FRED HEBERT

STUFF GOES BAD: ERLANG IN ANGER

Благодарность за дополнительную работу, вычитку и редактирование:

Jacob Vorreuter, *Seth Falcon*, *Raoul Duke*, *Nathaniel Waisbrot*, *David Holland*, *Alisdair Sullivan*, *Lukas Larsson*, *Tim Chevalier*, *Paul Bone*, *Jonathan Roes*, и *Roberto Aloi*.

Изображение на обложке является изменённой версией ядерного убежища под авторством drouu с сайта sxc.hu.

v1.0.3 *2014-10-28*

Contents

List of Figures

Вступление

О выполнении приложений

В Erlang есть кое-что уникальное в его подходе к сбоям, если сравнить его с большинством других языков. Существует такой общепринятый ход мыслей, при следованию которому и сам язык, и окружение, в котором работает программист, и методология делают всё возможное, чтобы предотвратить ошибки. Если во время исполнения что-то может пойти не так, то это нужно предотвратить, а если предотвращение невозможно, то оно выходит за пределы любого другого решения, о котором могли бы подумать люди.

Программа пишется один раз и затем отдаётся на производство (*production*), что бы там с ней ни произошло. Если случатся ошибки, то придётся приготовить и доставить заказчику новые версии.

Erlang, с другой стороны, использует такой подход, при котором сбои считаются неизбежными, независимо от того, что явилось их причиной — разработчик, оператор или аппаратное обеспечение. Редко считается практичным избавляться от всех ошибок в программе или системе[1] Если вы можете справиться с ошибками вместо того, чтобы любой ценой их не допустить, то подавляющее количество неожиданных поведений программы может быть разрешено с помощью этого подхода «справься с ситуацией сам».

Вот откуда появилась известная идея «Дай ему упасть» (*Let it crash*)[2]: Потому что вы теперь готовы справиться с неудачным завершением работы алгоритма, и поскольку стоимость выведения всех сложных ошибок из системы до момента её сдачи часто является заоблачной, программисты должны помнить и обрабатывать только те ошибки, к которым они готовы, а остальные ситуации позволить решать другому процессу (наблюдателю) или виртуальной машине.

Поскольку большая часть ошибок являются временными[3], простой перезапуск процессов в состояние, известное ранее как стабильное, при обнаружении ошибки может оказаться

[1] Обычно в эту категорию не входят жизненно-важные системы

[2] Люди из мира Erlang чаще предпочитают альтернативное «Дай ему завершиться неудачей» (*Let it fail*), поскольку такое словосочетание меньше нервирует других людей.

[3] 131 из 132 ошибок являются временными (они недетерминированны и исчезают, как только вы на них пристально посмотрите, а повторная попытка сделать то же самое часто завершается успешно), согласно Jim Gray в его статье Почему компьютеры останавливаются и что с этим делать? (на английском).

удивительно удачной стратегией.

Erlang является программным окружением, в котором выбранный подход аналогичен иммунной системе человеческого тела, когда большая часть языков программирования только беспокоятся о гигиене, и блокируют доступ бактерий к телу. Обе формы кажутся мне очень важными. Почти любое окружение предлагает различные степени гигиены. Но почти никакое из окружений не предлагает аналог иммунной системы, в которой ошибки времени выполнения решаются на месте и расцениваются как некритичные, после которых система продолжает работу.

Поскольку система не обрушивается при первом чихе в её сторону, Erlang/OTP также позволяет вам стать доктором. Вы можете зайти в систему, открыть крышку прямо на производственном (*production*) сервере, осторожно изучить всё внутри, не прерывая работы, и даже попытаться интерактивно починить его. Чтобы продолжить эту цепочку аналогий, Erlang позволояет вам выполнять широкий спектр тестов для диагностики проблем и проводить хирургические операции (даже на сердце), без необходимости для пациента ложиться на стол, садиться или прерывать его каждодневные обычные дела.

Эта книга задумана в качестве небольшой инструкции на тему того, как стать доктором для Erlang-систем в военное время. Это в первую очередь коллекция подсказок и секретов, которые помогут понять, откуда приходят сбои, а также набор фрагментов кода и методик, которые уже помогли другим разработчикам при отладке их рабочих Erlang-систем.

Для кого предназначена эта книга?

Эта книга не предназначена для начинающих. Существует некоторая дистанция между существующими уроками, книгами, учебными занятиями и умением управлять, диагностировать и отлаживать работающие системы после того, как они попали в производство (*production*). Есть некий этап неуверенного нащупывания решений, присущий изучению программистом нового языка и окружения, когда им приходится просто разобраться, как выбираться из пошаговых инструкций и идти в реальный мир к которому в нагрузку прилагается сообщество других людей.

Эта книга предполагает, что читатель хорошо освоил базовый Erlang и систему OTP. Возможности Erlang/OTP описываются здесь по желанию автора — обычно когда автор считает, что в данной ситуации кроется некая хитрость — и ожидается, что читатель, которого вдруг собьёт с толку некоторый материал об Erlang/OTP, знает где искать пояснения при необходимости[4].

Вот что здесь явно не потребуется — так это то, что читатель должен знать об отладке программ на Erlang, уметь копаться в существующем исходном коде, диагностировать проблемы или иметь представление о лучших практиках установки Erlang-программ в производственном окружении[5].

[4] Я рекомендую посетить сайт Изучай Erlang во имя добра (имеется в продаже русский перевод книги на сайте издательства ДМК Пресс), а также Стандартную документацию Erlang.

[5] Заметьте, что запуск Erlang в экране screen или tmux *не* является стратегией установки программы.

Как читать эту книгу

Книга разделена на две части.

Часть I сосредоточена на методике написания приложений. Она включает рекомендации по работе с существующим исходным кодом (Глава 1), общие советы по написанию программ с открытым исходным кодом (*open source*) на языке Erlang (Глава 2), и как учитывать чрезвычайную нагрузку в будущем при проектировании вашей системы (Глава 3).

Часть II сосредоточена на вашей роли в качестве Erlang-доктора и касается существующих, живых систем. Она содержит инструкции о том, как подключиться к работающему узлу (Глава 4), и перечень доступных во время выполнения метрик (Глава 5). Здесь также объясняется как выполнить посмертное вскрытие погибшей системы с помощью аварийного дампа (*crash dump*) (Глава 6), как определить и исправить утечки памяти (Глава 7), и как найти украденное процессорное время (Глава 8). Последняя глава содержит инструкции о том, как трассировать вызовы функций Erlang на живой производственной системе с помощью приложения `recon`[6] чтобы понять причины проблем до того, как они приведут к падению системы (Глава 9).

После каждой главы имеются несколько необязательных упражнений в виде вопросов или задач, которые можно попробовать решить для того, чтобы проверить ваше понимание материала, или если вы хотите узнать чуточку больше по каждой данной теме.

[6] http://ferd.github.io/recon/ — библиотека, содержащая функции в целом безопасные для использования на работающих системах и помогающая сделать текст книги короче и понятнее.

Part I

Написание приложений

Chapter 1

Как нырять в чужой исходный код

Один из самых раздражающих комментариев, которые можно услышать в ответ на любой вопрос — это «Читай исходники», но в случае с Erlang-программистами вам придётся делать это довольно часто. Либо окажется, что документация к библиотеке неполна, устарела или отсутствует. В других случаях Erlang-программисты чем-то подобны программистам на языке Lisp, в том, что они часто пишут библиотеки, решающие их собственную проблему, и не тестируют их в других ситуациях, оставляя вам свободу расширения или удовольствие исправления проблем, которые вылезают при использовании кода в новом контексте.

Таким образом практически гарантированно вам придётся нырять в чьи-то исходные коды, о которых вам ничего не известно, потому что вы унаследовали их на новом месте работы или поскольку вам нужно починить что-то в коде или понять принципы его работы, чтобы продолжить работу над вашей собственной системой. Это утверждение на самом деле истинно для многих языков, когда ваш проект не спроектирован вами самими.

Есть три основных вида исходных кодов на Erlang, которые могут вам встретиться в дикой природе: базовый Erlang, приложения OTP и релизы OTP. В этой главе мы посмотрим на каждый из них и попробуем предоставить полезные подсказки для навигации по ним.

1.1 Ныряем в сырой Erlang

Если вам встретился исходный код чего-нибудь на базовом (сыром) Erlang, вам остаётся надеяться только на самих себя. Такой код редко следует принятым стандартам и вам придётся нырять в него по-старинке, чтобы попытаться понять, что в коде происходит.

Это означает надеяться на наличие файла `README.md` или чего-то подобного, где можно найти точку входа в приложение, и начать оттуда. Или, может, удастся найти контактные данные и связаться с автором, чтобы задать ему вопросы по его коду.

К счастью такой код вам может встретиться не так часто, и обычно это работы новичков или отличные проекты, которые были когда-то начаты новичками, а теперь требуют серьёзного

переписывания. В общих словах, появление инструментов таких, как `rebar`[1] сделало так, что многие люди начали создавать приложения OTP.

1.2 Ныряем в OTP-приложения

Разбираться с OTP-приложениями обычно несложно. Часто они имеют одинаковую структуру директорий, которая выглядит так:

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

Могут быть небольшие отличия, но основная структура будет одинакова для всех.

Каждое приложение OTP должно обязательно содержать *app-файл*, который находится в `ebin/<ИмяПриложения>.app` либо чаще, `src/<ИмяПриложения>.app.src`[2]. Существует два основных вида app-файлов:

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

И такой:

```
{application, dispcount, [
  {description, "A dispatching library for resources and task "
                "limiting based on shared counters"},
  {vsn, "1.0.0"},
```

[1] https://github.com/rebar/rebar/ — инструмент для сборки приложений, кратко описанный в Главе 2

[2] Система сборки генерирует окончательный файл, который затем помещается в папку `ebin`. Заметьте что в этих случаях многие файлы `src/<ИмяПриложения>.app.src` не перечисляют все модули проекта, а позволяют системе сборки позаботиться об этом.

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

Первый вариант называется *библиотечным приложением*, а второй — обычным *приложением*.

1.2.1 Библиотечные приложения

Библиотечные приложения обычно содержат модули, названные однообразно в стиле `имяприложения_чтото`, и один модуль с именем `имяприложения`. Обычно это интерфейсный модуль, являющийся центральным для всей библиотеки и предоставляет быстрый доступ к имеющимся функциям библиотеки.

Глядя на исходный текст модуля можно понять, как он работает, без особых усилий: если модуль соответствует некоторому данному поведению (например, `gen_server`, `gen_fsm`, и так далее), от вас ожидается запуск процесса под одним из ваших наблюдателей и вызов его стандартным способом. Если поведение не указано, то, вероятнее всего, это функциональная библиотека без состояния. Для таких случаев взгляд на экспортированные функции должен дать вам возможность понять его назначение.

1.2.2 Обычные приложения

Для обычного приложения OTP потенциально имеются два модуля, которые могут действовать в качестве точки входа:

1. `имяприложения`
2. `имяприложения_app`

Первый файл должен вести себя подобно тому, как он вёл себя в библиотечном приложении (точка входа), тогда как второй будет реализовать поведение `application`, и представлять собой верхушку иерархии процессов приложения. В некоторых случаях первый файл будет играть обе роли одновременно.

Если вы планируете просто добавить приложение в качестве зависимости для вашей программы, то смотрите внутрь модуля `имяприложения` для получения подробностей. Если же вам нужно обслуживать и/или исправить что-то в приложении, то вместо этого смотрите модуль `имяприложения_app`.

Приложение запустит наблюдателя верхнего уровня и возвратит его *идентификатор процесса*. Этот наблюдатель затем будет содержать спецификации всех дочерних процессов, которые он запустит уже самостоятельно[3].

Чем выше процесс находится в дереве наблюдения, тем более вероятно он окажется жизненно важным для работы приложения. Вы также можете оценить важность процесса по порядку их запуска (все подчинённые процессы в дереве наблюдения запускаются по порядку, начиная с самых глубоко вложенных). Если процесс запущен позднее в дереве наблюдения, то вероятно он зависит от ранее запущенных процессов.

Более того, рабочие процессы, которые зависят друг от друга внутри одного приложения (скажем, процесс, который собирает в буфер данные, прилетающие из сокета, и передаёт их в конечный автомат, который отвечает за разбор протокола) вероятно будут сгруппированы внутри одного наблюдателя и должны падать все вместе, если что-то идёт не так. Это добровольный выбор программиста, поскольку обычно проще начинать работу с чистым состоянием, перезапуская оба процесса вместо того, чтобы пытаться выяснить, как заживить раны, нанесённые утратой или повреждением состояния другим процессом из связки.

Стратегия перезапуска процессов наблюдателем отражает отношение между процессами, которые находятся под наблюдением:

- `one_for_one` и `simple_one_for_one` используются для тех процессов, которые не зависят прямо друг от друга, хотя их отказы посчитаются в пользу полной остановки приложения[4].
- `rest_for_one` будет использован для процессов, которые зависят линейно друг от друга.
- `one_for_all` используется когда в связке процессов все зависят от всех.

Эта структура означает, что легче всего изучать OTP-приложения сверху вниз, следуя по ветвям деревьев наблюдения.

Для каждого рабочего процесса под наблюдением поведение, которое он реализует, даст хорошую подсказку о его назначении:

- `gen_server` хранит некоторые ресурсы и обычно следует модели поведения клиент/сервер (или, обобщённо, шаблон поведения запрос/ответ)
- `gen_fsm` будет обрабатывать последовательность событий или входных данных и реагировать на них следуя модели поведения конечного автомата. Часто конечные автоматы используются для реализации протоколов.
- `gen_event` будет вести себя как обработчик событий и центр регистрации обратных вызовов, или как способ справиться с уведомлениями какого-нибудь вида.

[3] В некоторых случаях наблюдатель не описывает никаких дочерних процессов: тогда они либо будут запущены динамически с помощью какой-нибудь функции в API, либо на этапе запуска приложения, либо наблюдатель присутствует здесь только для того, чтобы позволить загрузить переменные окружения OTP (из кортежа `env` в app-файле).

[4] Некоторые разработчики будут использовать наблюдатели `one_for_one` когда на самом деле лучше бы подошла стратегия `rest_for_one`. Такие процессы требуют строгого порядка при запуске, но забывают об этом порядке, когда дело доходит до перезапуска, или если предыдущий процесс в связке умирает.

Все эти модули будут иметь некоторую схожую структуру: экспортированные функции, которые представляют интерфейс пользователя, ещё экспортированные функции, которые реализуют некоторый модуль обратных вызовов (поведение), приватные функции, обычно в таком порядке.

На основании их отношений в дереве наблюдателя и типичной роли каждого из поведений, глядя на интерфейс, который будет использован другими модулями и реализованные поведения, можно раскрыть много информации о той программе, в которую вы ныряете.

1.2.3 Зависимости

Все приложения имеют зависимости[5], и эти зависимости могут в свою очередь тоже иметь зависимости. Приложения OTP обычно не разделяют состояния между собой, так что нельзя узнать, какой код зависит от какого просто глядя на app-файл, и предполагая, что они написаны автором в самой корректной манере. Рисунок 1.1 показывает диаграмму, которая может быть построена глядя на app-файлы, чтобы помочь лучше понять структуру OTP-приложений.

Используя такую иерархию и глядя на короткое описание каждого из приложений, может оказаться полезным нарисовать приблизительную грубую карту где что находится. Для создания подобной диаграммы найдите в приложении `recon` директорию сценариев `script/` и вызовите `escript script/app_deps.erl`[6]. Подобные иерархии можно найти используя `observer`[7], но для отдельных деревьев наблюдения. Если сложить это вместе, у вас может получиться удобный способ быстро нарисовать и разобраться, что происходит в незнакомом коде.

[5] Как минимум — это зависимость от приложений `kernel` и `stdlib`.

[6] Этот скрипт требует наличия у вас в системе программы graphviz

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

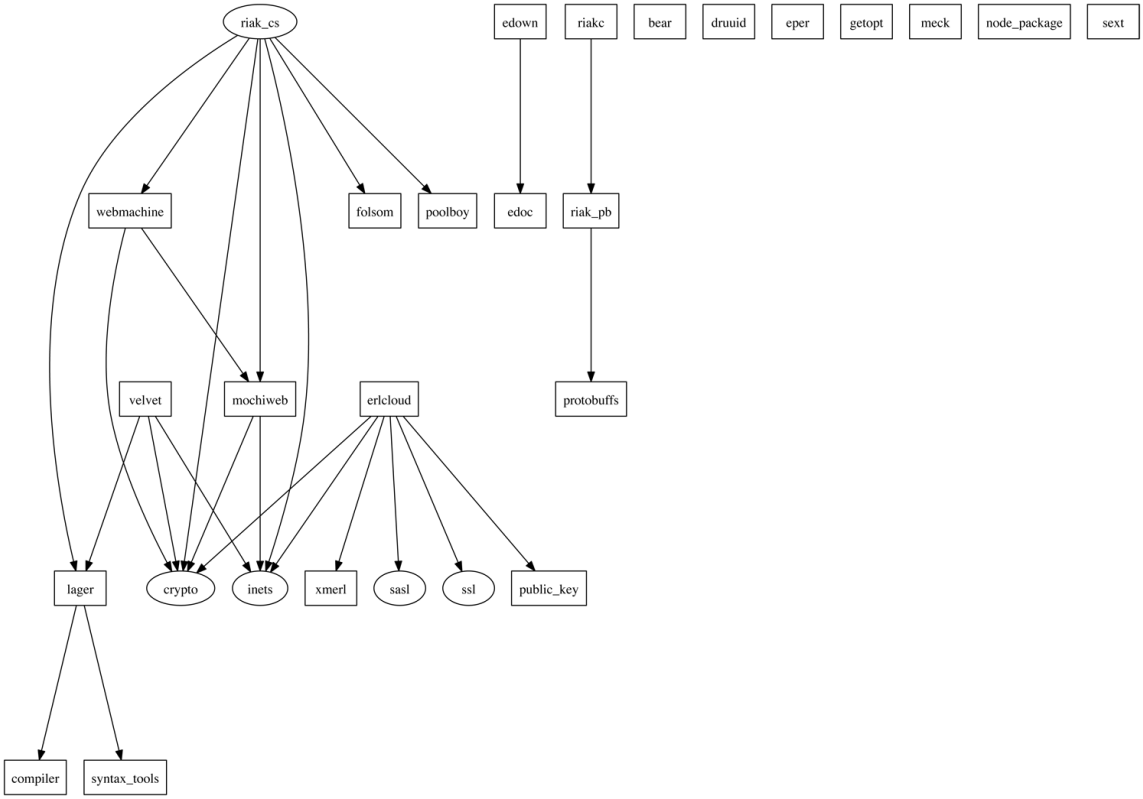


Figure 1.1: Граф зависимостей riak_cs, облачной библиотеки с открытым исходным кодом, написанной компанией Basho. Граф игнорирует зависимости от общих приложений, таких как kernel и stdlib. Овалами показаны приложения, прямоугольниками — библиотечные приложения.

1.3 Ныряем в OTP-релизы

OTP-релизы не сильно сложнее для понимания, чем обычные OTP-приложения, которые могут вам встретиться в дикой природе. Релиз — это набор OTP-приложений, упакованных для доставки на производственные (*production*) сервера так, что они запускаются и завершают работу без ручного вызова `application:start/2` в любом нужном вам приложении. Конечно есть и другие подробности касательно релизов, но обычно применяется тот же процесс исследования, как и для обычных приложений OTP.

Обычно у вас будет файл, похожий на файлы конфигурации, которыми пользуется `systools` или `reltool`, который содержит перечень всех приложений, включённых в релиз и несколько[8] параметров касательно способа их упаковки. Для лучшего их понимания я рекомендую обратиться к чтению существующей документации и книгам. Если вам повезло, проект мог быть собран с помощью `relx`[9], более удобного инструмента, который был официально выпущен в начале 2014.

1.4 Упражнения

Вопросы для закрепления материала

1. Как вы можете узнать, что данный код является приложением? Релизом?
2. Что отличает обычное приложение от библиотечного?
3. Что можно сказать о процессах, которые запускаются под наблюдателем по схеме `one_for_all`?
4. Почему кто-то может решить использовать поведение `gen_fsm` вместо `gen_server`?

Тренировочные задания

Скачайте код отсюда: https://github.com/ferd/recon_demo. Этот проект будет использоваться как основа для упражнений по всей книге. Подразумевается, что вы пока не знакомы с этим кодом, давайте посмотрим, как вы воспользуетесь знаниями, полученными в текущей главе и разберётесь что к чему.

1. Задумано ли это приложение как библиотека или как отдельно работающая система?
2. Что оно делает?
3. Есть ли у него зависимости? Какие?
4. Файл `README` упоминает что-то о недетерминированности. Можете ли вы найти доказательства, что это правда? Как?

[8] На самом деле множество.
[9] https://github.com/erlware/relx/wiki

5. Можно ли здесь выразить цепочку зависимостей приложений? Создать диаграмму?

6. Можно ли добавить новые процессы к главному приложению кроме тех, что были описаны в файле `README`?

Chapter 2

Построение проектов с открытым исходным кодом на Erlang

Большинство книг по Erlang стараются объяснить, как построить приложения на Erlang/OTP, но совсем немногие углубляются в вопросы взаимодействия с сообществом, которое работает над приложениями с открытым исходным кодом (*open source*). Некоторые даже специально избегают эту тему. Эта глава посвящена небольшой экскурсии по состоянию отношений между людьми и программами в Erlang.

OTP-приложения являются подавляющим большинством исходных кодов, которые вы можете встретить. На самом деле многие люди, кому нужно собрать OTP-релиз, делают это в виде ещё одного приложения-зонтика, под которым размещаются все остальные части проекта.

Если то, что вы пишете, является отдельным набором файлов и кода, которые могут пригодиться кому-то ещё при построении их продукта, то вероятнее всего это будет оформлено, как приложение OTP.

