```

Matching on argument lists can be done in a more complex manner:

```
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
```

Note that in the pattern above, no specific function (`'_'`) was matched against. Instead, the fun used restricted functions to those having two arguments, the first of which is either a

list or an integer greater than 1.

Be aware that extremely broad patterns with lax rate-limitting (or very high absolute limits) may impact your node's stability in ways `recon_trace` cannot easily help you with. Similarly, tracing extremely large amounts of function calls (all of them, or all of `io` for example) can be risky if more trace messages are generated than any process on the node could ever handle, despite the precautions taken by the library.

In doubt, start with the most restrictive tracing possible, with low limits, and progressively increase your scope.

2.4 Exercises

Review Questions

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

Open-ended Questions

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

Hands-On

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

1. Can chatty processes (`council_member`) message themselves? (*hint: can this work with registered names? Do you need to check the chattiest process and see if it messages itself?*)

2. Can you estimate the overall frequency at which messages are sent globally?
3. Can you crash a node using any of the tracing tools? (*hint: dbg makes it easier due to its greater flexibility*)

おわりに

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

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

みなさんのシステム運用がうまく行くことを願っています。