Основные инструменты для сборки, которые получают поддержку авторов и сообщества — это `rebar` и `erlang.mk`. Первое — это портативный самодостаточный скрипт на языке Erlang, который является интерфейсом к ряду стандартных функций и добавляет некоторые свои, тогда как второе — хитро закрученный стандартный сборочный файл (*Makefile*), который делает немного меньше, но обычно выполняет компиляцию быстрее. В этой главе я в основном сосредоточусь на сборке с помощью `rebar`, поскольку он принят почти всеми, как стандарт, хорошо известен, а приложения, использующие `erlang.mk`, обычно поддерживаются при сборке с помощью `rebar` в качестве зависимостей.

2.1 Структура проекта

Структуры приложений и релизов OTP отличаются. Ожидается, что OTP-приложение имеет один наблюдатель верхнего уровня (или ни одного) и, возможно, ряд зависимостей, которые находятся под ним. OTP-релиз обычно состоит из ряда OTP-приложений, которые могут зависеть (или

не зависеть) друг от друга. Это является причиной существования двух очень разных структур директорий.

2.1.1 OTP-приложения

Для OTP-приложений правильной структурой является точно такая же, как мы рассмотрели ранее в секции 1.2:

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

Какие появились отличия — это директория `deps/`, которую полезно иметь и она будет создана автоматически при работе с `rebar`[1] по мере необходимости. Это происходит потому, что Erlang не имеет стандартного управления пакетами зависимостей. Вместо этого люди решили использовать `rebar`, который скачивает зависимости локально для каждого проекта свои. Это приемлемое решение и убирает массу проблем с конфликтами, но означает, что каждому проекту придётся скачивать свой набор зависимостей.

Зависимости для `rebar` задаются с помощью добавления строк в специальный файл `rebar.config`, которого ваш проект может не иметь — тогда создайте его сами:

```
{deps,
 [{application_name, "1.0.*",
   {git, "git://github.com/user/myapp.git", {branch,"master"}}},
  {application_name, "2.0.1",
   {git, "git://github.com/user/hisapp.git", {tag,"2.0.1"}}},
  {application_name, "",
   {git, "https://bitbucket.org/user/herapp.git",  "7cd0aef4cd65"}},
  {application_name, "my regex",
   {hg, "https://bitbucket.org/user/theirapp.hg" {branch, "stable"}}}]}.
```

[1] Многие люди пакуют `rebar` прямо в свои приложения. Изначально это помогало другим людям, которые никогда ранее им не пользовались, начать немедленно им пользоваться. На ваш вкус можно установить rebar глобально в вашей системе, или хранить локальную копию, если вам требуется особенная его версия.

Приложения и их зависимости скачиваются рекурсивно с помощью стандартных утилит `git` (или `hg`, или `svn`) с указанных серверов. Затем они готовы к сборке и вы можете добавить особые параметры компиляции с помощью строки `{erl_opts, СписокПараметров}.` в файле конфигурации[2].

В пределах этих директорий вы можете выполнять обычную разработку вашего OTP-приложения. Для компиляции вызовите команду `rebar get-deps compile`, которая докачает недостающие зависимости и затем соберёт всё вместе с вашим приложением.

Когда вы публикуете ваше приложение для всех желающих, то зависимости *не включаются* в архив. Вполне возможна ситуация, когда приложения другого разработчика зависят от тех же приложений, что и ваше, и нет смысла поставлять ему наши копии зависимостей. Система сборки на компьютере другого разработчика (в нашем случае, `rebar`) сможет разобраться, какие зависимости нужны и докачает недостающие.

2.1.2 Строим OTP-релизы

Для релизов структура должна немного отличаться[3]. Релизы — это наборы приложений, и их структура должна отражать этот факт.

Вместо того, чтобы делать одно приложение верхнего уровня, следует разместить все приложения на один уровень глубже и разделить на две категории: собственно приложения (*apps*) и зависимости (*deps*). Директория `apps/` содержит исходный код вашего приложения (скажем, ваша бизнес-логика), а директория `deps/` содержит ваши зависимости, управляемые другими авторами или командами.

```
apps/
doc/
deps/
LICENSE.txt
README.md
rebar.config
```

Эта структура годится для генерации релизов. Инструменты, такие как Systool и Reltool были описаны в других публикациях[4], и могут дать пользователю большую власть. Более лёгкий инструмент, который появился совсем недавно — это `relx`[5].

[2] Можно узнать подробнее, если выполнить `rebar help compile`

[3] Я пишу *должна* потому что многие Erlang-разработчики размещают свою готовую программу в единственном приложении на верхнем уровне проекта (в папке `src/`) и ряд вспомогательных приложений оформляются, как зависимости (в папке `deps/`), что не совсем идеально для нужд распространения программы и конфликтует с рекомендуемой стандартом OTP структурой директорий. Люди, которые так делают, обычно собирают код из исходников при установке на производственные сервера и выполняют особые самодельные команды для их запуска.

[4] http://learnyousomeerlang.com/release-is-the-word

[5] https://github.com/erlware/relx/wiki

Файл конфигурации для `relx`, описывающий структуру директорий, показанную выше, будет выглядеть так:

```
{paths, ["apps", "deps"]}.
{include_erts, false}. % релиз будет использовать установленный в системе Erlang
{default_release, demo, "1.0.0"}.

{release, {demo, "1.0.0"},
    [members,
     feedstore,
     ...
     recon]}.
```

Вызов команды `./relx` (если исполняемый файл relx находится в текущей директории) начнёт сборку релиза, который будет помещён в директорию `_rel/`. Если вам нравится пользоваться программой `rebar`, то вы можете построить релиз в процессе компиляции проекта используя такую строчку в `rebar.config`:

```
{post_hooks,[{compile, "./relx"}]}.
```

Тогда каждый раз, когда вы вызовете команду `rebar compile`, будет создаваться релиз.

2.2 Наблюдатели и семантика start_link

В сложной производственной (*production*) системе многие сбои и ошибки являются временными, и повторная попытка выполнить операцию — это хороший подход к ведению дел, согласно работе Джима Грэй [6], которая цитирует, что *среднее время наработки до отказа* (*Mean Times Between Failures или сокращённо MTBF*) систем, которые обрабатывают такие временные ошибки, улучшается в 4 раза. И всё же тема наблюдателей затрагивает больше, чем просто перезапуски.

Одним из важных фактов о наблюдателях в Erlang и их деревьях наблюдения является то, что *фазы их запуска синхронны*. Каждый процесс в OTP имеет возможность заблокировать запуск своих дочерних и сестринских процессов. Если процесс завершается аварийно (умирает), то попытка повторяется снова и снова, пока он не запустится успешно или наблюдатель не сочтёт, что процесс падает слишком часто.

Здесь люди часто делают одну общую ошибку. Нет никакой паузы перед тем, как наблюдатель перезапускает упавший дочерний процесс. Когда сетевое приложение пытается установить подключение во время фазы своей инициализации в то время, как удалённый сервис не работает,

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

приложение не может загрузиться после ряда безуспешных перезапусков. Затем система останавливается целиком.

Много Erlang-разработчиков скатываются в споры о создании наблюдателя, который бы имел период ожидания между перезапусками. Я выступаю против этой идеи по одной простой причине: *дело в том, какие вы получаете гарантии*.

2.2.1 Всё дело в гарантии

Перезапуск процесса — это перевод его в стабильное и заранее известное состояние. С этого момента можно повторно выполнять какие-то неудачные операции. Если инициализация нестабильна, то наблюдение за процессом не стоит и гроша. Инициализированный начисто процесс должен быть стабильным что бы ни произошло. Таким образом, когда дочерние и сестринские процессы перезапустятся в будущем, они будут уверены в том, что остальная система, запущенная до них, в хорошем состоянии и исправна.

Если вы не обеспечите такое стабильное состояние, или если вы запускаете всю систему асинхронно, то такая структура даст вам мало преимуществ по сравнению с запуском `try ... catch` в цикле.

Процессы под наблюдением *обеспечивают гарантии* в фазе их инициализации, в отличие от *попыток сделать всё возможное*. Это означает, что когда вы пишете клиент для базы данных или сервис, вам не потребуется проверять наличие подключения в ходе фазы инициализации, если вы можете утверждать, что оно всегда будет доступно вне зависимости от ситуации.

Например вы могли бы принудительно выполнить подключение во время инициализации, если вы знаете, что база данных находится на одном с вами сервере и должна запуститься до старта вашей Erlang-системы. Тогда перезапуск сработал бы. В случае, если случится нечто непонятное и неожиданное, нарушающее эти гарантии, то ваш узел завершит работу аварийно, что вполне нам подходит: предварительное условие для запуска системы не было выполнено. Обязательная для старта системы тестовая проверка не прошла.

Если же, с другой стороны, ваша база данных находится на удалённом сервере, то следует быть готовыми к тому, что подключение завершится неудачей. Это повседневная реальность распределённых систем — их части постоянно отказывают[7]. В таком случае единственной гарантией, которую вы можете обеспечить в клиенстком процессе будет то, что ваш клиент сможет обработать запросы, но не подключиться к базе данных. Например, он мог бы возвращать `{error, not_connected}` на все вызовы во время разделения сети.

Повторное подключение к базе данных затем может произойти с любой удобной вам частотой или задержкой, не влияя на стабильность работы системы. Попытка также может быть выполненя в фазе инициализации, в качестве оптимизации, но процесс должен уметь повторно подключиться позже в случае отказа или отключения.

Если вы ожидаете, что внешний сервис откажет, не делайте его наличие обязательной гарантией для вашей системы. Мы здесь работаем с реальным миром, и отказ внешних зависимостей может произойти в любой момент.

[7] Или сетевая задержка увеличивается настолько, что становится трудно отличить ситуацию от полного отказа.

2.2.2 Побочные эффекты

Само собой, библиотеки и процессы, которые используют такой клиент, могут вылететь с ошибкой, если они не ожидают неожиданно оказаться без базы данных. Эта проблема совсем иная и лежит в другой плоскости, которая зависит от правил вашей бизнес-логики и того, что вы можете или не можете делать с клиентом, но проблема вполне решаемая. Например, представьте себе клиент некоторого сервиса, хранящий операционные метрики — код, который вызывает этого клиента, может игнорировать ошибки и они не окажут ощутимого влияния на работу системы в целом.

Разница в подходах к инициализации и наблюдению за процессами в том, что не наш клиент, а вызывающие его процессы принимают решение о том, насколько они готовы терпеть сбои и ошибки. Это очень важное различие, когда дело касается проектирования устойчивых к сбоям систем. Да, наблюдатели только и делают, что занимаются перезапусками, но суть в том, что перезапуски должны приводить к стабильному и известному заранее состоянию.

2.2.3 Пример: Инициализация без гарантии подключения

Следующий код пытается гарантировать наличие живого соединения при запуске процесса:

```
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}) ->
    %% попытаемся подключиться в цикле
    case connect(Opts) of
        {ok, New} -> {noreply, S#state{sock=New}};
         _ -> self() ! reconnect, {noreply, S}
    end;
```

Вместо этого попробуйте переписать его так:

```
init(Args) ->
    Opts = parse_args(Args),
    %% вы всё равно можете попытаться подключиться прямо здесь, чтобы по
    %% возможности улучшить доступность, но будьте готовы, что подключение
    %% не удастся
```

```
6       self() ! reconnect,
7       {ok, #state{sock=undefined, opts=Opts}}.
8
9   [...]
10
11  handle_info(reconnect, S = #state{sock=undefined, opts=Opts}) ->
12      %% попытаемся подключиться в цикле
13      case connect(Opts) of
14          {ok, New} -> {noreply, S#state{sock=New}};
15          _ -> self() ! reconnect, {noreply, S}
16      end;
```

Теперь вы можете позволить инициализацию с меньшим количеством гарантий: раньше требовалась гарантия *наличия подключения*, а сейчас требуется просто *наличие менеджера подключений*.

2.2.4 В двух словах

Производственные (*production*) системы, с которыми мне довелось работать, использовали и тот и другой подход.

Такие вещи, как файлы конфигурации, доступ к файловой системе (например, для потребностей ведения файлов журналов), локальные ресурсы, от которых могла бы зависеть система (открытие UDP-портов для ведения журналов), восстановление стабильного состояния из диска или по сети и так далее. Это те вещи, которые я помещаю в требования к наблюдателю и могу решить загрузить их синхронно независимо от того, сколько бы это ни потребовало времени (некоторые приложения иногда могут требовать более десяти минут для запуска, но это приемлемо, поскольку мы в этот момент перелопачиваем гигабайты, которые нам *нужны* для работы в качестве базового состояния, если мы не хотим отдавать клиентам некорректные или неполные ответы).

С другой стороны код, который зависит от удалённых баз данных и внешних сервисов, будет использовать запуск с частичными подключениями для ускорения запуска деревьев наблюдения, поскольку сбои ожидаются достаточно часто во время обычной работы, то и нет разницы в том, случится ли сбой при запуске или позже. Ваш подход к обработке этих ситуаций будет одинаковым, и для этих частей системы намного лучшим решением будет уменьшение гарантий.

2.2.5 Стратегии приложений

Вне зависимости от того, что произошло, ряд отказов не является смертельным приговором для узла. Как только система разделена на различные OTP-приложения, становится возможным выбирать, какие из приложений являются жизненно-важными для узла. Каждое из приложений в OTP может запускаться в трёх режимах: как временное (*temporary*), преходящее или

недолговечное (*transient*) или постоянное (*permanent*). Это делается либо вручную, выполняя команду `application:start(Имя, Тип)`, либо из файла конфигурации вашего релиза:

- Постоянное (`permanent`): если приложение завершит работу, то вся система останавливается, кроме тех случаев, когда приложение было остановлено вручную командой `application:stop/1`.
- Преходящее (`transient`): если приложение завершится по причине `normal`, то всё в порядке. Любая другая причина завершения работы останавливает всю систему.
- Временное (`temporary`): приложению разрешено останавливаться нормально и аварийно, по любой причине. О ситуации будет доложено в журналы или на экран, но ничего плохого больше не произойдёт.

Также возможно запустить приложение в качестве *вложенного приложения* (*included application*), что запустит его под наблюдателем OTP со своей собственной стратегией перезапуска.

2.3 Упражнения

Вопросы для закрепления материала

1. Запускаются ли деревья наблюдения в Erlang в глубину или в ширину? Синхронно или асинхронно?
2. Какие бывают три стратегии приложений? Что они делают?
3. Какие главные различия между структурами директорий приложения и релиза?
4. Когда следует использовать релиз?
5. Дайте два примера состояний, которые могут быть помещены в функцию инициализации процесса и два примера состояний, которые не следует туда помещать.

Тренировочные задания

Используйте код, который находится по адресу https://github.com/ferd/recon_demo:

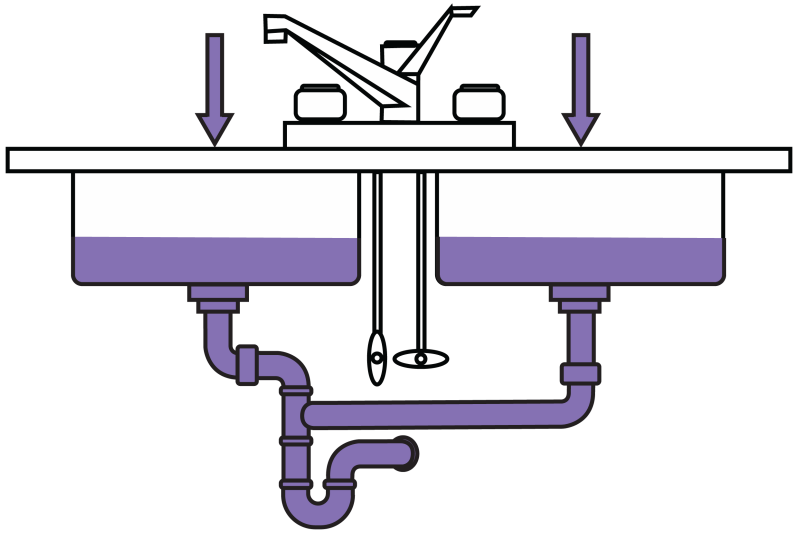
1. Извлеките главное приложение, которое хранится в релизе, и сделайте его независимым с возможностью включения в другие проекты.
2. Разместите приложение где-нибудь (Github, Bitbucket или ваш собственный сервер) и соберите релиз с этим приложением в качестве зависимости.
3. Рабочий процесс главного приложения (`council_member`) запускает сервер и подключается к нему в своей функции `init/1`. Можно ли сделать это подключение за пределами функции init? Есть ли выгода от этого в данном конкретном случае?

Chapter 3

В ожидании перегрузки

Без сомнения одной из самых частых причин сбоев, которые мне встречались в ситуациях реального мира, это переполнение памяти на узлах. Более того, обычно это связано с очередями сообщений, которые растут бесконтрольно[1]. Есть множество способов справиться с этой бедой, но выбор правильного решения потребует хорошего понимания системы, с которой вам довелось работать.

Если говорить упрощённо, то большинство проектов, с которыми мне пришлось работать, могут быть изображены в виде кухонной раковины. Пользовательские и входящие данные вытекают из крана. Erlang-система является раковиной и трубами, а то, куда идёт вывод результатов (неважно, что это, база данных или внешний сервис) — является канализацией.



[1] Как разобраться, что именно очередь сообщений оказалась проблемой, объясняется в Главе 6, а именно в Секции 6.2

Когда узел Erlang умирает по причине переполнения очереди, очень важно найти виновного. Вдруг налили слишком много воды? Справляется ли канализация? Не занизили ли вы размеры труб в проекте?

Определить, какая из очередей взлетела на воздух будет нетрудно. Эту информацию можно обнаружить в файле дампа после смерти узла. А вот поиск причины, по которой это произошло — более хитрая задача. Основываясь на роли процесса или изучении ситуации во время работы системы, можно вычислить причины — произошёл ли резкий всплеск числа сообщений, заблокированные процессы, которые не успевали выбирать сообщения, и так далее.

Самой трудной частью будет принятие решения о способе починки. Когда раковина заполняется отходами, мы обычно пытаемся увеличить размеры раковины (та часть программы, которая упала, первое, что мы видим). Затем мы догадываемся, что сток слишком мал и пытаемся улучшить этот момент. Затем обнаруживается, что трубы слишком узкие, и решаем эту проблему. Перегрузка оттесняется вглубь системы до тех пор, пока канализация не отказывается увеличивать принятые объёмы. В этот момент мы можем решить добавить больше раковин или увеличить число ванных комнат для решения проблемы с входными данными.

Затем наступает момент, когда на уровне ванной комнаты мы больше ничего не можем улучшить. Слишком много событий отправляется в журналы сообщений, или база данных с обязательным *требованием целостности* оказалась узким местом, или в нашей компании просто не хватает людей или ума, чтобы улучшить что-нибудь ещё.

Когда мы находим эту границу, можно считать, что *истинное узкое место* системы было обнаружено, и вся предыдущая оптимизация была полезной (и, наверное, дорогой), но всё было сделано зря.

Нам нужно быть умнее, и перенести наше внимание на уровень выше. Мы попытаемся заняться информацией, входящей в систему, и сделать её легче (с помощью сжатия, улучшения алгоритмов или представления данных, кэширования и так далее).

Бывают моменты, когда перегрузка слишком сильная, и нам приходится принимать трудный выбор между ограничением ввода в систему, уничтожением части входящих данных или смириться с фактом, что система понизит качество обслуживания вплоть до полной аварийной остановки. Эти механизмы можно объединить в две широкие категории: обратное давление (*back-pressure*) и сброс лишней нагрузки (*load-shedding*).

Мы рассмотрим их в этой главе вместе с общими событиями, которые могут привести к тому, что Erlang-системы взлетают на воздух.

3.1 Обычные источники перегрузок

Есть несколько известных причин того, что размеры очередей сообщений в Erlang-системах выходят из-под контроля, и рано или поздно многие люди с этими ситуациями сталкиваются независимо от их подхода к работе с системой. Обычно эти причины симптоматично проявляются в виде необходимости роста системы и требуют некоторой помощи для

облегчения масштабирования, или в виде неожиданного сбоя, который в результате каскадно распространяется и приводит к более тяжёлым последствиям, чем следовало бы.

3.1.1 Взрыв в error_logger

По странной иронии, процесс, отвечающий за журналирование ошибок одновременно является одним из самых хрупких. В установленном по умолчанию Erlang, процесс `error_logger`[2] не торопясь журналирует события на диск или по сети, и часто делает это медленнее, чем появляются новые ошибки.

Это в особенности верно для пользовательских сообщений (не для ошибок), и для аварийного завершения особо больших процессов. Для первого это является проблемой потому, что `error_logger` просто не готов к приёму произвольных сообщений, которые идут постоянным потоком. Он предназначен для исключительных ситуаций и не ожидает большого движения данных. А второе потому, что всё состояние процесса целиком копируется для записи в журнал (включая содержимое его почтового ящика). Всего лишь несколько таких сообщений и потребление памяти вспучивается пузырём, и даже если это не обрушивает весь узел по перерасходу памяти (*out of memory или OOM*), то однозначно замедлит процесс журналирования настолько, что дополнительные сообщения его добьют.

Лучшим решением в таком случае (на момент написания этого текста) будет замена библиотеки журналирования на `lager`.

В то время как `lager` не решит всех ваших проблем, он будет обрезать слишком длинные сообщения, которые пишутся в журнал, и пропускать некоторые ошибки OTP, количество которых превышает некоторый заданный порог, и автоматически будет переключаться между синхронным и асинхронным режимами для сообщений, отправленных пользователем, чтобы попытаться самостоятельно отрегулировать нагрузку.

Он не сможет справиться с некоторыми очень особенными случаями, когда, например, сообщения пользователя оказываются очень большими и все идут из одного процесса. Однако это намного более редкий случай, чем все остальные, и в таком случае у программиста чаще имеются рычаги влияния на ситуацию.

3.1.2 Замки (locks) и блокирующие операции

Операции над замками (*locks*) и блокирующие операции часто могут стать источником проблем, когда время исполнения операций неожиданно увеличивается, а процесс-владелец имеет постоянный приток новых заданий.

Одним из самых обычных примеров, которые я видел, можно назвать процесс, который блокируется во время приёма нового соединения (*accept*) или ждёт сообщения из TCP-сокетов. Во время блокирующих операций такого рода все другие сообщения накапливаются в очереди сообщений процесса.

[2] Описан здесь: http://www.erlang.org/doc/man/error_logger.html

Один особенно плохой пример мне встретился в диспетчере пула HTTP-соединений, который я написал для моего продолжения проекта `lhttpc`. Всё работало нормально почти для всех наших ситуаций, и мы даже одно время использовали время ожидания 10 мс, чтобы гарантировать, что операции не занимали слишком долго [3]. После нескольких недель идеальной работы, пул HTTP-клиента привёл к отказу в обслуживании, когда один из удалённых серверов остановился.

Причиной такого ухудшения было то, что когда удалённый сервер останавливался, то внезапно все операции подключения начинали занимать не менее 10 миллисекунд, как раз то самое максимальное время ожидания, после которого попытка подключения завершалась. Центральный процесс с потоком в 9.000 входящих сообщений в секунду тратил обычно не более 5 миллисекунд на обработку каждого сообщения, эффект оказался аналогичен двойной нагрузке в 18.000 сообщений в секунду и ситуация быстро вышла из-под контроля.

Решение, которые мы выбрали, передавало задачу подключения вызывающему процессу и усиливало ограничения так, как будто менеджер делал эту работу сам. Блокирующие операции таким образом оказывались распределены по всем процессам-пользователям библиотеки и требовалось совсем немного усилий от менеджера, который смог принимать больше запросов.

Когда в вашей программе находится *любая* центральная точка обработки чего-нибудь, постарайтесь по возможности вынести оттуда все долгие задачи. Обработка предсказуемых ситуаций перегрузки[4] с помощью добавления новых процессов — которые либо обрабатывают блокирующие операции или действуют в качестве буфера в то время, как главный процесс занят — это хорошая идея.

Увеличенное число процессов, обслуживающих не совсем параллельные операции будет сложнее в управлении, так что убедитесь, что вам подходит такое решение перед тем, как начинать строить защиту такого рода.

Другим вариантом может быть изменение задачи блокирования и превращение её в асинхронную задачу. Если ваша работа позволяет такое решение, запустите долгую задачу в новом процессе и храните токен, который уникально её идентифицирует, а также ссылку на того, для кого выполняется эта работа. Когда ресурс становится доступен (работа готова) — отправляете сообщение обратно серверу и передаёте в нем этот токен. Сервер в какой-то момент получит сообщение, сопоставит токен и того, кто запрашивал выполнение данной работы, и ответит ему, не блокируясь при этом на ожидании чего-нибудь[5].

Такое решение может оказаться более непрозрачным, чем использование множества процессов, и может быстро превратиться в ад обратных вызовов (*callback hell*), но при этом будет более эффективным с точки зрения используемых ресурсов.

---

[3] 10 миллисекунд — это очень короткий интервал, но вполне подходил нам, потому что мы использовали сервера в одной сети для ставок в реальном времени.

[4] Там, где на реальной системе, как вы уверены, могут появиться большие нагрузки.

[5] Приложение `redo` является примером библиотеки, которая так делает в модуле redo_block. Модуль [плохо документированный] превращает потоковое подключение в блокирующее, но при этом сохраняет аспекты потоковой работы с точки зрения вызывающего клиента — это позволяет клиенту узнать, что во время срабатывания таймаута завершился неудачей только один вызов, а не все вызовы подряд, и при этом сервер не прекращает принимать новые запросы.

3.1.3 Неожиданные сообщения

Сообщения, о которых вы не знали, обычно являются редкостью при использовании OTP-приложений. Поскольку поведения OTP практически всегда требуют от вас обрабатывать все входящие сообщения с помощью функции `handle_info/2`, то неожиданные сообщения не будут накапливаться в ощутимом количестве.

Однако все виды OTP-совместимых систем в конце концов обзаводятся процессами, которые не реализуют одно из стандартных поведений, или процессы, время от времени выполняющие нестандартный код, перехватывающий обычную обработку сообщений. Если вам повезло, то инструменты мониторинга[6] покажут постоянный рост расхода памяти и поиск длинных очередей[7] позволит найти процессы, которые в этом виноваты. Затем вы можете исправить проблему, дописав код, который обработает незнакомые сообщения по мере их прихода.

3.2 Ограничение ввода

Ограничение входящих данных — это простейший способ управления ростом длины очередей сообщений в Erlang-системах. Этот подход — простейший, поскольку фактически означает, что вы замедляете работу ваших пользователей (применяя к ним обратное давление, *back-pressure*), что немедленно исправляет проблему, не требуя дополнительных оптимизаций. С другой стороны вы можете здорово усложнить или испортить жизнь вашим пользователям.

Самый распространённый способ ограничить входящие данные — это сделать вызовы к процессу, чья очередь сообщений бесконтрольно растёт, синхронными. Требуя ответ перед тем, как перейти к следующему запросу, вы в общем случае гарантируете, что непосредственный источник вашей проблемы замедлится.

Сложность этого подхода в том, что узкое горлышко, которое привело к росту очереди, обычно находится не на краю системы, а глубоко внутри неё, и вы обнаружите это когда оптимизируете практически всё, что оказалось перед ним. Такие бутылочные горлышки часто оказываются операциями с базой данных, с диском или каким-то удалённым сетевым сервисом.

Это означает, что когда вы добавляете синхронное поведение глубоко в системе, то вам, вероятно, придётся обрабатывать случаи обратного давления на каждом уровне до краёв системы, и там вы сможете сообщить пользователю: ”Эй там, помедленнее!”. Разработчики, которые видят такой шаблон поведения, часто пытаются внести ограничения на API для каждого пользователя[8] на точках входа в систему. Это разумный подход, особенно поскольку он в состоянии гарантировать базовое качество сервиса (*QoS*) для системы и позволяет распределить ресурсы как можно более честно (или нечестно), по желанию.

---

[6] Смотрите секцию 5.1

[7] Смотрите подсекцию 5.2.1

[8] Есть компромисс между замедлением всех запросов в одинаковой мере, или использования ограничений скорости по каждому пользователю, и оба варианта вполне хороши. Ограничение по пользователю означает, что вам всё равно придётся увеличить ёмкость или понизить лимиты, когда большее количество пользователей начнут бомбить запросами вашу систему, тогда как синхронная система, блокирующая всех подряд, подстроится сама под возможную максимальную нагрузку, но вероятно не совсем честно по отношению к отдельным пользователям.

3.2.1 Каким делать ограничение времени ожидания

Что может вызвать сложности при оказании обратного давления посредством синхронных вызовов — так это необходимость определить, сколько же занимает типичная операция, вернее когда система должна прекращать ожидание и генерировать ошибку.

Лучший способ выразить существующую проблему — первый таймер будет запущен на границе системы, но все важные операции будут происходить глубоко в системе. Это означает, что таймер на границе системы должен быть длиннее, чем внутренние таймеры, если только вас не устраивает отказ на краю системы, когда часть операций внутри неё оказалась успешной.

Лёгким способом выбраться из этой проблемы может оказаться бесконечное время ожидания. Pat Helland в его работе[9] даёт интересный ответ на этот вопрос:

> Некоторые разработчики приложений могут настаивать на том, что бесконечное время ожидания очень даже приемлемо. Я обычно предлагаю установить время ожидания, равное 30 годам. Это вызывает ответы в духе «прекрати глупить и предложи что-нибудь поразумнее». *Почему 30 лет ожидания — это глупость, а бесконечное ожидание — разумно?* Я ещё не видел приложения, например, чата, которое готово ждать завершения сетевых операций в течение неограниченного времени.

Эта проблема в значительной мере зависит от каждого конкретного случая. Во многих случаях было бы более практично использовать иной механизм для контроля потока данных[10].

3.2.2 Спросить разрешения

В некотором роде более простой подход к оказанию обратного давления — это определить те ресурсы, которые мы блокируем, их нельзя никак ускорить и являющиеся критичными для вашего бизнеса и пользователей. Закрывайте на замок (*lock*) эти ресурсы с помощью отдельного модуля или некоей процедуры, следуя которой вызывающий процесс должен запросить право выполнить запрос и тогда имеет право начать использовать ресурсы. Есть множество переменных, по которым можно принять решение об разрешении или отказе: свободная память, процессор, общая нагрузка, некоторый счётчик вызовов, параллельность, время прошлых ответов, комбинация предыдущих признаков и так далее.

Приложение *SafetyValve* (защитный клапан)[11] является общесистемной библиотекой, которую можно использовать, когда вам понадобится обратное давление.

Для более специфичных случаев использования, связанных со сбоями служб или системы, есть ряд приложений, помогающих ограничить доступ к частям системы, которые

[9] Idempotence is Not a Medical Condition (Идемпотентность — не болезнь), 14 апреля 2012

[10] В Erlang использование значения `infinity` не создаёт таймер, таким образом избегая некоторых расходов ресурсов. Если вы пойдёте по этому пути, то не забудьте вставить чётко определённое ограничение времени ожидания где-нибудь в цепочке ваших вызовов.

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

внезапно отказали, не приводя к общей аварии. Например, такие: `breaky`[12], `fuse`[13], или `circuit_breaker`[14] под авторством компании Klarna.

В других случаях можно написать локальные решения прямо на месте, используя процессы, ETS или любой другой доступный инструмент. Важным момент здесь является то, что край системы (или подсистемы) может заблокировать вызов или запросить право обработки данных, но именно критический участок кода имеет право решать, давать это право или нет.

Преимущество обработки этим способом в том, что можно избежать всех тонкостей и сложностей, которые относятся к таймерам, и сделать все слои абстракции синхронными. Вместо этого вы размещаете охранную логику в узком месте программы и в контрольной точке или на краю системы, а всё между ними выражается в очень простой форме.

3.2.3 Что видят пользователи

Сложным моментом при работе с обратным давлением является уведомление о нём. Если обратное давление оказывается посредством синхронных вызовов, то единственным способом узнать о нём при перегрузке это то, что система становится медленнее и менее отзывчивой. К сожалению, это также может быть признаком плохого аппаратного обеспечения, плохой сети, другой, не связанной с известной уже нам перегрузкой или, возможно признаком медленного клиента.

Попытки выяснить что в системе оказывает обратное давление посредством измерения времени её ответа примерно равняется попытке выполнить диагностику больного, наблюдая его повышенную температуру и лихорадку. Видно, что больному нехорошо, но почему — неясно.

Запрос разрешения в качестве механизма обычно позволит вам определить собственный интерфейс таким образом, что вы сможете явно сообщить о происходящем: система в целом перегружена либо вы упёрлись в некоторый лимит частоты выполнения операций и должны подстроиться под него.

При проектировании системы следует сделать выбор: будут ли пользователи иметь ограничения на каждую учётную запись или вы используете общие лимиты, глобальные для вашей системы?

Ограничения, глобальные для системы или узла, обычно делаются достаточно легко, но имеют недостаток в том, что они не совсем честные. Один пользователь, выполняющий 90% всех запросов в вашей системе может сделать платформу недоступной для всех остальных.

С другой стороны, ограничения на каждую учётную запись, стремятся быть честными и позволяют хитроумные схемы, например особые повышенные лимиты для платных пользователей. Это очень удобно, но имеется и недостаток — при росте числа пользователей в вашей системе, эффективное глобальное системное ограничение тоже начинает расти. Возьмём 100 пользователей, каждый из которых имеет право делать 100 запросов в минуту, это даст нам общее ограничение в 10.000 запросов в минуту. Но стоит добавить ещё 20 пользователей с таким

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

же ограничением и внезапно ёмкость системы оказывается превышена и вы регулярно падаете при каждом всплеске числа работающих одновременно пользователей.

Безопасная дистанция от предела возможностей системы, которую вы закладывали при оригинальном проектировании, медленно размывается при росте числа пользователей. Важно определить компромиссы, на которые готов пойти ваш бизнес с этой точки зрения, поскольку пользователи не очень-то ценят, когда их разрешённые ограничения начинают уменьшатся, причём раздражаются даже больше, чем если бы система просто иногда падала.

3.3 Уничтожение лишних входных данных

Когда клиентам за пределами вашей Erlang-системы замедляться уже нет возможности, как и нет возможности отмасштабировать саму систему, вам следует начать выбрасывать часть лишних входных данных либо завершать работу аварийно (что автоматически уничтожит и данные, передаваемые в данный момент, если отправитель не умеет посылать их повторно).

Это грустная реальность, с которой никто на самом деле не хочет связываться. Программисты, инженеры и учёные в области компьютерных наук обучены удалять лишние данные и оставлять всё, что может оказаться полезным. Успех приходит путём оптимизации, а не когда мы сдаёмся и перестаём бороться.

Однако имеется вполне вероятная ситуация, когда данные входят быстрее, чем успевают выходить, даже если Erlang-система сама по себе способна всё обработать с достаточной скоростью. В некоторых случаях блокировать работу может быть компонент *в конце* системы.

Если у вас нет возможности ограничить количество входящих данных, вам может быть придётся выбрасывать часть сообщений, чтобы избежать плачевных последствий перегрузки.

3.3.1 Выбрасываем случайно

Самый лёгкий способ реализовать такой подход — случайно выбрасывать заданный процент сообщений, и благодаря своей простоте может также оказаться самым надёжным способом.

Следует определить некоторый уровень порогового значения между 0.0 и 1.0, и выбирать случайное число в этом диапазоне:

```
-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.
```

Если вы стремитесь сохранить 95% сообщений, то авторизация клиента может быть записана, как `case drop:random(0.95) of true -> send(); false -> drop() end`, или более коротко `drop:random(0.95) andalso send()`, если вам не нужно делать никаких дополнительных действий, когда вы уничтожаете сообщение.

Функция `maybe_seed()` проверит, что имеется подходящее начальное значение в словаре процесса и использует его вместо не очень удачного стандартного, но только если оно не было определено ранее, чтобы избежать слишком частого вызова функции `now()` (монотонно растущей функции времени, которая использует глобальный замок).

Однако в этом методе есть один рискованный момент: случайное отбрасывание данных должно в идеале происходить на уровне источника, а не очереди (получателя). Лучшим способом избежать перегрузки очереди будет, в первую очередь, отказ от отправки лишних данных. Поскольку в Erlang нет ограниченных почтовых ящиков, удаление в получающем процессе гарантирует лишь то, что он будет крутиться в цикле, пытаясь избавиться от лишних сообщений, и мешать планировщикам делать по-настоящему полезную работу.

С другой стороны удаление на уровне источника данных гарантированно распределит работу поровну между всеми процессами.

Это может открыть возможности для интересных оптимизаций, где рабочий процесс или заданный процесс-монитор[15] использует значения в ETS-таблице или `application:set_env/3` чтобы динамически увеличить или уменьшить количество выбрасываемых при перегрузке сообщений, и конфигурационные данные могут быть получены любым процессом достаточно эффективно с помощью `application:get_env/2`.

Подобную технику можно также использовать для реализации особых пропорций выбрасывания лишних данных для разных приоритетов сообщений вместо того, чтобы пытаться рассортировать всё на уровне потребителя.

3.3.2 Буфера с очередью

Буфера с очередью являются хорошей альтернативой, когда вам нужно больше контроля над тем, от каких сообщений вам нужно избавиться с помощью случайного выбора, особенно когда вы ожидаете перегрузки по причине коротких всплесков нагрузки, а не постоянного потока данных, который надо как-нибудь уменьшить.

Даже несмотря на то, что обычный почтовый ящик процесса имеет вид очереди, обычно вы предпочтёте выбрать из него *все* сообщения так быстро, как это возможно. Буфер очереди потребует два процесса для безопасности.

[15] Любой процесс, имеющий задачу проверить загрузку других заданных процессов, используя некоторые эвристики, такие как `process_info(Pid, message_queue_len)` может играть роль монитора.

- Обычный процесс, с которым вы бы работали (вероятно это будет `gen_server`);
- Новый процесс, который ничего не будет делать, а лишь накапливать сообщения. Сообщения, идущие снаружи должны входить в этот процесс.

Чтобы эта схема начала работать, буферный процесс должен забирать все сообщения, какие он может найти, из своего почтового ящика и складывать их в структуру данных типа «очередь»[16], которой он сам владеет и управляет. Когда сервер готов выполнить новую работу, он запрашивает у буферного процесса заданное количество сообщений, над которыми он готов начать работу. Буферный процесс выбирает сообщения из своей очереди, передаёт их рабочему серверу, а сам возвращается к задаче накопления входящих данных.

Когда очередь вырастает до определённого размера[17] и вам приходит очередное сообщение, вы можете извлечь самое старое и поместить в очередь новое, продолжая выбрасывать самые старые элементы по ходу работы[18].

Это будет удерживать количество полученных сообщений в более-менее стабильном диапазоне и обеспечит хорошую устойчивость к перегрузке, по свойствам чем-то похожую на функциональную версию кольцевого буфера.

Библиотека *PO Box*[19] реализует как раз такой буфер с очередью.

3.3.3 Стековые буфера

Стековые буфера идеально подходят, когда вам требуется столько же возможностей по контролю, как в буферах очередей, но дополнительно имеется важное требование быстрой скорости ответа.

Для того, чтобы использовать стек в качестве такого буфера, вам понадобятся два процесса, очень похоже на то, как мы делали с очередями, но вместо очереди будет использована структура данных типа «список»[20].

Причина, по которой стековый буфер особенно хорошо подходит для уменьшения времени отклика, относится к проблеме раздутия буфера[21]. Если вы начинаете опаздывать на несколько миллисекунд с обработкой нескольких сообщений, то все оставшиеся сообщения в буфере тоже замедляются и приобретают дополнительные миллисекунды. Через время все сообщения устаревают и весь буфер можно выбрасывать по причине лежащих слишком долго данных.

---

[16] Стандартный модуль `queue` в Erlang обеспечивает чисто-функциональную структуру данных типа «очередь», которая как раз подошла бы для такого буфера.

[17] Чтобы вычислить размер очереди предпочтительно использовать некоторый счётчик, который будет увеличиваться или уменьшаться при каждом пришедшем или отданном сообщении вместо перебора всех элементов очереди. Это занимает немного больше памяти, но распределит нагрузку более равномерно, поможет улучшить предсказуемость и избежать внезапного нагромождения в почтовом ящике процесса-буфера

[18] Как вариант, можно создать очередь, которая выбрасывает самые новые элементы и откладывает самые старые, если предыдущие данные вам более важны.

[19] Доступна по адресу: https://github.com/ferd/pobox, использовалась в течение долгого времени на производственных серверах в масштабных проектах компании Heroku и считается зрелым продуктом.

[20] Списки в Erlang *являются* стеками, с точки зрения нашей задачи. Они обеспечивают функции помещения и извлечения из стека со сложностью $O(1)$ и в целом очень быстры.

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

С другой стороны стек сделает так, что лишь ограниченное число элементов будут продолжать ожидать, в то время, как более новые успешно попадают на обработку сервером в кратчайшие сроки.

Когда вы видите рост стека более некоторого размера или замечаете, что элемент в стеке слишком стар для ваших требований качества сервиса (*QoS*), можно выбросить остаток стека и продолжать далее с этого места. *PO Box* также предлагает такую реализацию буфера.

Большим недостатком стековых буферов является то, что сообщения не обязательно будут обработаны в порядке их отправки — это прекрасно подходит для не связанных между собой задач, но может испортить вам весь день, если вы ожидали прихода события в том же порядке, в каком вы их отправляли.

3.3.4 Буфера, которые считают время

Если вам нужно реагировать на старые события *до того*, как они слишком устареют, то всё становится немного сложнее, поскольку вы не можете выяснить этот факт, не перебирая содержимое стека каждый раз, а выбрасывать элементы с конца было бы неэффективно. Интересным подходом здесь является вариант с вёдрами (*buckets*), в котором используются сразу несколько стеков, каждый из которых содержит элементы в заданном диапазоне времени. Когда весь такой стек устаревает по меркам ваших требований к качеству сервиса, то он очищается, но не весь буфер целиком.

Это может звучать не совсем логично: ухудшать время ответа на часть запросов, чтобы большинство оставшихся стало лучше — у вашего сервиса будут хорошие средние показатели, но плохие 99-процентили — но это случится в то время, когда вы бы всё равно начали выбрасывать часть сообщений, и это предпочтительнее в тех случаях, когда вам требуется хорошее короткое время ответа.

3.3.5 Жизнь в постоянной перегрузке

В случае, когда ситуация перегрузки становится постоянным явлением, требуется новое решение. В то время, как очереди и буферы прекрасно помогут, если перегрузка случается время от времени (даже если она при этом продолжается относительно долгое время), они хорошо работают когда вы ожидаете, что скорость входящих данных упадёт и позволит вам раскидать накопившиеся задачи.

Чаще всего вы получите проблемы, когда попытаетесь отправлять настолько много сообщений, что принимающий процесс не сможет их все принять без создания ситуации перегрузки. В данном случае хорошо работают два следующих подхода:

- Запустить множество процессов, которые будут выполнять роль буферов, и балансировать нагрузку с их помощью (горизонтальное масштабирование);
- Использовать ETS-таблицы в качестве замков и счётчиков (уменьшить поток входящих данных).

ETS-таблицы способны обработать намного больше запросов в секунду, чем Erlang-процесс, но поддерживаемые ими операции намного более простые. Чтение одного элемента, атомарное увеличение или уменьшение счётчика — это самые сложные операции, на которые можно рассчитывать в общем случае.

Для обоих подходов вам потребуется использование ETS-таблиц.

В общих словах, первый подход может хорошо сработать и с обычным реестром процессов: вы берёте `N` процессов, чтобы разделить между ними нагрузку, даёте им известное заранее имя, и выбираете, кто получит очередное сообщение. Поскольку вы заранее знаете, что будете работать в состоянии перегрузки, то случайный выбор с помощью равномерного распределения будет достаточно надёжен: не требуется никакой связи между состояниями процессов, работа будет разделена примерно поровну, и эта схема довольно снисходительно относится к возникновению ошибок.

Однако на практике мы хотим избежать динамического создания новых атомов, поэтому я предпочитаю регистрировать рабочие процессы в ETS-таблице с включённым флагом `read_concurrency`. Это несколько более трудоёмкий подход, но он даёт больше гибкости, когда дело касается изменения числа рабочих процессов в будущем.

Подобный этом подход был использован в библиотеке `lhttpc`[22], которую мы упоминали ранее, чтобы разделить балансировщики нагрузки по доменам.

Для второго подхода с использованием замков и счётчиков используется та же основная структура (выберите один вариант из множества, и отправьте ему сообщение), но перед тем, как сообщение будет отправлено, вам следует атомарно обновить ETS-счётчик[23]. Задаётся некоторый известный лимит, общий для всех клиентов (либо посредством их наблюдателя, либо файла конфигурации или значения в ETS-таблице) и на каждый запрос к процессу мы обязаны проверить, проходит ли он по этому лимиту.

Этот подход использовался в модуле `dispcount`[24], чтобы избежать очередей сообщений и гарантировать быстрые ответы на любое сообщение, которое нет возможности обработать, так чтобы не приходилось ждать, если ваш запрос всё равно закончится отказом. Затем пользователь библиотеки решает, что с этим делать — сразу оставлять эту затею и экономить время, или продолжать пробовать повторно и пытаться связаться с другими рабочими процессами.

3.3.6 Как избавляться от данных

Большая часть решений, описанных здесь, работают основываясь на количестве сообщений, но также имеется возможность попробовать делать это на основании других метрик: размера сообщений, ожидаемой сложности выполняемой работы, если её можно заранее предсказать. При использовании очереди или стекового буфера вместо того, чтобы считать число сообщений, можно посчитать их размер и назначить им заданный уровень нагрузки в качестве ограничения.

[22] Модуль lhttpc_lb в этой библиотеке реализует такой подход.
[23] Для этого имеется функция `ets:update_counter/3`.
[24] https://github.com/ferd/dispcount

Я обнаружил, что на практике, отбрасывание данных без учёта специфики самих данных работает вполне хорошо, но каждое приложение имеет свой набор компромиссов, которые могут быть приемлемы или нет[25].

Существуют также случаи, в которых данные отправляются вам в режиме «бросил и забыл» — вся система является частью асинхронного конвейера — и налаживание обратной связи с конечным пользователем, чтобы сообщить о неудаче или утрате некоторых из запросов, оказывается трудным. Если вы можете зарезервировать особый тип сообщений, которое бы собирало статистику об выброшенных сообщениях и сообщало бы пользователю «`N` сообщений были выброшены по причине `X`», что в свою очередь может сделать компромисс для пользователя намного более приемлемым. Это тот выбор, который мы сделали в logplex, системе роутинга журналов в компании Heroku, которая может отдавать ошибки с кодом L10, сообщающие пользователю, что часть системы сейчас неспособна справиться с объёмом данных.

В конце концов, что является приемлемым для того, чтобы справиться с перегрузкой, зависит от людей, использующих систему. Часто легче немного изменить требования, чем разрабатывать новую технологию, но иногда это оказывается неизбежным.

3.4 Упражнения

Вопросы для закрепления материала

1. Назовите самые частые источники перегрузки в Erlang-системах?
2. Назовите два основных класса стратегий борьбы с перегрузками?
3. Как долго выполняющиеся операции можно сделать безопаснее?
4. Чем следует руководствоваться в выборе ограничений времени ожидания при синхронной отправке задач процессам?
5. Какая существует альтернатива ограничению времени ожидания?
6. В каких случаях вы предпочтёте буфер с очередью стековому буферу?

Открытые вопросы

1. Что такое *настоящее узкое место*? Как его найти?

[25] Старые труды, как например Hints for Computer System Designs (Советы по проектированию компьютерных систем) под авторством Butler W. Lampson, рекомендуют выбрасывать сообщения: ”Сбрасывайте нагрузку для того, чтобы контролировать спрос, вместо того, чтобы позволить системе оказаться перегруженной.” Таке упоминается следующее: ”Не следует ожидать хорошей работы от системы, если спрос на любой ресурс превышает $2/3$ ёмкости, если только нагрузка не определена очёнь чётко.”, добавляя при этом, что ”Умный подход работает только в тех системах, где нагрузка очень хорошо известна.”

2. В приложении, которое вызывает API третьей стороны, время ответов сильно различается в зависимости от того, насколько хорошо себя чувствуют другие сервера. Как можно спроектировать систему так, чтобы предотвратить блокировку других параллельных вызовов некоторого сервиса особо медленными запросами?

3. Что, вероятнее всего, произойдёт с новыми запросами к перегруженному сервису, время ответа которого очень важно для нас, если данные накопились в стековом буфере? А что случится со старыми запросами?

4. Объясните, как можно превратить механизм сбрасывания лишней нагрузки в такой, который может оказывать обратное давление на потребителя.

5. Объясните, как превратить механизм обратного давления в такой, который сбрасывает лишнюю нагрузку.

6. Каковы риски для пользователя, если мы отбросим или заблокируем запрос? Как можно предотвратить дублирование или потерю сообщений?

7. Что может случиться с вашим API, если при проектировании вы забудете добавить обработку перегрузки, и внезапно вам понадобится оказывать обратное давление или сбрасывать лишнюю нагрузку?

Part II

Диагностика приложений

Chapter 4

Подключение к удалённым узлам

Взаимодействие с работающей серверной программой традиционно делается двумя способами. Один способ — подключиться к интерактивному интерпретатору, который остаётся доступным с помощью приложений мультиплексирования консолей таких, как `screen` или `tmux`, и выполняет ваше приложение в фоне, но позволяет подключить к нему экран и вводить команды. Другой способ — добавить функции управления или подробные файлы конфигурации, которые можно перезагружать на ходу.

Подход с интерактивной сессией обычно хорошо подходит для программ, которые работают в строгом цикле ввод команды-исполнение-печать (*REPL, Read-Eval-Print-Loop*). С другой стороны запрограммированные команды управления и конфигурирования требуют внимательного планирования тех задач, которые по вашему мнению могут понадобиться, и затем надеяться, что этого будет достаточно. Практически все системы могут попробовать этот подход, так что я просто пропущу его поскольку мы больше заинтересованы в тех случаях, когда всё уже плохо и нет заготовленной функции, чтобы это исправить.

Erlang использует нечто ближе к прямому общению с машиной, чем цикл ввода-исполнения (*REPL*). Проще говоря, обычная виртуальная машина Erlang вообще не требует наличия никакого цикла ввода-исполнения, она будет с радостью исполнять свой байтовый код и ничего кроме этого делать ей не нужно. Однако поскольку она работает с параллельностью и одновременной обработкой данных, и имеет хорошую поддержку распределённого исполнения, то есть возможность запускать встроенные интерактивные интерпретаторы (*REPL*), которые являются обычными процессами на Erlang.

Это означает, что в отличие от обычной единственной сессии с подключенным терминалом пользователя, вполне возможно иметь столько интерактивных интерпретаторов, работающих с одной виртуальной машиной, сколько вам необходимо[1].

Самые распространённые варианты использования будут зависеть от наличия секретного

[1] Подробнее об этих механизмах здесь: http://ferd.ca/repl-a-bit-more-and-less-than-that.html

куки (*cookie*) на обоих узлах, которые вы желаете соединить[2], но имеются способы сделать это даже без куки. Большинство вариантов будет требовать от вас использования именованных узлов, и все из них потребуют *заранее* принять меры, чтобы у вас была возможность связаться с узлом.

4.1 Режим управления задачами

Режим управленя задачами (или сокращённо *JCL*) это небольшое текстовое меню, которое выпадает при нажатии в интерактивном интерпретаторе комбинации клавиш ^G. В этом меню есть команда подключения к консоли интерпретатора на удалённом узле:

```
(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>
```

Когда такое происходит, локальный интерпретатор выполняет работу с клавиатурой, редактирование команды и управление задачами на локальном узле, а выполнение самой команды происходит на удалённом, к которому вы подключились. Весь вывод команды перенаправляется обратно на локальный экран на нашем узле.

Чтобы завершить работу в консоли, вернитесь обратно в режим JCL нажатием ^G. Управление задачами, как я писал выше, выполняется локально и можно безопасно выходить нажатием ^G q:

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

[2] Подробнее о куки в книге: http://learnyousomeerlang.com/distribunomicon#cookies или в документации http://www.erlang.org/doc/reference_manual/distributed.html#id83619

Вы можете решить запустить первую консоль в скрытом режиме (с параметром `-hidden`), чтобы избежать автоматического подключения вашего узла ко всем остальным узлам в кластере.

4.2 Remsh

Существует механизм совершенно подобный тому, что доступен через режим JCL, хотя вызывается он несколько иначе. Все шаги, связанные с режимом JCL, можно пропустить если запустить интерпретатор следующим образом (для длинных имён):

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

И таким образом для коротких имён:

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

Все другие аргументы Erlang (такие, как `-hidden` и `-setcookie $COOKIE`) тоже разрешены. Внутренние механизмы действуют точно так же, как и в режиме JCL, но интерпретатор сразу запускается удалённо вместо локального (JCL работает локально). Клавиша `^G` остаётся самым безопасным способом, чтобы покинуть удалённую консоль интерпретатора.

4.3 SSH-демон

Erlang/OTP содержит в стандартной поставке реализацию протокола SSH, которая может действовать как в качестве сервера, так и клиента. Часть его является демонстрационным приложением, которое открывает удалённый терминал, работающий с Erlang.

Обычно требуется, чтобы ключи находились в соответствующей директории, чтобы получить удалённый доступ к SSH. Но для нашего теста будет вполне достаточно выполнить следующие команды:

```
$ 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>}
```

Здесь я задал всего лишь несколько параметров, в частности `system_dir`, которая указывает, где находятся файлы ключей, и `user_dir`, указывающий директорию с файлами конфигурации SSH. Есть множество других параметров, позволяющих задавать пароли, настраивать управление публичными ключами и так далее[3].

Для подключения к демону подходит любой SSH-клиент:

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

Таким образом вы сможете взаимодействовать с установленной Erlang-системой без необходимости устанавливать её на вашей локальной машине. Простое отключение SSH-сессии достаточно для того, чтобы покинуть консоль. *Не выполняйте* функции остановки узла, такие как `q()` или `init:stop()`, которые приведут к остановке удалённого сервера[4].

Если у вас возникли проблемы с подключением, можно добавить к `ssh` опцию `-oLogLevel=DEBUG` для получения подробного пошагового вывода и помощи в поиске проблемы.

4.4 Именованные каналы

Малоизвестный способ подключения к узлу Erlang, который не требует никаких особых параметров для распределённого режима — с помощью именованных каналов (*named pipes*). Чтобы это сделать, надо запустить Erlang командой `run_erl`, которая обернёт его ввод и вывод в именованный канал[5]:

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

Первый аргумент – это имя файла, который будет именованным каналом. Второе — директория, куда будут писаться файлы журналов[6].

Для подключения к такому узлу используйте программу `to_erl`:

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

1>
```

[3] Полная инструкция по всем параметрам доступна здесь: http://www.erlang.org/doc/man/ssh.html#daemon-3.

[4] Этот совет действителен для всех способов взаимодействия с удалённым узлом Erlang.

[5] На самом деле выполняется команда `"erl"`. Дополнительные аргументы можно добавить в конце команды. Например, `"erl +K true"` включит режим kernel poll.

[6] Использование этого метода будет вызывать fsync для каждого фрагмента вывода, что может дать ощутимый удар по производительности, если ваша программа выводит много на экран (стандартный вывод).

И вот, консоль подключена. Закрытие стандартного ввода-вывода (нажатием клавиши ^D) отключится от вашей консоли, оставляя программу запущенной.

4.5 Упражнения

Вопросы для закрепления материала

1. Какие есть 4 способа подключения к удалённому узлу?
2. Можно ли подключиться к узлу, если ему не было дано имя?
3. Какая команда переходит в режим Job Control Mode (JCL)?
4. Какой(какие) методы подключения к удалённой консоли не очень подходят, если система выводит много данных на стандартный вывод?
5. Какие из удалённых подключений не следует отключать нажатием клавиш ^G?
6. Какие команды нельзя использовать для отключения сессии?
7. Все ли перечисленные методы поддерживают множество одновременно подключенных к одной Erlang-системе пользователей без проблем?

Chapter 5

Метрики времени выполнения

Одной из наиболее хорошо продаваемых характеристик виртуальной машины Erlang для производственного использования является то, как она прозрачна для всех способов диагностики, отладки, профилирования и анализа на ходу во время выполнения.

Преимуществом наличия таких метрик, доступных программно, является то, что построение как инструментов так и скриптов для автоматизации, которые на них основаны оказывается вполне простым[1]. Затем, когда понадобится, возможно избавиться от инструментов и обратиться прямо к виртуальной машине и запросить нужную информацию.

Практическим подходом к росту системы и поддержанию её в хорошем здоровье на производственном (*production*) сервере — это убедиться, что систему можно наблюдать со всех углов, как в большом масштабе, так и все мельчайшие её детали. Нет общего рецепта, который сразу скажет вам является ли тот или иной показатель нормальным или аварийным. Вам следует сохранять множество данных и время от времени смотреть на них, чтобы сформировать своё мнение о том, как система выглядит в обычных условиях. В тот день, когда что-то пойдёт не так, у вас будет опыт всех этих точек зрения, к которым вы привыкли, и вам будет легче обнаружить, что вышло из-под контроля и требует ремонта.

Для этой главы (и многих глав после этой), многие показанные концепции или возможности доступны прямо из вашего кода с помощью стандартных библиотек, которые являются частью каждого дистрибутива OTP.

Однако эти возможности не сложены в одном месте, и могут очень облегчить задачу простреливания себе ноги на производственной системе. Также они стремятся быть скорее строительными блоками, чем готовыми к использованию инструментами.

Таким образом, чтобы сделать текст книги легче для чтения и удобнее, я сгруппировал общие операции в библиотеке `recon`[2] и они в целом безопасны для использования на производственной системе.

[1] Гораздо сложнее гарантировать, что автоматизированные процессы не переборщили с исправительными мерами.
[2] http://ferd.github.io/recon/

5.1 Вид сверху

Для осмотра состояния виртуальной машины в большом масштабе полезно собирать статистику и метрики общие для виртуальной машины, независимо от того, что на ней выполняется. Более того, следует стремиться к решению, которое позволяет хранить долгосрочную историю каждой метрики — некоторые проблемы проявляются в виде долгого накопления в течение недель, которое нельзя увидеть за небольшой промежуток времени.

Хорошие примеры проблем, которые можно обнаружить глядя на историю за долгое время — это утечки памяти или процессов, обычные или необычные всплески активности относительно времени дня или недели, что часто требует наличие многомесячных архивов данных для укрепления уверенности в догадках.

Для этих случаев полезно использовать существующие Erlang-приложения для сбора метрик. Общепринятыми вариантами обычно являются следующие:

- `folsom`[3] сохраняет метрики в памяти внутри виртуальной машины, как глобальные так и для конкретных приложений.
- `vmstats`[4] и `statsderl`[5], отправляет метрики узла на сервис graphite посредством `statsd`[6].
- `exometer`[7], хорошо выглядящая система метрик, которая может интегрироваться с `folsom` (и другими), и рядом серверных модулей (graphite, collectd, `statsd`, Riak, SNMP, и т.д.). Один из новейших проектов на рынке.
- `ehmon`[8] для сбора вывода, идущего на стандартный вывод, который потом можно собрать другими специальными программами, например splunk и так далее.
- Различные самодельные решения, в основном использующие таблицы ETS и процессы, которые периодически сохраняют данные на диск.[9]
- Если у вас ничего нет, а проблемы уже начались, то функция, печатающая данные в цикле[10] поможет в трудную минуту.

В целом является хорошей идеей посвятить немного времени изучению этих инструментов, выбрать один и настроить хранение данных так, чтобы можно было изучать метрики, собранные за долгое время.

---

[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] Общие шаблоны использования их могут совпадать с приложением `ectr`, которое находится здесь https://github.com/heroku/ectr
[10] Приложение `recon` имеет функцию `recon:node_stats_print/2`.

5.1.1 Память

Расход памяти в виртуальной машине Erlang, показанный в большинстве инструментов, будет результатом вызова одного из вариантов команды `erlang:memory()`:

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

Это требует некоторых пояснений.

Первым делом все значения, которые возвращаются, измеряются в байтах и представляют *выделенную* память (память, активно используемую виртуальной машиной, а не зарезервированную операционной системой для виртуальной машины). Рано или поздно эти цифры начнут расходиться с информацией от операционной системы и будут выглядеть меньше.

Поле `total` содержит сумму памяти, которая использована для процессов (`processes`) и системы (`system`). Информация о памяти в системе неполная, если виртуальная машина не скомпилирована с особыми параметрами инструментации. Поле `processes` — это память, используемая Erlang-процессами, их стеками и кучами. Поле `system` — всё остальное: память ETS-таблиц, атомов в виртуальной машине, большие двоичные данные[11], и некоторые скрытые данные, которые, как я упоминал, могут приводить к несовпадению цифр.

Если вам нужно общее количество памяти, которой владеет виртуальная машина, то число, которое может упереться в системный лимит (`ulimit`), это значение получить изнутри виртуальной машины немного сложнее. Если вам нужны данные без вызова внешней программы `top` или `htop`, вам придётся заглянуть внутрь аллокаторов памяти виртуальной машины.[12]

К счастью recon имеет функцию `recon_alloc:memory/1`, которая выясняет ответ на этот вопрос, и аргумент функции может быть одним из следующих значений:

- `used` сообщает о памяти, активно используемой для выделенных в Erlang данных.
- `allocated` сообщает о зарезервированной в виртуальной машине памяти. Это включает использованную память, а также ещё незанятую, но уже выделенную операционной системой. Это то число, которое вам может понадобиться, если ваш узел упирается в системные ограничения, заданные посредством `ulimit`.

[11] Смотрите секцию 7.2 «Двоичные данные»

[12] Смотрите секцию 7.3.2 «Модель памяти в Erlang».

- `unused` сообщает о количестве памяти, которая зарезервирована виртуальной машиной но ещё не выделена. Эквивалент выражения `allocated - used`.
- `usage` возвращает значение соотношения использованной к выделенной памяти (число от 0 до 1).

Есть и другие опции, но вероятно вам они понадобятся лишь при чтении главы 7 «Утечки памяти».

5.1.2 Утилизация процессора

К несчастью для Erlang-программистов, расход процессорного времени оценивать очень сложно. Этому есть ряд причин:

- Виртуальная машина выполняет много задач, не связанных с процессами, когда дело касается планировки задач — большие расходы на планировку и расходы процессов на выполнение их работы трудно охарактеризовать.
- Виртуальная машина внутри использует модель редукций (уменьшений счётчика, или *reductions*), который представляет некоторое число некоторых действий. Каждый вызов функций, включая встроенные функции, изменяет этот счётчик. После заданного числа редукций процесс прекращает исполняться и планировщик передаёт управление следующему в очереди.
- Чтобы избежать засыпания, когда работы недостаточно, потоки, которые контролируют планировщики задач Erlang, занимают себя пустыми циклами. Это гарантирует наилучшую возможную скорость ответа при внезапном всплеске нагрузки. Флаг виртуальной машины `+sbwt none|very_short|short|medium|long|very_long` может изменить это значение при старте.

Эти факторы сочетаются и делают сложной задачей поиск абсолютной меры того, насколько ваш центральный процессор занят выполнением Erlang-кода. Обычным делом является ситуация, когда Erlang-узлы на производстве делают умеренное количество работы и используют ощутимо много процессорного времени, но способны выполнить много новой работы, используя оставшуюся свободную ёмкость, при возрастании нагрузки.

Самым точным представлением таких данных является время «настенных часов» (*wall*) планировщика. Это необязательная метрика, которую следует включить на узле вручную и опрашивать с некоторой периодичностью. Она покажет отношение времени, в течение которого планировщик выполняет процессы и обычный Erlang-код, встроенные функции (NIF, BIF), сборку мусора и так далее к времени, потраченному на пустые циклы ожидания или попытки выбрать новый процесс для запуска.

Это значение представляет скорее *занятость планировщика*, чем занятость центрального процессора. Чем выше это соотношение, тем больше занят планировщик.

В то время, как базовый способ употребления разъясняется в документации по Erlang/OTP[13], эти числа можно получить, вызывая библиотеку recon:

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

Функция `recon:scheduler_usage(N)` будет опрашивать в течение `N` миллисекунд (в данном случае 1 секунды) и выведет значение для каждого планировщика. В этом случае виртуальная машина имеет два очень нагруженных планировщика (занятых на уровне 99.2% и 93.7% соответственно) и два в основном свободных, с утилизацией сильно меньше 1%. Однако инструмент операционной системы, такой как `htop` сообщит нам другие цифры:

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

Причина такого результата в том, что некоторая часть «использованного» процессорного времени на самом деле доступна для выполнения работы на Erlang (предполагается, что некоторые планировщики заняты пустым ожиданием, а не выбором очередной задачи для запуска), но операционная система видит это как нагрузку.

Возможно и другое интересное поведение, когда утилизация планировщика может показать значение выше, чем сообщила операционная система. Планировщики, ждущие некоторого ресурса ОС считаются занятыми, поскольку они не в состоянии обработать новые задачи. Если сама ОС блокирует некторые не связанные с процессором задачи, то вполне возможна ситуация, когда планировщики Erlang не могут выполнить больше работы и сообщают о полной загрузке.

Эти шаблоны поведения могут вам пригодиться при планировании ёмкости вашей системы и могут быть более хорошими индикаторами запаса «свободного места над головой», чем взгляд на загрузку или использование процессора.

5.1.3 Процессы

Попытка получить общий вид всех процессов может оказаться полезной, когда вы пытаетесь оценить, сколько работы выполняется виртуальной машиной с точки зрения *задач*. Обычной хорошей практикой в Erlang является использование процессов для истинно параллельных

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

действий — на веб-серверах, вы обычно можете использовать один процесс на каждый запрос или каждое подключение, на системах с полноценным состоянием, вы могли бы использовать один процесс на пользователя — и таким образом число процессов на узле может быть метрикой текущей загрузки.

Большая часть инструментов, упомянутых в секции 5.1 «Вид сверху», будут отслеживать их так или иначе, но если вам нужно получить только их количество, то такого выражения будет достаточно:

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

Сохранение истории этого значения в виде например, графика, может оказаться полезным чтобы попытаться охарактеризовать нагрузку или выявить утечку процессов с помощью этой и других метрик, которые будут у вас под руками.

5.1.4 Порты

Подобно процессам, следует обратить внимание и на *порты*. Порты являются типом данных, которым представлены все виды подключений и сокетов, открытых во внешний мир: TCP-сокеты, UDP-сокеты, SCTP-сокеты, дескрипторы открытых файлов и так далее.

Имеется общая функция (снова-таки подобно процессам), которая их пересчитает: `length(erlang:ports())`. Однако, эта функция соединяет порты всех типов в один список. Вместо этого можно использовать `recon` для того, чтобы разобрать порты по типам:

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

Этот список содержит виды и количество портов каждого вида. Имя типа — строка, и её содержимое определяет сама виртуальная машина.

Все порты типа `*_inet` обычно являются сокетами, где приставка указывает на протокол (TCP, UDP, SCTP). Тип `efile` относится к файлам, а `"0/1"` и `"2/2"` — файловые дескрипторы стандартных каналов ввода-вывода (пара *stdin* и *stdout*) и *stderr* соответственно.

Большинство других типов получат имена согласно драйверу, который с ними работает, и будут являться примерами *программ, работающих с портами*[14] или *драйверов портов*[15].

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

Снова-таки отслеживание этих ресурсов может пригодиться для оценки загрузки или использования системы, поиска утечек и так далее.

5.2 Копаем поглубже

Когда что-то в «виде сверху» (или в ваших файлах журналов, возможно) указало в направлении потенциальной причины имеющейся проблемы, появляется интерес копать в поисках уже с конкретной целью. Находится ли процесс в странном состоянии? Может быть, ему поможет трассировка[16]! Трассировка — это прекрасный инструмент, когда у вас имеется некоторый вызов функции или входные или выходные данные, за полётом которых надо проследить, но часто перед тем, как они приведут нас к проблеме, требуются большие раскопки.

За пределами утечек памяти, для которых нужны свой особенный подход, и которые обсуждаются в главе 7 «Утечки памяти», чаще всего попадаются задачи, которые относятся к процессам и портам (файловым дескрипторам и сокетам).

5.2.1 Копаем в процессы

Со всех сторон процессы являются важной частью работающей Erlang-системы. И поскольку они такие важные для всего, есть множество причин желать узнать о них побольше. К счастью виртуальная машина даёт очень много информации, часть из которой безопасна, и ещё часть опасна для использования на производственном сервере (поскольку возвращаемые объёмы данных могут оказаться настолько большими, что копирование их в процесс интерактивного интерпретатора и попытка распечатать на экране могут убить узел).

Все значения могут быть получены с помощью функции `process_info(Pid, Ключ)` или `process_info(Pid, [СписокКлючей])`[17]. Вот некоторые часто используемые ключи[18]:

Общие (meta)

`dictionary` возвратит все записи в словаре процесса[19]. Обычно можно спокойно использовать, поскольку людям не следует сохранять гигабайты данных в этом хранилище.

`group_leader` — лидер группы процесса определяет, куда направляется ввод и вывод (файлы, печать на экран функцией `io:format/1-3`)[20]

[16] Смотрите главу 9 «Трассировка»

[17] В тех случаях, когда процессы содержат секретную информацию, данные можно скрыть с помощью вызова `process_flag(sensitive, true)` — это ограничит печать содержимого процесса на экран.

[18] Для списка *всех* опций загляните в документацию http://www.erlang.org/doc/man/erlang.html#process_info-2

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

[20] Для подробного описания смотрите книгу http://learnyousomeerlang.com/building-otp-applications#the-application-behaviour и http://erlang.org/doc/apps/stdlib/io_protocol.html.

registered_name если процесс имеет имя (зарегистрированное, например, вызовом `erlang:register/2`), оно будет здесь.

exiting процесс завершил работу но ещё не закончил очистку;

waiting процесс ждёт в операторе `receive ... end`;

running очевидно из названия, процесс выполняется в данный момент;

runnable готов выполняться, но пока не получил процессорного времени, потому что другой процесс выполняется в данный момент;

garbage_collecting выполняется сборка мусора;

suspended процесс заморожен встроенной (BIF) функцией или механизмом обратного давления сокета или драйвера порта, поскольку их буфер переполнен. Процесс сможет стать исполняемым снова, когда порт освободится для продолжения работы.

Сигналы (signals)

links покажет список всех связей, которые соединяют его с другими процессами и также портами (сокетами и дескрипторами файлов). Обычно является безопасной функцией, но будьте осторожны, поскольку на больших наблюдателях эта функция может вернуть многие тысячи элементов.

monitored_by даст список процессов, которые производят мониторинг текущего процесса (с помощью функции `erlang:monitor/2`).

monitors опция, противоположная предыдущей (`monitored_by`), которая даёт список всех процессов, мониторинг которых производит текущий процесс.

trap_exit будет иметь значение `true`, если процесс перехватывает сигналы выхода и превращает их в сообщения, иначе `false`.

Местонахождение (location)

current_function выведет текущую исполняемую функцию, в виде кортежа `{Модуль, ИмяФункции, Арность}`.

current_location покажет текущее положение внутри модуля, в виде кортежа `{Модуль, Функция, Арность, [{file, ИмяФайла}, {line, Строка}]}`.

current_stacktrace более многословный вариант предыдущей опции, который покажет текущий стек вызовов в виде списка «текущих положений».

initial_call покажет функцию, с которой начал исполнение процесс при своём порождении, записывается в виде `{Модуль, Функция, Арность}`. Это может помочь найти источник, где процесс был порождён, а не только место, где он сейчас исполняется.

Расход памяти (memory_used)

binary выводит все ссылки на блоки двоичных данных, считаемые по количеству ссылок (*refc*)[21] вместе с их размерами. Может оказаться небезопасной функцией, если процесс владеет большим их количеством.

garbage_collection содержит информацию относительно сборки мусора в процессе. Содержимое документировано, как «может быть изменено в будущем» и не должно расцениваться, как имеющее известную стабильную структуру. Информация содержит записи, такие как число сборок мусора, проведённых для данного процесса, опции для сборок мусора в режиме полных проходов (*full-sweep*), и размеры куч.

heap_size обычный Erlang-процесс содержит «старую» и «новую» версии кучи, которые подвергаются сборке мусора по поколениям. Эта запись показывает размер кучи новейшего поколения, и обычно включает в себя размер стека. Возвращаемое значение измеряется в *словах*, а не байтах.

memory возвратит, в *байтах*, размер процесса, включая стек, кучи и внутренние структуры, которые использует виртуальная машина и являющиеся частью процесса.

message_queue_len сообщает, сколько сообщений ждут в почтовом ящике процесса.

messages возвращает все сообщения в почтовом ящике процесса. Эта опция *исключительно опасна* при вызове на производственном сервере, поскольку почтовые ящики могут хранить миллионы сообщений. Если вы отлаживаете процесс, который оказался заблокирован, то *всегда* сначала проверяйте длину почтового ящика посредством `message_queue_len`, чтобы убедиться в безопасности получения всех сообщений.

total_heap_size подобно опции `heap_size`, но также включает в себя все другие фрагменты кучи, включая старое поколение. Возвращаемое значение измеряется в *словах*.

Работа (work)

reductions виртуальная машина выполняет планирование задач на основании числа *редукций*, некоторой произвольно выбранной единицы работы, которая позволяет переносимые на разные платформы реализации планирования задач (планирование на основании времени обычно трудно заставить работать эффективно на всех операционных системах, где работает Erlang). Чем выше число редукций, тем больше работы на центральном процессоре и вызовов функций выполнил процесс.

К счастью для всех часто употребляемых опций, которые к тому же безопасны, recon имеет вспомогательную функцию `recon:info/1`:

```
1> recon:info("<0.12.0>").
[{meta,[{registered_name,rex},
```

[21] Смотрите секцию 7.2 «Двоичные данные»

```
       {dictionary,[{'$ancestors',[kernel_sup,<0.10.0>]},
                    {'$initial_call',{rpc,init,1}}]},
       {group_leader,<0.9.0>},
       {status,waiting}]},
 {signals,[{links,[<0.11.0>]},
          {monitors,[]},
          {monitored_by,[]},
          {trap_exit,true}]},
 {location,[{initial_call,{proc_lib,init_p,5}},
           {current_stacktrace,[{gen_server,loop,6,
                                 [{file,"gen_server.erl"},{line,358}]},
                                {proc_lib,init_p_do_apply,3,
                                 [{file,"proc_lib.erl"},{line,239}]}]}]},
 {memory_used,[{memory,2808},
               {message_queue_len,0},
               {heap_size,233},
               {total_heap_size,233},
               {garbage_collection,[{min_bin_vheap_size,46422},
                                    {min_heap_size,233},
                                    {fullsweep_after,65535},
                                    {minor_gcs,0}]}]},
 {work,[{reductions,35}]}]
```

Для удобства `recon:info/1` примет первым аргументом любое значение, похожее на идентификатор процесса и обработает его: подойдут обычные pid, строки (`"<0.12.0>"`), зарегистрированные атомы, глобальные имена (`{global, Атом}`), имена, зарегистрированные в отдельном реестре (например `gproc`: `{via, gproc, Имя}`), или кортежи в виде (`{0,12,0}`). Процесс всего лишь должен находиться локально на узле, который вы сейчас отлаживаете.

Если требуются все значения в категории, можно использовать её название:

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

Можно использовать точно так же, как и стандартную `process_info/2`:

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

Эта вторая форма может быть использована для получения небезопасной информациию

Со всеми этими данными можно выяснить всё, что нам требуется для отладки системы. Сложнее всего определить, какие процессы потребуют отладки, сравнивая эти данные для каждого процесса и глобальную статистику.

Глядя на высокий расход памяти, например, интересно было бы получить все процессы данного узла и отобрать `N` самых больших потребителей. Используя атрибуты, показанные выше, и функцию `recon:proc_count(Атрибут, N)`, можно получить эти результаты:

```
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}}]}]
```

Любые из атрибутов, упомянутых ранее, могут сработать. Для узлов, на которых выполняются долгоживущие процессы, могущие причинить сложности, эта функция довольно полезна.

Однако имеется проблема, когда большинство процессов короткоживущие, обычно слишком короткие, чтобы их можно было рассмотреть другими инструментами, или когда нам требуется скользящий интервал (*moving window*) (например, какие процессы заняты накоплением памяти или выполнением кода *прямо сейчас*).

На этот случай recon имеет функцию `recon:proc_window(Атрибут, Количество, Миллисекунды)`.

Важно рассматривать эту функцию как «снимок» состояния на скользящем интервале. Время жизни программы во время взятия образцов может выглядеть так:

```
--w---- [Образец1] ---x-------------y----- [Образец2] ---z--->
```

Функция также возьмёт два образца с интервалом, заданным параметром `Миллисекунды`.

Некоторые процессы будут живы после момента `w` и умрут в момент `x`, некоторые между `y` и `z`, а некоторые между `x` и `y`. Эти образцы не будут иметь большого значения, поскольку они неполны.

Если подавляющее большинство ваших процессов исполняются в интервале времени между `x` и `y` (в абсолютном измерении), вам следует убедиться, что окно интервала сбора образцов короче этого времени, так чтобы время жизни процессов было аналогично интервалу между `w` и `z`. Если не следовать этому правилу, то результаты могут быть искажены: долгоживущие процессы, имеющие в 10 раз больше времени для накопления данных (например, редукций),

будут выглядеть огромными потребителями ресурса, когда на самом деле они таковыми не являются[22].

Функция после выполнения даст подобные следующему результаты:

```
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}}]}]
```

С этими двумя функциями становится возможным рассмотреть конкретный процесс, который причиняет проблемы или ведёт себя неадекватно.

5.2.2 OTP-процессы

Когда процессы, которые вызывают вопросы, являются процессами OTP (большинство процессов на производстве определённо должны быть такими), то вы моментально приобретаете ещё несколько инструментов для их изучения.

В общем модуль `sys`[23] это то, что вам потребуется. Прочтите документацию для него и вы поймёте, почему он такой полезный. Он может выдать следующую информацию для любого OTP-процесса:

- журналирование всех сообщений и переходов между состояниями, как в консоль, так и в файл или во внутренний буфер, который затем можно анализировать;
- статистика (редукции, счёт числа сообщений, время и так далее);
- статус процесса (метаданные, включая состояние);
- выборка состояния процесса (содержимое той самой знаменитой записи `#state{}`);
- изменение содержимого этого состояния;
- ваши собственные функции могут быть использованы в качестве функций обратного вызова.

Также он обеспечивает возможность заморозить и продолжить исполнение процесса.

Я не буду углубляться в подробное описание этих функций, просто знайте, что такие возможности существуют.

[22] Внимание: эта функция зависит от данных, собранных несколькими снимками, и затем хранит их в словаре под разными ключами. Это может привести к росту расхода памяти, если у вас имеются десятки тысяч процессов и вы занимаетесь анализом в течение некоторого времени.

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

5.2.3 Порты

Подобно процессам, порты в Erlang позволяют выполнять разную диагностику. Доступ к информации можно получить через функцию `erlang:port_info(Порт, Ключ)`, и ещё больше можно узнать с помощью модуля `inet`. Большая часть этой диагностики также собрана в функциях `recon:port_info/1-2`, которые работают подобно функциям анализа процессов.

Общие (meta)

`id` внутренний индекс порта. Не очень полезен кроме как для того, чтобы различать порты между собой.

`name` тип порта — содержит имена, такие как `"tcp_inet"`, `"udp_inet"`, или `"efile"`, для примера.

`os_pid` если порт не является сокетом модуля inet, а представляет внешний процесс или приложение, то это значение содержит идентификатор процесса ОС этой программы.

Сигналы (signals)

`connected` каждый порт имеет контролирующий процесс, который за него отвечает, `connected` возвращает идентификатор этого процесса.

`links` порты могут быть связаны с процессами, так же как и обычные процессы. Список связанных процессов содержится здесь. Если процессом не владеют или не связаны с ним вручную множество других процессов, то это значение должно быть безопасным.

`monitors` порты, представляющие внешние программы, могут позволить этим программам мониторить процессы Erlang. Такие процессы перечислены здесь.

Ввод-вывод (io)

`input` число байтов, прочитанных из порта.

`output` число байтов, записанных в порт.

Расход памяти (memory_used)

`memory` покажет размер памяти (в байтах), выделенной системой Erlang для этого порта. Число это обычно небольшое и не включает размер памяти, которую порт выделил для себя сам.

`queue_size` программы портов имеют отдельную очередь, которая называется очередью драйвера[24]. Эта опция вернёт размер этой очереди, в байтах.

Зависящие от типа (type)

[24]Очередь драйвера доступна для накопления вывода из эмулятора к драйверу (данные в обратную сторону от драйвера к эмулятору накапливаются в обычных очередях сообщений Erlang). Это может пригодиться, если драйвер ждёь ответа например от медленных устройств, и хочет быстрее вернуть управление эмулятору.

Сетевые порты вернёт данные по сокету, включая статистику[25], локальный адрес и номер порта для сокета (`sockname`), и использованные опции модуля inet[26]

Прочие в данное время другие типы портов, кроме как порты inet, не поддерживаются библиотекой recon, и поэтому вы можете увидеть здесь пустой список.

Список можно получить таким образом:

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

Дополнительно к этому, как и для процессов, имеются функции поиска конкретных проблемных портов. На данный момент recon поддерживает их только для портов модуля inet и с ограниченным числом атрибутов: число октетов (байтов) отправлено, принято, или то и другое (`send_oct`, `recv_oct`, `oct`, соответственно), или число пакетов отправлено, принято, или то и другое (`send_cnt`, `recv_cnt`, `cnt`, соответственно).

Итак для накопительной суммы, которая может помочь найти тех, кто медленно но уверенно съедает вашу пропускную способность сети:

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

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

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

```
 {#Port<0.6757848>,15762095249,
  [{recv_oct,15762095249},{send_oct,0}]},
 {#Port<0.6718690>,15630954707,
  [{recv_oct,15630954707},{send_oct,0}]}]
```

Это показывает нам, что некоторые порты выполняют лишь приём и потребляют очень много байтов. Затем можно использовать `recon:port_info("#Port<0.6821166>")`, чтобы вкопаться поглубже и найти, кто владеет сокетом, и что с ним происходит.

Или для любого другого случая, можно посмотреть на тех, кто посылает больше всего данных в течение интервала времени[27] с помощью функции `recon:inet_window(Атрибут, Количество, Миллисекунды)`:

```
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}]}]
```

Для этого примера, значение в середине кортежа это изменение `send_oct` (или любой выбранный атрибут для каждого вызова), во время выбранного интервала времени (в данном случае 5 секунд).

Нужно проделать дополнительную работу вручную, тобы установить связь между странно ведущим себя портом и процессом (и затем, вероятно, конкретным клиентом или пользователем системы), но все инструменты имеются в наличии.

5.3 Упражнения

Вопросы для закрепления материала

1. Какого рода значения можно получить о памяти Erlang?
2. Назовите ценную метрику, которую можно получить при обзоре состояния процессов издалека, в глобальном масштабе.
3. Что такое порт, и как следует выполнять его глобальный мониторинг?
4. Почему не следует доверять показаниям утилит операционной системы `top` или `htop` относительно расхода процессорного времени в Erlang-системах?
5. Назовите два вида информации о процессах, связанной с сигналами.
6. Как можно узнать, какой код сейчас исполняет некоторый процесс?

[27] Смотрите пояснения к `recon:proc_window/3` в предыдущей подсекции.

7. Какие разные типы информации о памяти имеются для каждого конкретного процесса?
8. Как можно узнать, что процесс выполняет очень много работы?
9. Назовите несколько значений, получение которых может оказаться опасным при диагностике и отладке?
10. Какие дополнительные возможности доступны через модуль sys для OTP-процессов?
11. Какие значения можно получить при диагностике интернет сокетов (портов модуля inets)?
12. Как можно узнать тип порта (файловый, TCP- или UDP-сокет)?

Открытые вопросы

1. Почему в глобальных метриках для вас очень полезно иметь историю значений за долгое время?
2. Какая из функций `recon:proc_count/2` и `recon:proc_window/3` больше подходит для поиска следующих проблем:
   (a) Количество редукций
   (b) Память
   (c) Длина очереди сообщений
3. Как можно выяснить, какой процесс является наблюдателем для некоторого процесса?
4. Когда вам следует использовать `recon:inet_count/2`? `recon:inet_window/3`?
5. Чем можно объяснить разницу в сообщаемом расходе памяти между операционной системой и внутренними функциями диагностики Erlang?
6. Почему Erlang-узел иногда выглядит очень занятым снаружи, когда на самом деле он ничего не делает?
7. Как можно узнать пропорцию числа процессов на узле, которые готовы к работе, но не могут немедленно получить от планировщика процессорное время?

Тренировочные задания

Используя код, который находится по адресу https://github.com/ferd/recon_demo ответьте на вопросы:

1. Что такое память системы?
2. Использует ли узел очень много ресурсов процессора?

3. Переполняется ли почтовый ящик какого-то из процессов?
4. Какой из разговорчивых процессов (`council_member`) занимает больше всего памяти?
5. Какой из разговорчивых процессов потребляет больше всего ресурсов процессора?
6. Какой из разговорчивых процессов потребляет больше всего ресурсов сети?
7. Какой из разговорчивых процессов посылает больше всего сообщений по протоколу TCP? А какой меньше всего?
8. Можете ли вы найти такой процесс, который удерживает множество подключений или открытых дескрипторов файлов на данном узле?
9. Можете ли вы найти, какая функция вызывается наибольшим количеством процессов одновременно в данный момент на этом узле?

Chapter 6

Читаем файлы аварийных дампов

Когда Erlang-узел завершает работу аварийно, он создаёт файл аварийного дампа[1].

Формат такого файла в основном описан в официальной документации[2], и любой, кто желает копнуть поглубже, вероятно сможет разобраться, что означают данные, глядя в документацию. Есть некоторые значения, которые особенно трудно понять, если не понимать ту часть виртуальной машины, на которую они ссылаются, но для этого документа такое описание может оказаться неоправданно сложным.

Аварийный дамп получает имя `erl_crash.dump` и находится в директории, в которой по умолчанию исполнялся Erlang-процесс. Это поведение (и имя файла) могут быть изменены с помощью указания переменной окружения `ERL_CRASH_DUMP`[3].

6.1 Общий вид

Чтение аварийного дампа пригодится для того, чтобы *посмертно* выяснить возможные причины, почему узел погиб. Одним из способов быстро посмотреть на эти данные является скрипт `erl_crashdump_analyzer.sh` в составе recon[4] — просто выполните его и укажите параметром файл аварийного дампа.

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

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

[1] Если он не нарушил ограничения ОС (ulimit) во время записи файла или не упал с ошибкой сегментации (segfault).

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

[3] Команды маршрутизации и телеметрии в компании Heroku используют приложение `heroku_crashdumps`, чтобы устанавливать путь и имя файлов дампов. Его можно добавить в проект, чтобы давать имя дампам во время запуска и помещать их в заранее заданную директорию.

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

```
(of type "old_heap").

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

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

Error logger queue length:
===
0

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

Number of processes:
===
36496

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

```
===
      1 284745853
      1 5157867
      1 4298223
      2 196650
     12 121536

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

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

Этот дамп не преподносит проблему прямо вам под нос, но даст хорошие подсказки, куда смотреть. Например, у узла здесь закончилась память и было использовано 11.079 мегабайт из 15 гигабайт (я знаю максимальную цифру, потому что это размер виртуальной машины, которую мы использовали в этот день!). Это может быть симптомом следующих проблем:

- фрагментации памяти;
- memory leaks in C code or drivers;
- утечек памяти в C-коде или драйверах;
- очень много блоков памяти, которую следовало бы собрать сборщику мусора перед тем, как создавать аварийный дамп[5].

В общем, ищите среди цифр памяти что угодно, привлекающее внимание. Сравнивайте это с числом процессов и размерами почтовых ящиков. Одно может объяснить причины второго.

В этом конкретном дампе один из процессов имел 5 миллионов сообщений в почтовом ящике. Это всё объясняет. Он не выбирал все сообщения, а лишь некоторые, его интересующие, или просто был перегружен. Также имеются десятки процессов с сотнями сообщений в очереди — это

[5]Особенно бросается в глаза память больших двоичных данных, которая находится в глобальной куче, но ей достаётся проход сборщика мусора перед созданием дампа. Расход памяти на двоичные данные, таким образом, в этом отчёте может быть занижен. Смотрите главу 7 «Утечки памяти» для подробного описания.

может указывать на перегрузку либо борьбу за ресурсы. Трудно дать общий совет для дампа, который может иметься в вашем случае, но всё-же имеются некоторые подсказки, с которых можно начать выяснять причины проблем.

6.2 Полные почтовые ящики

Для нагруженных почтовых ящиков, взгляд на большие счётчики является лучшим способом выяснить наличие проблемы. Если имеется один большой почтовый ящик — расследуйте процесс в аварийном дампе. Выясните, происходит ли это потому, что процесс не выбирает какой-то вид сообщений, или перегружен. Если у вас окажется множество нагруженных почтовых ящиков, вы можете захотеть воспользоваться скриптом `queue_fun.awk` в пакете recon, чтобы выяснить, какую функцию все они выполняли на момент аварийного завершения:

```
$ 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
...
```

Этот скрипт пробежит по содержимому аварийного дампа и выведет все функции, которые были подготовлены планировщиком к запуску, и имели не менее 10.000 сообщений в своих почтовых ящиках. В моём случае, например, сценарий показал, что весь узел был заблокирован в ожидании ввода-вывода для печати посредством `io:format/2`.

6.3 Слишком много (или мало) процессов

Количество процессов пригодится, если вы знаете обычную повседневную цифру количества для вашего узла[6], чтобы понять, когда начнёт происходить что-то необычное.

Количество больше обычного может указывать на некоторую утечку или перегрузку, в зависимости от вашего приложения.

Если количество процессов очень мало, в сравнении с обычным количеством, проверьте нет ли в журналах сообщений вроде следующего:

[6] Смотрите подсекцию 5.1.3 «Процессы» для подробностей

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

В таком случае проблемой является то, что некоторое приложение (`<ИмяПриложения>`) достигло максимальной частоты перезапусков своих наблюдателей и это потребовало аварийной остановки всего узла. Следует вычитать журналы в поисках ошибок, приведших к каскадному отказу.

6.4 Слишком много портов

Подобно количеству процессов, количество портов — это простая и в целом полезная характеристика, когда вы знаете обычные значения для вашей системы[7].

Большое количество может быть последствием перегрузки, DoS атак или старых добрых утечек ресурсов. Глядя на тип утекших портов (TCP, UDP или файлы) можно выяснить, была ли борьба за какой-то ресурс или код, с ними работающий, содержит ошибку.

6.5 Невозможно выделить память

Эти ошибки встречаются намного чаще других. Причин такой остановки настолько много, что я посвятил их пониманию и выполнению необходимой отладки на живых системах целую главу 7 «Утечки памяти».

В любом случае, аварийный дамп поможет выяснить, что случилось, по факту когда программа завершилась. Почтовые ящики процессов — это отдельные кучи и обычно являются хорошими индикаторами возникновения проблем. Если у вас заканчивается память при том, что ни один почтовый ящик не вырос до заоблачных высот, посмотрите на кучи процессов и размеры стеков, которые возвращает скрипт recon.

Если у вас имеются резко выделяющиеся числа в отчёте, то вы сможете составить некоторый ограниченный набор процессов, которые подозреваются в перерасходе памяти. В случае же если все они более менее равны, посмотрите на сообщаемое количество использованной памяти, вдруг она покажет большие числа.

Если всё выглядит более-менее разумно, перейдите в секцию с отчётом о памяти («Memory») и проверьте, если один из типов памяти (ETS или двоичная куча, например) выглядит неоправданно большим. Они могут указывать на утечки ресурсов, которых вы не ожидали.

6.6 Упражнения

Вопросы для закрепления материала

1. Как можно выбрать директорию, в которой будет создан аварийный дамп?

[7] Смотрите подсекцию 5.1.4 «Порты» для подробностей

2. Назовите что следует искать в дампе, когда причиной смерти узла был назван недостаток памяти?

3. Что следует искать, если количество процессов необычно мало?

4. Как вы узнали, причиной смерти узла стало то, что один из процессов начал использовать слишком много памяти. Что можно сделать, чтобы найти этот процесс?

Тренировочные задания

Используя анализ аварийного дампа в секции 6.1 «Общий вид»:

1. Какие числа являются необычно выделяющимися и могут указывать на возможные причины?

2. Могли ли повторяющиеся ошибки стать проблемой? Если нет, что же тогда могло?

Chapter 7

Утечки памяти

Для Erlang-узла существует огромное множество причин утечек памяти. Их выявление может оказаться как крайне простым, так и удивительно трудным (к счастью второй вид также и более редкий) и вполне вероятно, вы никогда не встретите таких проблем.

Об утечках памяти можно узнать двумя способами:

1. Аварийный дамп умершего узла (смотрите главу 6 «Читаем файлы аварийных дампов»);
2. Если вам удалось найти подозрительную тенденцию в изменении цифр данных, которые вы мониторите.

Эта глава в основном сосредоточится на втором виде утечек, потому что их легче расследовать и смотреть, как они растут в реальном времени. Мы сосредоточимся на поиске того, что растёт на данном узле, обычные способы оказания первой помощи, что делать с утечками бинарных данных (они являются особым случаем), и обнаружение фрагментации памяти.

7.1 Общие источники утечек

Когда кто-нибудь зовёт на помощь и говорит: «О, нет, мои Erlang-узлы постоянно падают», первым дело следует попросить больше информации. Интересным будет задавать следующие вопросы и смотреть на такие подсказки:

- Есть ли у вас аварийный дамп и указывает ли он конкретно на память? Если нет, проблема может быть не связана с памятью. Обязательно копните поглубже, там очень много интересного.
- Являются ли падения циклическими? Насколько легко их предсказать? Что ещё происходит примерно в то же время и может повлиять на проблему?
- Совпадают ли аварийные падения с всплесками нагрузки ваших систем, или происходят ли они в разное время? Падения, происходящие особенно *во время* пиковых нагрузок, часто связаны с плохой обработкой ситуаций перегрузки (смотрите главу 3 «В ожидании

перегрузки»). Падения, происходящие в разное время, даже когда нагрузка спадает после пика, могут на самом деле являться проблемами с памятью.

Если все признаки указывают на утечку памяти, установите в вашу систему одну из библиотек метрики, которые упоминались в главе 5 «Метрики времени выполнения» и/или `recon` и приготовьтесь нырнуть в данные.[1].

Первое, что следует искать в любом из этих случаев — это тренды (тенденции). Проверяйте все виды памяти, используя вызов `erlang:memory()` или некий аналогичный вариант, который может иметься в библиотеке или системе сбора метрик. Смотрите на следующие моменты:

- Растёт ли один из видов памяти быстрее других?
- Занимает ли какой-нибудь тип памяти большую часть доступного пространства?
- Есть ли какой-нибудь тип памяти, который никогда заметно не уменьшает потребление и только растёт вверх (кроме атомов)? Растущий график в форме пилы тоже является тревожным.

Имеется ряд решений зависящих от типа памяти, которая начала расти.

7.1.1 Атомы

Не создавайте атомы динамически! Атомы сохраняются в глобальной таблице навсегда. Ищите места, где вы вызываете `erlang:binary_to_term/1` и `erlang:list_to_atom/1`, и обдумайте переключение на более безопасные варианты (`erlang:binary_to_term(Bin, [safe])` и `erlang:list_to_existing_atom/1`).

Если вы использовали стандартное приложение `xmerl` в библиотеке Erlang, продумайте возможность перехода на альтернативное решение[2] или найти способ добавить собственный SAX-парсер, который мог бы оказаться безопасным[3].

Если вы не делали ни того ни другого, задумайтесь, как вы взаимодействуете с вашим узлом. Однажны на реальной производственной системе оказалось так, что один из инструментов для подключения к узлам использовал случайные имена, или создавал узлы со случайными именами, которые подключались друг к другу с центрального сервера[4]. Имена узлов Erlang превращаются в атомы, так что этого оказалось достаточно, чтобы медленно и уверенно заполнить пространство таблицы атомов. Убедитесь, что вы выбираете «случайные» атомы из некоторого фиксированного набора, или создаёте их достаточно медленно так, чтобы это не оказалось проблемой через какое-то время.

---

[1] Смотрите главу 4 «Подключение к удалённым узлам» если вам нужны подсказки, как подключиться к работающей Erlang-системе.

[2] Я не против использования exml или erlsom

[3] Смотрите комментарий Ulf Wiger в рассылке http://erlang.org/pipermail/erlang-questions/2013-July/074901.html

[4] Это общепринятый подход к выяснению того, как узлы подключаются друг к другу: держите два центральных узла с известными неизменными именами и остальные к ним подключаются. Тогда взаимные переподключения узлов произойдут автоматически.

7.1.2 Двоичные данные

Смотрите отдельную посвящённую им секцию 7.2 «Двоичные данные».

7.1.3 Код

Код на Erlang-узлах загружается в свою особенную зону памяти, и остаётся там до тех пор, пока сборщик мусора его не освободит. Только две копии каждого модуля могут существовать одновременно в памяти, так что поиск очень больших модулей должен быть относительно лёгким.

Если никакой модуль не выделяется из общего ряда, поищите код, скомпилированный с включенным HiPE[5]. HiPE-код, в отличие от обычного BEAM-кода, является машинным кодом и не может быть подхвачен сборщиком мусора, когда загружаются новые версии. Память может накапливаться, обычно медленно, если много или большие HiPE-модули загружаются во время работы программы.

Как вариант, можно поискать странные модули, которые вы не загружали на узел и устроить панику, если кто-то смог получить доступ к вашей системе.

7.1.4 ETS

ETS-таблицы никогда не подвергаются сборке мусора и будут сохранять имеющийся расход памяти так долго, пока в них не удаляются записи. Только удаление записей вручную (или очистка/удаление всей таблицы) может вернуть память.

В редких случаях, когда у вас действительно потекла ETS-таблица, используйте недокументированную функцию `ets:i()` в интерактивном интерпретаторе. Она распечатает информацию о количестве записей (`size`) и сколько памяти они занимают (`mem`). Попробуйте использовать эти данные для поиска чего-нибудь подозрительного.

Совершенно возможен случай, когда все данные в таблице находятся честно и согласно алгоритму вашей программы, и вы столкнулись с проблемой необходимости разделения таблицы на секции (*sharding*) и распределения на разные узлы. Это выходит за пределы книги, так что удачи вам. Однако стоит посмотреть на опцию сжатия таблиц, если вам нужно выиграть время[6].

7.1.5 Процессы

Есть множество разных причин роста памяти процессов. Самые интересные случаи относятся к нескольким самым частым причинам: утечки процессов (то есть, течёт не память, а теряются целые процессы), утечки памяти в особых процессах, и так далее. Возможно в вашем случае будет более одной причины одновременно, так что смотрите сразу на многие метрики. Заметьте, что счёт процессов был описан ранее и поэтому на нём здесь не останавливаемся.

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

[6] Смотрите опцию `compressed` для `ets:new/2`

Связи и мониторы

Является ли глобальное количество процессов индикатором утечки? Если так, то вам может понадобиться рассмотреть процессы, с которыми была разорвана связь (*unlinked*), или заглянуть в списки дочерних процессов наблюдателей на случай, если они выглядят необычно.

Поиск процессов, с которыми была разорвана связь и мониторинг — это несложная задача при помощи нескольких простых команд.

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

Эта команда вернёт список процессов не имеющих ни того ни другого. Для наблюдателей хорошей подсказкой может быть простая выборка с помощью `supervisor:count_children(SupervisorPidOrName)` и проверка на глаз: выглядит ли список хорошо или подозрительно.

Использованная память

Модель памяти с отдельной кучей для каждого процесса коротко описана в подсекции 7.3.2 «Взгляд на уровне процесса», но в общих словах вы можете найти конкретный процесс, использующий больше всех памяти, если посмотрите на атрибут `memory`. Можно смотреть за абсолютными значениями или за их изменением в скользящем интервале.

Для утечек памяти, если только у вас нет предсказуемого быстрого роста, стоит копнуть первым делом абсолютные значения:

```
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}}]}]
```

Атрибуты, которые могут оказаться интересными для анализа, кроме `memory`, — это любые другие поля описанные в подсекции 5.2.1 «Копаем в процессы», включая `message_queue_len`, но счётчик `memory` обычно включает в себя все остальные типы памяти.

Утечки сборщика мусора

Очень возможна ситуация, что процесс использует много памяти, но лишь в течение коротких промежутков времени. Для долгоживущих узлов с большими накладными расходами на выполнение работы это обычно не является проблемой, но когда ресурсы памяти становятся ограничены, то вы можете начать обдумывать план избавления от такого поведения.

Мониторинг всех сборок мусора в реальном времени из консоли интерпретатора может оказаться дорогим удовольствием. Вместо этого лучшим из решений может оказаться установка системного монитора Erlang[7].

Системный монитор позволит вам отслеживать такую информацию, как долгие периоды сборки мусора и большие кучи процессов, а также и другие интересные вещи. Монитор можно временно запустить такими командами:

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

Первая команда проверяет, что ничего (или никто) ещё не начал использовать системный монитор — вам не хотелось бы забрать его у существующего приложения или испортить работу сотруднику.

Вторая команда включает режим отправки уведомлений каждый раз, когда сборка мусора занимает дольше 500 миллисекунд. Результат можно увидеть с помощью третьей команды (которая принимает сообщения, пришедшие в процесс интерактивного интерпретатора). Попробуйте также добавить `{large_heap, КоличествоСлов}`, если вам интересна ситуация роста размера кучи более заданного количества слов. Будьте осторожнее с запуском и пробуйте сначала большие значения, если не уверены в том, какие числа выбрать. Вы не хотите затопить

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

почтовый ящик своего процесса сообщениями о том, что все кучи на узле имеют размер больше 1 слова.

Команда 5 отключает системный монитор (выход или смерть процесса монитора тоже его отключит) и команда 6 проверяет, что всё сработало.

Вы можете затем выяснить, совпадают ли подобные сообщения монитора с увеличениями размеров памяти, которые по видимости приводят к утечкам или перерасходу, и попробовать найти причину до того, как всё станет слишком плохо. Быстрое реагирование и раскопки информации по процессу (возможно с помощью `recon:info/1`) могут помочь выяснить, что не так с приложением.

7.1.6 Ничего особенного

Если ничего из предыдущего материала не привлекло вашего внимания, то причиной могут быть утечки памяти двоичных данных (секция 7.2 «Двоичные данные») и фрагментация памяти (секция 7.3 «Фрагментация памяти»). Если даже это подозрение не оправдалось, то вполне возможно причиной является C-драйвер, встроенная (NIF) функция или даже сама виртуальная машина. Конечно, возможным сценарием может быть и то, что нагрузка на узле и расход памяти были пропорциональны, и ничего на самом деле не утекало, а расходовалось в результате нормальной работы. Системе просто требуются больше памяти или новые узлы.

7.2 Двоичные данные

Двоичные данные в Erlang на низком уровне делятся на два вида: значения на куче процесса (*ProcBin*) и большие данные в двоичной куче, освобождаемые автоматически с подсчётом ссылок (*Refc*)[8]. Двоичные значения короче 64 байтов располагаются непосредственно в куче процесса и весь их жизненный цикл протекает там же. Двоичные значения большого размера размещаются в глобальной куче, предназначенной только для двоичных данных, и каждый процесс, который использует блок данных, хранит ссылку на него в своей локальной куче. Эти блоки данных освобождаются автоматически только тогда, когда все ссылки на них очищены сборщиком мусора во всех процессах, которые когда-либо ссылались на это значение.

Этот механизм работает совершенно удовлетворительно в 99% случаев. Однако в некоторых ситуациях процесс может:

1. выполнять слишком мало работы, чтобы оправдать выделение памяти и сборку мусора;
2. вырастить большой стек или кучу с разными структурами данных, собрать их сборщиком, и затем заняться работой с большими двоичными данными. Повторное заполнение кучи такими блоками данных (даже несмотря на то, что для учёта реального размера таких блоков используется виртуальная куча) может занять долгое время, давая большие задержки между сборками мусора, соответственно и освобождение этой памяти будет отложено на будущее.

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

7.2.1 Обнаружение утечек

Обнаружить утечку блоков двоичных данных, которые освобождаются по счётчику ссылок (*refc*), довольно просто: измерьте все ссылки на двоичные данные во всех процессах (используя атрибут `binary`), выполните глобальную сборку мусора, сделайте снимок ещё раз и подсчитайте разницу.

Это можно непосредственно выполнить с помощью функции `recon:bin_leak(Количество)` и затем сравнить общую память на узле до и после вызова:

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

Это покажет нам, сколько отдельных блоков двоичных данных были заняты и затем освобождены каждым процессов в виде разности. Значение `-5580` означает, что после вызова стало на 5580 меньше блоков двоичных данных (*refc*), счётчики ссылок которых обнулились после первой сборки мусора.

Вполне обычная ситуация иметь некоторое их количество в любой момент времени, и не любое уменьшение их количества указывает на наличие проблем. Если вы увидите, что использованная виртуальной машиной память сильно уменьшилась после вызова этой команты, у вас было значительное количество блоков двоичных данных, ссылки на которые удерживались «спящими» процессами.

Подобно этому, если вы увидите, что некоторые процессы удерживают впечатляюще большие их количества[9], это может быть верным знаком наличия у вас проблемы.

Вы можете продолжить поиск самых активных пользователей блоков двоичной памяти, используя специальный атрибут `binary_memory`, который поддерживается в `recon`:

[9] Во время наших расследований утечек памяти в Heroku мы видели процессы, удерживавшие сотни тысяч блоков памяти!

```
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}}]}]
```

Эта команда вернёт первые `N` процессов, отсортированные по количеству памяти, которую удерживают refc-блоки двоичных данных, и может помочь указать на конкретные процессы, владеющие небольшим количеством особо больших блоков, чего нельзя было бы увидеть, если бы вы анализировали только их количество. Вы можете решить выполнить эту функцию *до* `recon:bin_leak/1`, поскольку она соберёт мусор на всём узле и это повлияет на результаты.

7.2.2 Исправляем утечки

Когда вы установили факт, что у вас течёт память двоичных данных (*refc*) с помощью `recon:bin_leak(Количество)`, будет несложно посмотреть первые из процессов в списке и разобраться, кто они и какую работу выполняют.

В общих словах, утечки памяти двоичных данных решаются несколькими способами в зависимости от источника проблем:

- вызывать сборку мусора вручную с заданной периодичностью (неуклюжее решение, но в целом довольно эффективное);
- перестать пользоваться большими двоичными данными в целом (часто неподходящее решение);
- использовать копирование с помощью `binary:copy/1-2`[10] если требуется хранение небольшого фрагмента (обычно короче, чем 64 байта) от большого блока данных[11];
- перенести работу, задействующую большие двоичные данные во временные процессы, которые умрут сами по завершении работы (некая упрощённая форма сборки мусора вручную);
- добавить вызовы спящего режима (*hibernate*) там, где это имеет смысл (для часто спящих процессов — это самый лучший подход).

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

[11] Идея скопировать и более длинный фрагмент двоичных refc-данных может оказаться разумной. Например, копирование 10 мегабайт из 2-гигабайтового блока данных стоит своей небольшой цены, поскольку позволяет освободить 2-гигабайтовый блок в то время, как более короткий фрагмент будет нужен вам долгое время.

Первые два варианта, честно говоря, можно считать крайними случаями и не следует пытаться их выполнять до того, как вы попробовали другие варианты. А вот последние три варианта обычно самые подходящие.

Маршрутизация двоичных данных

Есть одно особое решение для одного специального случая, о котором сообщали некоторые пользователи Erlang-систем. Проблемная ситуация обычно имела процесс-посредник, который маршрутизировал движение двоичных данных от одного к другому процессу. Этот процесс-посредник, таким образом, получал ссылку на каждый проходящий блок двоичных данных и рисковал оказаться основной причиной утечек или позднего освобождения двоичных данных.

Решением такой ситуации является перенос работы к исполнителю — процесс-маршрутизатор возвращал бы идентификатор процесса-получателя блока данных и вызывающий процесс передавал бы данные ему сам. Таким образом только те процессы, которым *нужны* эти данные — прикоснутся к ним и создадут ссылки на них в своих кучах.

В качестве решения для этого можно реализовать прозрачные функции API маршрутизатора, позволяющие изменить способ передачи данных не требуя никаких изменений в клиентах.

7.3 Фрагментация памяти

Проблемы фрагментации памяти очень близко связаны с моделью памяти Erlang, как описывалось в секции 7.3.2 «Модель памяти в Erlang». Эта проблема — одна из самых сложных, при эксплуатации долгоживущих Erlang-узлов (часто возникающая, когда время жизни узла достигает многих месяцев), и вы будете встречать её относительно нечасто.

Общими симптомами фрагментации являются большие всплески распределения памяти операционной системой во время пиковой нагрузки, и то, что после спадания нагрузки память не торопится возвращаться. Смущающим фактором будет то, что внутренние отчёты виртуальной машины по памяти (`erlang:memory()`) будут показывать намного меньшие числа.

7.3.1 Поиски фрагментации

Модуль `recon_alloc` был разработан с целью помочь обнаружить и дать подсказку к решению таких проблем.

Зная насколько редко встречается этот тип проблем среди участников сообщества (или случался, когда разработчики не знали, что происходит), определены только широкие шаги поиска проблем. Они довольно общие и требуют оценки человеком.

Проверьте выделенную память

Вызов функции `recon_alloc:memory/1` сообщит ряд метрик памяти в более удобном формате, чем `erlang:memory/0`. Вот варианты подходящих аргументов:

1. вызовите `recon_alloc:memory(usage)`. Это вернёт значение от 0 до 1, представляющее процент активно используемой Erlang-термами памяти в сравнении с памятью, которая была взята на эти нужды у операционной системы. Если значение близко к 100%, то, вероятно, у вас нет проблемы фрагментации, вы просто используете много памяти.

2. проверьте, совпадает ли результат `recon_alloc:memory(allocated)` с числом, которое возвращает операционная система[12]. Если число близко совпало, то проблема вероятно в фрагментации или утечке памяти Erlang-термов.

Это должно помочь укрепиться в мысли, фрагментирована ли ваша память или нет.

Поиск виновного аллокатора

Вызовите функцию `recon_alloc:memory(allocated_types)` чтобы увидеть, какой тип вспомогательного аллокатора (смотрите секцию 7.3.2 «Модель памяти в Erlang») выделяет больше всего памяти. Проверьте, если одно из чисел выглядит очевидным виновником, когда вы сравниваете результаты с `erlang:memory()`.

Попробуйте вызвать `recon_alloc:fragmentation(current)`. Распечатка данных, полученная в качестве результата, покажет различные аллокаторы на вашем узле и уровни использования ими памяти[13].

Если вы видите небольшие числа, то проверьте отличаются ли они, если вызвать `recon_alloc:fragmentation(max)`. Этот вызов покажет, каким было соотношение уровней при максимальной загрузке вашей памяти.

Если вы видите большое различие, то вероятно у вас имеется проблема фрагментации для нескольких типов аллокаторов при всплеске нагрузки на узел.

7.3.2 Модель памяти в Erlang

Взгляд снаружи

Чтобы понять, куда уходит память, следует сначала разобраться с множеством используемых аллокаторов. Модель памяти в Erlang, во всей виртуальной машине, организована иерархически. Как показано на рисунке 7.1, имеется два основных аллокатора, которые используются из Erlang-кода и виртуальной машины для почти всех типов данных[14]:

1. `temp_alloc`: выполняет временные распределения памяти для коротких задач (например, данные, живущие внутри одной C-функции).
2. `eheap_alloc`: данные куч, используются для таких вещей, как кучи процессов в Erlang.

[12] Можно вызвать `recon_alloc:set_unit(Единица)`, чтобы задать единицу измерения для значений, возвращаемых из `recon_alloc` в байтах (bytes), килобайтах (kilobytes), мегабайтах (megabytes), или гигабайтах (gigabytes)

[13] Дополнительная информация http://ferd.github.io/recon/recon_alloc.html

[14] Полный список мест, где живут разные типы данных, можно найти здесь: erts/emulator/beam/erl_alloc.types

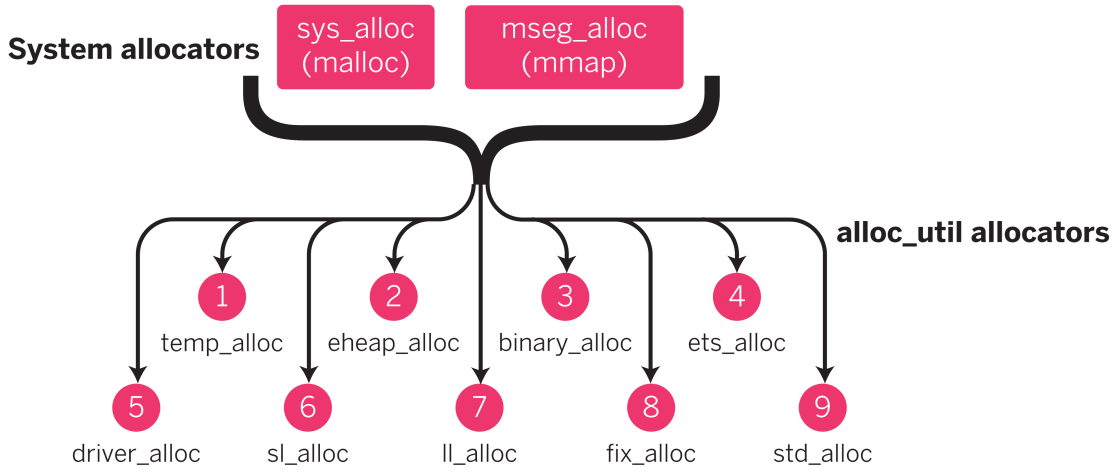


Figure 7.1: Аллокаторы памяти в Erlang и их иерархия. Не показаны особые *супер-носители* (*super carrier*), которые позволяют заранее распределять и ограничивать всю доступную память начиная с версии R16B03.

3. `binary_alloc`: аллокатор используется для хранения двоичных данных, освобождаемых по счётчику ссылок (это их «глобальная куча»). Двоичные данные, хранящиеся в ETS-таблице тоже используют этот аллокатор.
4. `ets_alloc`: ETS-таблицы хранят свои данные в отдельной части памяти, которая не обслуживается сборщиком мусора, но живёт пока термы хранятся в таблице.
5. `driver_alloc`: используется в частности для хранения данных драйверами, что впрочем не мешает им создавать Erlang-термы в других аллокаторах. Данные, выделенные здесь драйверами, могут содержать замки/мьютексы, настройки, порты Erlang и так далее.
6. `sl_alloc`: короткоживущие (*short-lived*) блоки памяти попадают сюда и включают такие элементы, как информация планировщика виртуальной машины или маленькие буферы для нужд обработки некоторых типов данных.
7. `ll_alloc`: долгоживущие (*long-lived*) распределения памяти идут сюда. Примерами является сам Erlang-код и таблица атомов, которые остаются надолго.
8. `fix_alloc`: аллокатор используется для частого распределения блоков одинакового размера. Примером данных может служить внутренняя структура описания процесса, которая используется виртуальной машиной.
9. `std_alloc`: стандартный аллокатор на все случаи жизни, используется, если предыдущие категории аллокаторов не подошли. Реестр для именованных процессов хранится здесь.

По умолчанию изначально имеется одна копия каждого аллокатора на каждый планировщик (и вам следует иметь один планировщик для каждого ядра процессора), плюс одна копия для встроенных драйверов, использующих асинхронные потоки (*async threads*). Это даёт вам

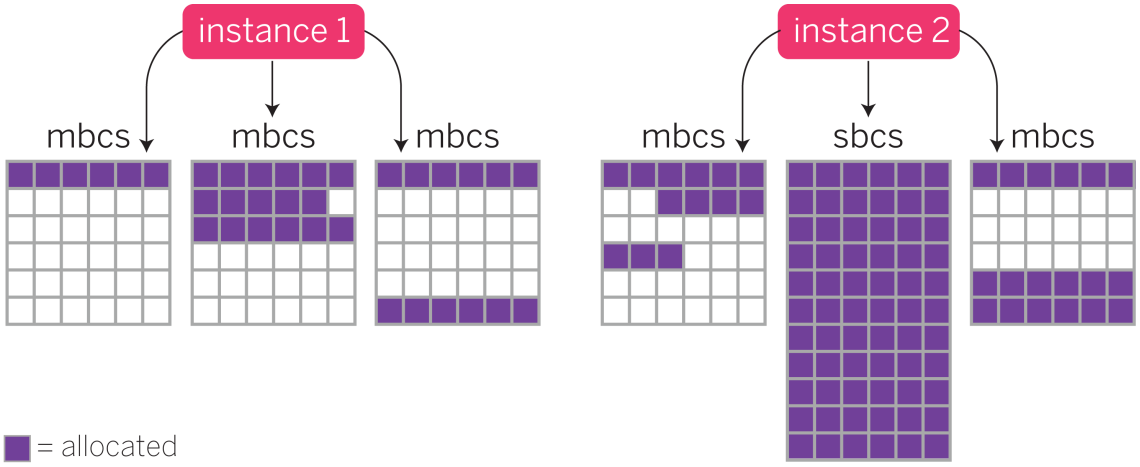


Figure 7.2: Пример памяти, выделенной некоторым вложенным аллокатором

структуру подобную той, что показана на рисунке 7.1, но каждый лист дерева разделяется на `N` частей.

Каждый из этих вложенных аллокаторов будет запрашивать память через `mseg_alloc` либо через `sys_alloc` в зависимости от цели использования, и двумя возможными способами. Первый способ — это действовать в качестве мультиблочного носителя (*multiblock carrier*, или `mbc`), который будет выделять блоки памяти для множества Erlang-термов одновременно. Для каждого `mbc` виртуальная машина зарезервирует заданное количество памяти (в нашем случае по умолчанию это будет 8 мегабайт, что можно регулировать в опциях виртуальной машины), и каждый терм при выделении своей памяти волен пройтись по множеству мультиблочных носителей в поиске достаточно большого свободного места, в котором можно поселиться.

Когда требуется выделить блок памяти размером больше, чем некоторый порог (*threshold* или `sbct`) для одноблочных носителей[15], аллокатор переключит режим распределения в одноблочный носитель (*single block carrier*, или `sbc`). Такой носитель затребует память непосредственно через `mseg_alloc` для первых `mmsbc`[16] попыток, а затем переключится на `sys_alloc` и будет хранить терм там до тех пор, пока он не освобождается.

Итак, глядя на нечто такое, как двоичный аллокатор, мы можем получить картинку, похожую на рисунок 7.2.

Когда мультиблочный носитель (или первые `mmsbc`[17] одноблочных носителей) могут быть возвращены операционной системе, `mseg_alloc` попытается удержать их в памяти в течение некоторого времени на случай, если возникнет всплеск запросов выделения памяти, чтобы можно было взять готовые блоки, а не требовать создания новых.

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

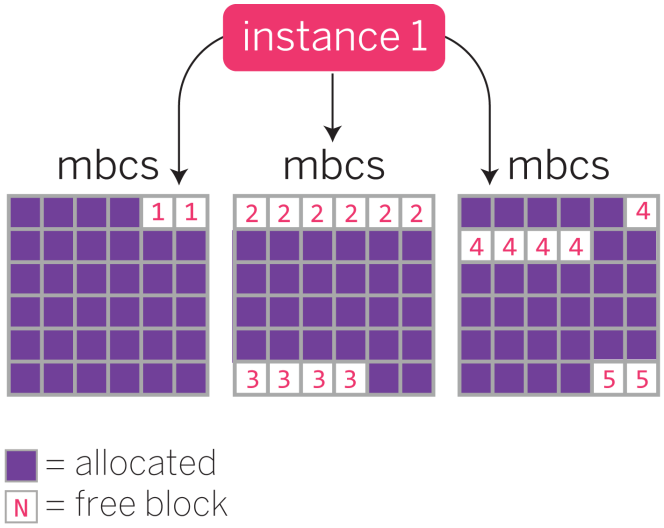


Figure 7.3: Пример памяти, выделенной некоторым вложенным аллокатором

Затем вам нужно узнать о различных стратегиях выделения памяти, которые есть в виртуальной машине Erlang:

1. Best fit (`bf`) — лучший подходящий размер.
2. Address order best fit (`aobf`) — лучший подходящий в порядке адресов.
3. Address order first fit (`aoff`) — первый подходящий в порядке адресов.
4. Address order first fit carrier best fit (`aoffcbf`) — первый подходящий в порядке адресов, лучший подходящий носитель.
5. Address order first fit carrier address order best fit (`aoffcaobf`) — первый подходящий в порядке адресов, лучший подходящий носитель в порядке адресов.
6. Good fit (`gf`) — достаточно хорошо подходящий.
7. A fit (`af`) – какой попало подходящий.

Каждая из этих стратегий может быть настроена индивидуально для каждого аллокатора `alloc_util`[18].

Для стратегии *best fit* (`bf`) виртуальная машина строит сбалансированное дерево, содержащее размеры всех свободных блоков и пытается найти наименьший подходящий свободный интервал и разместить блок в нём. На рисунке 7.3 при необходимости выделить блок данных длиной три клеточки, вероятнее всего он попадёт в область 3.

Стратегия *Address order best fit* (`aobf`) будет работать подобным образом, но вместо этого дерево будет построено на основе адресов блоков. Так что виртуальная машина будет искать наименьший доступный блок, который может принять заданный размер данных, но если

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

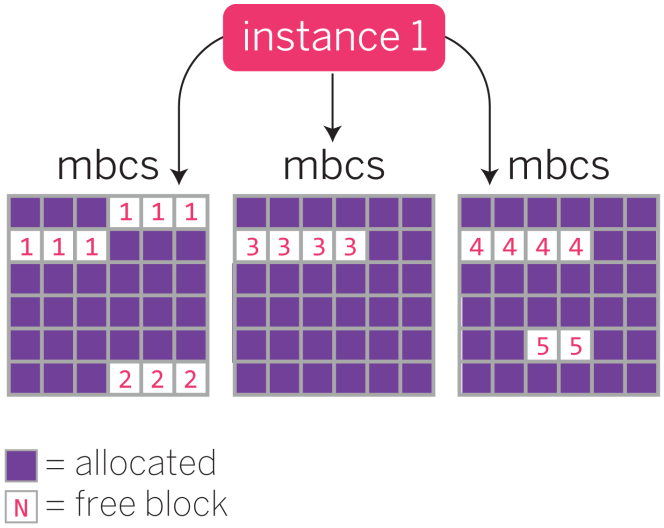


Figure 7.4: Пример памяти, выделенной некоторым вложенным аллокатором

существуют несколько вариантов выбора, то будет предпочитать тот, у которого адрес меньше. Если у меня есть фрагмент данных, который требует два блока, то эта стратегия предпочтёт первый мультиноситель (`mbcs`) на рисунке 7.3 с номером 1 (вместо зоны под номером 5). Это даёт виртуальной машине склонность предпочитать одни и те же носители для множества распределений памяти.

Address order first fit (`aoff`) будет руководствоваться порядком адресов при поиске, но как только найден первый подходящий блок, `aoff` его использует. В случае, когда стратегии `aobf` и `bf` использовали бы зону 3 для выделения четырёх блоков на рисунке 7.3, эта стратегия выбрала бы зону 2, поскольку её адрес меньше. На рисунке 7.4, если бы нам понадобилось выделить четыре блока, мы бы предпочли блок 1 блоку 3, поскольку его адрес меньше, тогда как `bf` выбрал бы вариант 3 или 4, а `aobf` выбрал бы 3.

Address order first fit carrier best fit (`aoffcbf`) — это стратегия, отдающая предпочтение носителю, который сможет принять нужный размер блока и затем в выбранном носителе выполняется поиск по лучшему подходящему размеру. Итак, если бы нам нужно было выделить два блока на рисунке 7.4, то стратегии `bf` и `aobf` отдали бы предпочтение зоне 5, `aoff` выбрала бы блок 1. `aoffcbf` выбрала бы зону 2, поскольку первый мультиноситель (`mbcs`) может принять его, и зона 2 подходит лучше зоны 1.

Address order first fit carrier address order best fit (`aoffcaobf`) ведёт себя подобно `aoffcbf`, но если внутри носитепля есть множество зон одинакового размера, он отдаст предпочтение той, у которой адрес наименьший вместо того, чтоб выбирать по какой-то своей неназванной логике.

Good fit (`gf`) — попытается сработать аналогично *best fit* (`bf`), но будет выполнять поиск только в течение ограниченного времени. Если не удалось за это время найти хорошо подходящий блок,

тогда он выберет лучший найденный до сих пор. Значение можно настроить параметром `mbsd`[19] виртуальной машины.

И наконец, *A fit* (`af`), поведение аллокатора для временных данных, которое ищет единственный блок памяти, в который могут поместиться данные, и использует его. Если данные не помещаются — то `af` выделит новый.

Каждая из этих стратегий может быть назначена индивидуально к каждому типу аллокаторов, так что аллокаторы кучи и двоичных данных могут вполне иметь разные стратегии.

Начиная с версии Erlang 17.0, каждый аллокатор `alloc_util` на каждом планировщике имеет некий так называемый пул мультиблочных носителей `mbcs` *pool*. Это возможность, используемая для борьбы с фрагментацией памяти на виртуальной машине. Когда один из мультиблочных носителей аллокатора почти опустошается,[20], то носитель становится *брошенным*.

Такой брошенный носитель прекращает использоваться для новых распределений памяти до тех пор, пока не начнут требоваться новые мультиблочные носители. Когда это случается, носитель извлекается из пула `mbcs`. Это может произойти в разных аллокаторах `alloc_util` одного и того же типа, на любом планировщике задач. Это позволяет виртуальной машине сохранить почти пустые носители не заставляя их освободить остатки памяти[21]. Также это позволяет миграцию носителей, содержащих немного данных, между планировщиками по необходимости.

Взгляд на уровне процесса

В меньшем масштабе, для каждого процесса Erlang, устройство памяти немного отличается. Можно представить себе некий контейнер, являющийся блоком в памяти, изобразим его так:

```
[                    ]
```

В одном конце его находится куча, а в другом — стек:

```
[куча |      | стек]
```

На практике данных больше (есть старая куча, и новая куча, для сборщика мусора, работающего с поколениями памяти, а также куча для двоичных данных, чтобы с их помощью считать ссылки на большие двоичные данные в большой куче, отдельно от процесса — `binary_alloc` против `eheap_alloc`):

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

[20] Порог срабатывания можно настроить http://www.erlang.org/doc/man/erts_alloc.html#M_acul

[21] Если оказывается так, что это приводит к большому расходу памяти, возможность можно отключить опцией `+MBacul 0`.

```
[куча   ||     стек]
```

Место продолжает выделяться с двух концов до тех пор, пока стек или куча не перестают помещаться. Это включает малый цикл сборки мусора. Эта сборка перемещает данные, которые продолжают существовать, в старую кучу. Затем сборщик собирает то, что осталось и, возможно, выделит новый блок памяти большего размера.

После заданного количества малых сборок мусора и перевыделений новой памяти, выполняется полный цикл сборки мусора (*full-sweep*), который анализирует новую и старую кучи, освобождает дополнительное место и так далее. Когда процесс умирает, стек и куча одновременно освобождаются, уменьшаются счётчики на двоичных данных, которые хранятся по количеству ссылок, и если они достигают нуля, то данные тоже освобождаются.

Когда это происходит, то в 80% случаях память оказывается отмеченной во вложенном аллокаторе, как свободная, и может быть взята новыми или другими процессами, которым потребовалось изменение размера. Только после того, как эта память не была использована в течение некоторого времени — и мультиблочный носитель не был использован тоже — память будет возвращена системному аллокатору, откуда она была взята (`mseg_alloc` или `sys_alloc`), который на своё усмотрение может удержать её ещё некоторое время.

7.3.3 Боремся с фрагментацией, изменяя стратегию распределения памяти

Можно попробовать настроить параметры виртуальной машины, отвечающие за распределение памяти.

Вероятно вам понадобится хорошее понимание вашей ситуации с нагрузкой на память и её использованием, и будьте готовы провести множество тестов для выяснения этих фактов. Модуль `recon_alloc` содержит несколько вспомогательных функций, которые направят ваш путь, и это будет самым подходящим временем прочесть документацию к модулю[22].

Вам нужно будет выяснить средний размер ваших данных в памяти, частоту распределений и освобождений, попадают ли ваши данные в мультиблоки (`mbcs`) или в одиночные блоки (`sbcs`), и затем попробуйте поиграть с множеством опций, которые упомянуты в модуле `recon_alloc`. Попробуйте разные стратегии, установите их, сравните ситуацию — улучшилась ли она или всё стало медленнее и хуже.

Это долгий кропотливый процесс для которого нет известного короткого пути, и если проблемы случаются каждые несколько месяцев, то будьте готовы к тому, что поиски затянутся.

7.4 Упражнения

Вопросы для закрепления материала

1. Назовите несколько известных вам источников утечек ресурсов в Erlang-программах.

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

2. Какие два вида блоков двоичных данных имеются в Erlang?
3. Где следует искать, если никакой из типов данных не выглядит источником утечки?
4. Если вы обнаружили смерть узла, в котором один из процессов занял очень много памяти, что вы можете сделать для поиска виновного процесса?
5. Как сам код может явиться причиной утечки?
6. Как можно выяснить, что сборка мусора начала занимать очень долгое время?

Открытые вопросы

1. Как можно проверить, является ли причиной утечки то, что вы забыли убить ненужные процессы или они просто сами начали использовать слишком много памяти?
2. Некий процесс открывает 150-мегабайтовый файл журнала в режиме двоичного чтения для извлечения некоторой информации, затем сохраняет информацию в ETS-таблице. После выяснения наличия у вас утечки двоичной памяти, что можно сделать, чтобы минимизировать расход двоичной памяти на данном узле?
3. Что можно использовать для выяснения того факла, что ваши ETS-таблицы растут слишком быстро?
4. Какие шаги вам следует пройти, чтобы узнать, что ваш узел вероятно страдает от фрагментации памяти? Как можно опровергнуть догадку, что виновником мог являться драйвер или встроенная (NIF) функция?
5. Как можно выяснить, что процесс с большим почтовым ящиком (посредством чтения параметра `message_queue_len`) вероятно имеет утечку памяти из ящика или не обслуживает новые сообщения?
6. Процесс с большим потреблением памяти вероятно редко выполняет сборку мусора. Чем можно это объяснить?
7. Когда вам следует изменять стратегии распределения памяти на ваших узлах? Предпочтёте ли вы настраивать стратегии или исправить ваш код?

Тренировочные задания

1. Используя любую систему, которая вам знакома или которую вам приходится обслуживать, написанную на Erlang (включая игрушечные системы для незначительных проектов), сможете ли вы выяснить, имеет ли эта система утечки двоичной памяти?

Chapter 8

Перерасход процессора и занятость планировщиков

В то время, как утечки памяти могут абсолютно легко убить вашу систему, истощение ресурса процессора действует в качестве узкого места и ограничивает количество работы, которую вы можете получить от узла. Erlang-разработчики при возникновении таких проблем предпочитают горизонтальное масштабирование. Часто достаточно несложно вынести и отмасштабировать несколько простых блоков кода. Обычно для этого следует изменить глобальное состояние (реестры процессов, ETS-таблицы и так далее)[1]. Если вы желаете оптимизировать систему локально перед тем, как начать масштабироваться, вам нужно будет найти причины перерасхода процессора и планировщиков.

Обычно достаточно трудно провести качественный анализ расхода процессора на Erlang-узле, чтобы отследить источник проблем с точностью до фрагмента кода. Когда всё выполняется одновременно и на виртуальной машине, нет гарантий того, что вы найдёте, кто съел весь процессор — конкретный процесс, драйвер, ваш собственный код, установленные вами встроенные (NIF) функции или какая-то дополнительная библиотека.

Существующие подходы обычно ограничиваются профилированием и подсчётом редукций, если проблема в вашем коде, либо мониторингом работы планировщика, если причина может быть где-то в другом месте (но тоже в вашем коде).

8.1 Профилирование и подсчёт редукций

Чтобы отследить причины проблем до конкретных мест в Erlang-коде, как упоминалось ранее, существуют два главных подхода. Один способ — выполнить старую стандартную процедуру

[1] Обычно это принимает форму фрагментирования (*sharding*) или поиска более-менее подходящей схемы репликации состояния, ну и ещё кое что. Это является существенным куском работы, но совсем немного по сравнению с затратами на выяснение того факта, что семантика вашей программы не подходит для распределённых систем, хотя обычно Erlang заставляет вас заняться именно этим с самого начала.

профилирования, вероятно с использованием следующих приложений[2]:

- `eprof`,[3] самый старый из известных профайлеров для Erlang. Он даёт общие значения в процентах и обычно расчёт ведётся на основании затраченного времени.
- `fprof`,[4] является более мощной заменой для eprof. Он поддерживает параллельное исполнение и строит подробные отчёты. На самом деле, отчёты настолько трудно читать, что их можно считать чёрным ящиком.
- `eflame`,[5] — один из новейших продуктов. Он создаёт огненные графики (*flame graphs*) для демонстрации глубоко идущих вызовов функций и подсвечивает нагруженные участки в заданном коде. Позволяет быстро найти проблемы по одному взгляду на окончательный результат.

Задачу прочитать документацию по этим приложениям я оставляю на совести читателя. Второй подход предполагает, что вы выполните функцию `recon:proc_window/3`, которая представлена в секции 5.2.1 «Копаем в процессы»:

```
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}}]}]
```

Счёт редукций прямо связан с вызовами функций в Erlang, и большое значение счётчика обычно является синонимом высокого потребления процессорной мощности.

Что интересно в этой функции это то, что её можно запустить, когда система довольно нагружена[6], с относительно коротким интервалом. Повторите этот шаг много раз и вы, вероятно, заметите появление похожих шаблонов в цифрах, когда одни и те же процессы (или один *mun* процессов) часто оказываются на первых местах.

Используя функции поиска местонахождения в коде[7] и информацию о текущих исполняющихся функциях, вы сможете определить, какой код занял все ваши планировщики.

[2] Все они используют встроенную в Erlang трассировку практически без ограничений. Они могут ощутимо ударить по производительности вашего приложения и их не следует использовать на производственных системах.

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

[4] http://www.erlang.org/doc/man/fprof.html

[5] https://github.com/proger/eflame

[6] Смотрите подсекцию 5.1.2 «Утилизация процессора»

[7] Вызовите `recon:info(PidTerm, location).` или `process_info(Pid, current_stacktrace)`, чтобы получить эту информацию.

8.2 Системные мониторы

Если профилирование или проверка счётчиков редукций не нашла выдающихся цифр, вероятно часть неучтённой работы выполняется во встроенных (NIF) функциях, в сборке мусора и так далее. Такие виды работы не всегда увеличивают счётчики редукций, так что они могут оказаться невидимыми при поиске предыдущими методами, и проявляют себя только в виде долгого времени исполнения.

Для поиска таких случаев лучшим способом является функция `erlang:system_monitor/2`, а именно опции `long_gc` и `long_schedule`. Первое покажет, когда сборка мусора выполняет большую работу (это занимает время!), а второе вероятно поймает проблемы с занятыми процессами по причине, например, выполнения встроенных (NIF) функций, или чего-то другого, что мешает им правильно отдать управление планировщику[8].

Мы видели, как установить такой системный монитор в секции 7.1.5 «Утечки сборщика мусора», но есть другой шаблон поиска решения[9], который я использовал для поиска долго исполняющихся фрагментов кода:

```
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}}]
```

Не забудьте установить параметры `long_schedule` и `long_gc` в некоторые достаточно высокие значения, которые для вас выглядят разумными. В этом примере я установил их в 1000

[8] Длинные сборки мусора включаются в расход времени планировщиком. Очень вероятно, что долгие переключения планировщика связаны со сборками мусора, в зависимости от вашей системы.

[9] Если вы пользуетесь Erlang версии 17.0 или более новым, функции в консоли интерпретатора можно делать рекурсивными гораздо проще, используя их именованную запись, но для совместимости со всеми версиями, я оставил их без изменений.

миллисекунд. Вы можете убить монитор командой `exit(whereis(temp_sys_monitor), kill)` (что также убьёт и интерпретатор, поскольку они связаны), или просто отключитесь от узла (что убьёт процесс, поскольку он связан с интерпретатором).

Такой вид кода и мониторинга можно перенести в собственный модуль, где он будет собирать статистику в некоторое долгоживущее хранилище, на основе которой потом можно выяснить о падении производительности или определять перегрузку.

8.2.1 Замороженные порты

Интересная часть системных мониторов, для которой я не нашёл места нигде в книге, связана с работой планировщика с портами. Когда процесс отправляет в порт слишком много сообщений, и внутренняя его очередь переполняется, планировщики Erlang принудительно снимут отправителя с очереди планировщика и переключат в спящий режим до тех пор, пока место для отправки данных не освободится. Это может оказаться сюрпризом для некоторых пользователей, не ожидавших такого обратного давления со стороны виртуальной машины.

Такое событие можно обнаружить монитором, передавая ему атом `busy_port`. Для узлов в составе кластера можно использовать атом `busy_dist_port`, это найдёт локальные процессы, которые были сняты с очереди планировщика и отправлены в спящий режим при попытке связаться с другим процессом на другом узле, чья связь между узлами оказалась заблокирована слишком занятым портом.

Если вы обнаружили, что у вас возникли проблемы, попробуйте заменить ваши функции отправки в критических фрагментах кода командой `erlang:port_command(Порт, Данные, [nosuspend])` для портов и `erlang:send(Pid, Данные, [nosuspend])` для сообщений в направлении процессов на других узлах. Они дадут вам знать, когда не смогут отправить сообщение, и таким образом ваш процесс будет снят с очереди планировщика и заморожен.

8.3 Упражнения

Вопросы для закрепления материала

1. Какие есть два основных подхода к поиску проблем с расходом процессора?
2. Назовите некоторые известные вам инструменты для профилирования. Какие подходы предпочтительнее использовать на производстве (*production*)? Почему?
3. Почему мониторы, срабатывающие на долгую работу планировщика, могут пригодиться к поиску перерасхода процессора или планировщиков?

Открытые вопросы

1. Если вы обнаружите, что процесс выполняет очень мало работы согласно счётчику редукций, и планировщик выделяет ему длительные периоды времени, какие можно сделать выводы о коде, который там исполняется?

2. Можно ли запустить системный монитор и заставить его сработать обычным кодом на Erlang? Можно ли с его помощью найти средний интервал, выдаваемый процессам планировщиком? Вам может быть понадобится запускать вручную из интерпретатора случайные процессы, которые будут выполнять некоторую работу более агрессивно чем те, что имеются в системе сейчас.

Chapter 9

Трассировка

Одна из менее известных и совершенно недостатчно популярных возможностей Erlang и виртуальной машины BEAM — это то, сколько видов трассировки можно в ней выполнить.

Забудьте о ваших отладчиках, они слишком ограничены[1]. Выполнение трассировки в Erlang имеет смысл на всех этапах жизни вашей системы, независимо от того, является ли эта система вашей тестовой, или работающей производственной системой.

Для трассировки Erlang-программ имеется несколько подходов:

- Модуль `sys`[2] идёт в стандартной поставке OTP и позволяет пользователю установить собственные функции для приёма событий трассировки, журналировать все события и так далее. Он в целом является полнофункциональным и подходит для разработки. Не совсем удачным является его использование на производственной системе, поскольку он не перенаправляет вывод на удалённые терминалы и не ограничивает скорость отправки сообщений трассировки. Если вы даже не будете его использовать, обязательно ознакомьтесь с документацией по этому модулю.

- Модуль `dbg`[3] также входит в поставку Erlang/OTP. Его интерфейс немного неуклюж в плане удобства использования, но он совершенно подходит для ваших задач. Проблема с ним в том, что вы *должны знать, что вы делаете*, поскольку `dbg` может журналировать абсолютно всё на узле и убить этой нагрузкой узел за пару секунд.

[1] Одна из общих проблем отладчиков это то, что возможность установки точек останова и выполнять шаги в программе несовместима со многими программами на Erlang: останови один процесс, а остальные продолжат работу. На практике это превращает отладку в очень ограниченную задачу, поскольку если процесс начинает взаимодействовать с другими во время отладки, начинают также срабатывать таймауты и процессы падают, вероятно лечь может и весь узел целиком. С другой стороны трассировка не вмешивается в работу программы, но даёт вам нужные данные.

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

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

- *Трассирующие встроенные BIF-функции* доступны в составе модуля `erlang`. Они в основном являются кирпичиками, из которых построены все приложения в этом списке, но их более низкий уровень абстракции делает их довольно трудными в использовании.
- Библиотека `redbug`[4] — разработана с учётом безопасности использования на производственных серверах, входит в состав набора инструментов `eper`[5]. Имеет встроенный ограничитель скорости подачи событий, и хороший удобный интерфейс. Для использования её, однако, вам придётся добавить её в список зависимостей вашего проекта. Набор инструментов в ней, кроме redbug, довольно обширен и может оказаться полезным.
- `recon_trace`[6] — это попытка `recon` подойти к проблеме трассирования. Целью создания его была возможность предложить те же уровни безопасности, что и `redbug`, но не добавляя зависимостей. Интерфейс отличается, и возможности ограничения скорости подачи сообщения не совсем идентичны. Он также может трассировать только вызовы функций, но не сообщения[7]

Эта глава сосредоточится на трассировке с помощью `recon_trace`, но терминология и концепция в целом переносятся и на другие инструменты трассировки, которые вы решите использовать.

9.1 Принципы трассировки

Встроенные (BIF) функции трассирования позволяют проследить работу совсем любого Erlang-кода[8]. Их работа состоит из двух частей: *спецификации идентификаторов процессов (pid)*, и *образцы трассировки*.

Спецификации pid позволяют пользователю решить, какой процесс его интересует. Они могут быть как конкретными значениями идентификаторов процессов, специальным атомом `all` (все процессы), `existing` (существующие процессы), или `new` (новые, созданные во время вызова функции).

Образцы трассировки представляют собой функции. Функции можно указывать двумя частями: задавая модули, имена функций и арность, либо с помощью стандартных спецификаций сопоставления Erlang[9]. Это позволяет добавить к аргументам условия и ограничения.

---

[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] Сообщения могут поддерживаться в будущих версиях библиотеки. На практике автор не нашёл потребности в этом, поскольку при использовании OTP, поведений и сопоставления с аргументами, можно получить примерно тот же эффект.

[8] В тех случаях, когда процессы содержат секретную информацию, данные можно оставить скрытыми с помощью вызова `process_flag(sensitive, true)`

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

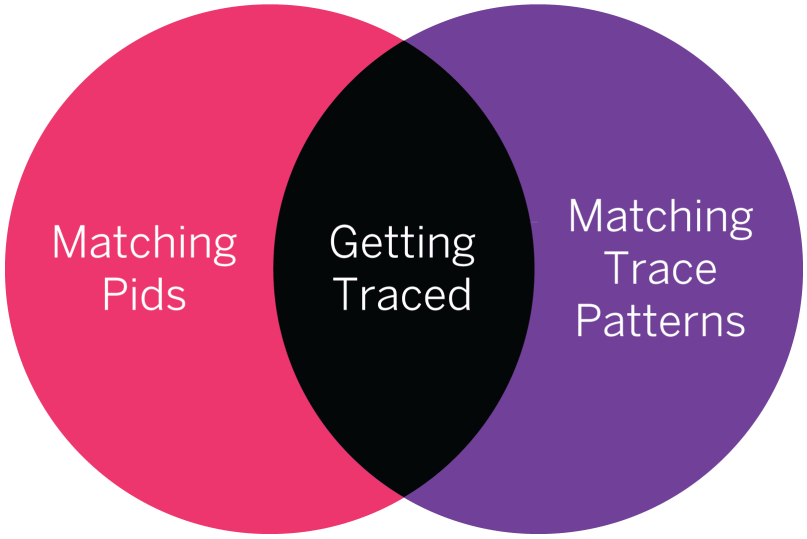


Figure 9.1: Что будет трассироваться выбирается из пересечения множества совпадающих процессов и подходящих образцов трассировки (matching trace patterns)

Решение, принадлежит ли некоторая функция к набору трассируемых, принимается по пересечению двух множеств, как видно на рисунке 9.1.

Если хотя бы одна из спецификаций pid исключает некоторый процесс, либо шаблон трассировки исключает некоторый вызов, то события этой функции не будут получены.

Инструменты вроде `dbg` (и встроенных трассировочных BIF-функций) заставляют вас помнить эту диаграмму при работе. Вы указываете наборы совпадающих идентификаторов процессов и наборы образцов трассировки независимо друг от друга, и то, что находится на пересечении обеих множеств — вы получаете в качестве результата.

С другой стороны инструменты вроде `redbug` и `recon_trace` стараются скрыть это от пользователя.

9.2 Трассируем с помощью Recon

По умолчанию `recon` будет смотреть за всеми процессами. Это часто является вполне приемлемым вариантом для множества случаев отладки. Чаще всего вас будет интересовать работа с образцами трассировки, и `recon` поддерживает несколько способов их объявления.

Основная базовая форма записи — выражение `{Модуль, Функция, Арность}`, где `Модуль` — это имя модуля, `Функция` — имя функции и `Арность` — число аргументов для точного выбора функции, которую мы собрались трассировать. Любой из этих параметров можно заменить подстановочным символом (`'_'`). Recon запрещает формы, которые слишком широко совпадут

с большим количеством функций, например {'_','_','_'}), поскольку будет очень опасно запускать такую трассировку на производстве.

Более сложная форма записи позволяет заменить параметр арности функцией, которая будет сопоставлять списки аргументов с образцами. Функция ограничена спецификациями, подобными тем, что доступны в ETS[10]. И наконец множество образцов можно поместить в список и расширить таким образом поиск.

Также имеется возможность ограничить скорость подачи событий двумя способами: ограничением количества или числа совпадений в интервал времени.

Вместо того, чтобы углубляться в скучные детали, ниже показан ряд примеров и как для них выполнить трассировку.

```
%% Все вызовы в модуле 'queue', печатаем не более 10 первых вызовов:
recon_trace:calls({queue, '_', '_'}, 10)

%% Все вызовы к lists:seq(A,B) и печатаем не более 100 вызовов:
recon_trace:calls({lists, seq, 2}, 100)

%% Все вызовы к lists:seq(A,B) и печатаем не более 100 вызовов в секунду:
recon_trace:calls({lists, seq, 2}, {100, 1000})

%% Все вызовы к lists:seq(A,B,2) (последовательности с шагом два) и не больше
%% 100 вызовов в сумме:
recon_trace:calls({lists, seq, fun([_,_,2]) -> ok end}, 100)

%% Все вызовы iolist_to_binary/1 выполненные когда аргумент уже является
%% двоичными данными (отслеживаем бесполезные приведения типов):
recon_trace:calls({erlang, iolist_to_binary,
                   fun([X]) when is_binary(X) -> ok end},
                  10)

%% Вызовы к модулю 'queue' только в заданном процессе Pid, со скоростью не чаще
%% 50 в секунду:
recon_trace:calls({queue, '_', '_'}, {50,1000}, [{pid, Pid}])

%% Печатать сообщения трассировки с указанием арности, вместо значений аргументов:
recon_trace:calls(TSpec, Max, [{args, arity}])

%% Совпадёт с функциями filter/2 в модулях dict и lists, только среди новых процессов:
recon_trace:calls([{dict,filter,2},{lists,filter,2}], 10, [{pid, new}])

%% Трассируем функции handle_call/3 заданного модуля для новых процессов, и тех
%% существующих, которые были зарегистрированы в gproc:
```

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

```
recon_trace:calls({Mod,handle_call,3}, {1,100}, [{pid, [{via, gproc, Name}, new]}

%% Показать результат данного вызова функции, важным моментом является вызов
%% return_trace() или значение {return_trace} в спецификации сопоставления.
recon_trace:calls({Mod,Fun,fun(_) -> return_trace() end}, Max, Opts)
recon_trace:calls({Mod,Fun,[{'_', [], [{return_trace}]}]}, Max, Opts)
```

Каждый выполненный вызов отменит действие предыдущего, и все вызовы можно отменить с помощью `recon_trace:clear/0`.

Возможно ещё несколько комбинаций, с дополнительными опциями:

`{pid, СпецификацияPid}`

Какие процессы трассировать. Допустимыми опциями является комбинация любых значений из такого набора: `all`, `new`, `existing`, или дескриптор процесса (`{A,B,C}`, `"<A.B.C>"`, атом, представляющий имя, `{global, Имя}`, `{via, Программа-Регистратор, Имя}`, или просто pid). Также возможно указать больше одного варианта, поместив их в список.

`{timestamp, formatter | trace}`

По умолчанию функция печати добавляет текущее время к полученным сообщениям. Если требуется более точное время, можно принудительно включить добавление времени внутри сообщений трассировки, добавив опцию `{timestamp, trace}`.

`{args, arity | args}`

Указывает что печатать в вызовах функций: арность функции (вывод получается более коротким) или их буквальные значения (по умолчанию включено, более длинный вывод).

`{scope, global | local}`

По уолчанию трассируются только глобальные (полностью заданные вызовы функций с именем модуля). Для трассировки локальных вызовов, передайте сюда выражение `{scope, local}`. Это полезно, когда вы отслеживаете изменения в коде в том процессе, который не был вызван полной формой (`Module:Fun(Args)`), а вместо этого был использован короткий вызов `Fun(Args)`.

С этими опциями имеется много способов сопоставить с образцом вызовы некоторых функций и всё такое, и можно диагностировать множество проблем на производстве гораздо быстрее. Если вы когда-нибудь задумываетесь «Хм, наверное надо вставить здесь io:format и посмотреть, почему этот код ведёт себя странно», то трассировка обычно является коротким путём к получению нужных вам данных без добавления нового кода или изменения его читаемости.

9.3 Примеры

Давайте попробуем трассировку функций queue:new во всех процессах:

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

Ограничение было установлено в 1 сообщение, и recon дал нам знать, когда лимит был достигнут.

Давайте вместо этого посмотрим на все вызовы `queue:in/2`, чтобы проверить что там вставляется в очереди:

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

Чтобы увидеть те данные, которые нам интересны, следует изменить образцы трассировки и использовать анонимную функцию, которая будет сопоставляться со всеми аргументами в списке (_) и возвратит `return_trace()`. Последняя часть создаст второе сообщение для каждого события трассировки, в котором будет возвращаемое значение:

```
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.
```

Сопоставление со списком аргументов можно сделать более сложным:

```
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
```

Заметьте, что в шаблоне, показанном выше, событие не сопоставляется ни с какой конкретной функцией. Вместо этого используются охранные значения, фильтрующие функции с двумя аргументами, первый из которых является списком или целым числом больше единицы.

Будьте осторожны, слишком широкие шаблоны со слишком свободными ограничениями скорости (или большими абсолютными лимитами) могут повлиять на стабильность вашего узла так сильно, что даже `recon_trace` не очень-то поможет. Подобно этому, трассирование очень большого количества вызовов функций (всех подряд, или, например всех из модуля `io`) может оказаться рискованным делом, когда будет создаваться больше сообщений, чем сможет обработать ваш узел. Даже несмотря на меры предосторожности, принятые в библиотеке.

Если не уверены, начните с самых ограниченных правил трассировки с пониженными лимитами и постепенно расширяйте лимиты и поиск.

9.4 Упражнения

Вопросы для закрепления материала

1. Почему полезность отладчика в Erlang ограничена?
2. Какие возможности у вас имеются для трассировки процессов OTP?
3. Какие правила определяют, будет ли трассироваться некий данный набор функций или процессов?
4. Как можно прекратить трассировку с помощью `recon_trace`? А с помощью других инструментов?
5. Как можно выполнить трассировку вызовов функций, которые не были экспортированы?

Открытые вопросы

1. Когда вы можете захотеть перенести установку отметок о времени внутрь механизмов трассировки виртуальной машины? Какой возможный недостаток здесь кроется?

2. Представьте, что некие данные, идущие с узла, проходят внутри шифрования SSL через сложную систему с рядом узлов. Однако, поскольку вы желаете проверять отправленные данные (реагируете на жалобу одного из клиентов), вам нужна возможность изучить данные без шифрования в чистом виде. Можете ли вы придумать план, как заглянуть в отправленные данные через сокет модуля `ssl`, не подслушивая данные, отправленные любому другому клиенту кроме того, что пожаловался?

Тренировочные задания

Используя код, находящийся по адресу https://github.com/ferd/recon_demo (вам может потребоваться хорошо разобраться с этим кодом):

1. Могут ли разговорчивые процессы (`council_member`) писать самим себе? (*подсказка: может ли это сработать с зарегистрированными именами? Нужно ли вам найти самый болтливый процесс и проверить не пишет ли он сам себе?*)

2. Сможете ли вы глобально оценить общую частоту, с которой посылаются сообщения?

3. Сможете ли вы аварийно завершить работу узла используя трассировочные инструменты (*подсказка: это легче сделать с помощью dbg благодаря большой гибкости и отсутствию тормозов.*)

Выводы

Обслуживание и отладка приложений никогда не прекращаются. Новые ошибки и странное поведение продолжат появляться всё время и везде. Вероятно накопится ещё материала достаточно для десятка подобных книг, даже при работе с самыми чистыми и надёжными системами.

Я надеюсь, что после прочтения этого текста следующий раз, когда ваша система поведёт себя плохо, ваши дела окажутся не *слишком* плохи. Всё же у вас вероятно будет множество возможностей отладить производственные системы. Даже самые надёжные мосты нужно иногда подкрашивать, чтобы избежать коррозии и обрушения.

Удачи вам.