Simon St. Laurent

Введение в Erlang (второе издание)

Перевод на русский язык - Косоруков А.С.

Оригинал книги можно найти по ссылке
https://learning.oreilly.com/library/view/introducing-erlang-2nd/9781491973363/

ACKNOWLEDGEMENT:
The translation was made possible thanks to kind permission of O'Reilly Media

БЛАГОДАРНОСТЬ:
Выражаю искреннюю признательность издательству O'Reilly Media, которое предоставило мне право на перевод этой книги на русский язык.

Предисловие

Erlang долгое время был загадочным темным уголком вселенной программирования, который посещали в основном разработчики, которым нужна чрезвычайная надежность или масштабируемость, и люди, которые хотят напрячь свой мозг.

Разработанный в Ericsson для обслуживания телефонных коммутаторов, он казался странным языком специального назначения до недавнего времени, когда наши компьютерные и сетевые архитектуры стали больше походить на массивно параллельное телефонное коммутационное оборудование. Благодаря появлению хранилищ данных NoSQL таких как CouchDB и Riak вы, возможно, уже используете Erlang, даже не осознавая этого, и Erlang переходит во многие другие области.

Erlang обеспечивает короткий путь от дисциплины к устойчивости. Решения, принятые создателями языка, позволяют среде Erlang беспрепятственно масштабироваться и обрабатывать ошибки способами, которыми должны управлять другие среды, добавляя еще больше инфраструктуры. По моему предвзятому мнению, любое приложение, которое должно работать в течение длительного времени и масштабироваться для многих взаимодействий, должно быть построено в Erlang (или его более поздней кузине Elixir).

Изучение Erlang, если вы пришли из какого-либо другого языка, кроме функционального программирования, потребует, чтобы вы очистили свой ум от многих методов, используемых в других языках программирования. Забудьте классы, забудьте переменные, которые меняют значения – даже забудьте о соглашениях о назначении переменных.

Вместо этого вам придется подумать о сопоставлении с образцом, передаче сообщений и создании путей для данных, а не указывать, куда идти. Программирование на Erlang может быть похоже на создание ключа, чьи зубцы ставятся прямо в нужно положение или как играть в пинбол и смотреть, как шары летают по лабиринту.

Звучит странно, правда? На Erlang приятно программировать. Надеюсь, что вы оцените это.

Мои первые исследования Erlang смутили меня и одновременно взволновали. У меня был некоторый опыт работы с так называемыми «инвариантными переменными», переменными, которые могут быть связаны со значением только один раз, в XSLT. Это создавало мне много головной боли, пока я не осознал, что я неправильно понял проблему. И всё сразу стало на свои места.

Для кого эта книга

Эта книга в основном для людей, которые программируют на других языках, но хотят присмотреться к Erlang. Может быть, вы очень практичны, и распределенная модель Erlang, масштабируемость и устойчивость привлекают вас. Может быть, вы хотите увидеть, что такое «функциональное программирование». Или, может быть, вы просто отправляетесь в поход, перенося свой разум на новое место.

Я думаю, что функциональное программирование – наилучший выбор первой парадигмы программирования. Однако, чтобы начать работать с Erlang, иногда даже просто

установить его, вам может потребоваться немало компьютерных навыков. Если вы новичок в программировании, добро пожаловать, но предупреждаю, будет непросто.

Для кого эта книга не предназначена

Эта книга не для людей, которые спешат.

Если вы уже знаете Erlang, вам вряд ли понадобится эта книга, если только вы не хотите освежить свои знания.

Если вы уже знакомы с функциональными языками, вы может быть найдёте безнадежно медленным темп изложения материала. Обязательно попробуйте перейти к другой книге, которая будет двигаться быстрее, если вам покажется скучно. Возвращайтесь, если найдете, что другие идут слишком быстро, и не стесняйтесь использовать её в качестве сопроводительного руководства или в качестве ссылок на другие книги.

Что эта книга сделает для вас

В книге «Семь языков за семь недель» Брюс Тейт писал, что «Erlang делает сложные вещи легкими, а простые – сложными». Эта книга проведет вас через «легкие вещи» и покажет вам легкие пути к«сложным вещам».

Вы научитесь писать простые программы на Erlang. Вы поймете, почему Erlang облегчает создание гибких программ, которые можно легко масштабировать. Возможно, самое главное, вы подготовитесь к чтению материала других ресурсов по Erlang, которые предполагают достаточно опыта.

В более теоретическом плане вы познакомитесь с функциональным программированием. Вы узнаете, как разрабатывать программы, основанные на передаче сообщений и рекурсии, так и создавать программы ориентированные на процесс и поток данных.

Вы также будете лучше подготовлены к чтению других книг по Erlang и участии в беседах о нём.

Как работать с книгой

Эта книга пытается рассказать историю о Erlang. Вы, вероятно, получите максимальную отдачу от этого, если прочитаете его по порядку хотя бы в первый раз. Конечно, вы всегда можете вернуться и найти все, что вам нужно.

Вы начнете с установки и запуска Erlang и запуска его программной оболочки. Вы проведете много времени с ней. Затем вы начнете загружать код в оболочку, чтобы было легче писать программы, и вы узнаете, как вызывать этот код.

Вы внимательно изучите работу с числами, потому что они удобны для ознакомления с основными структурами Erlang. Затем вы узнаете об атомах, сопоставлении с образцом и охранных конструкциях – вероятных основах структуры вашей программы. После этого вы узнаете о строках, списках и рекурсии, которые лежат в основе работы Erlang. После того, как вы прошли несколько миллионов рекурсий туда и обратно, пришло время

взглянуть на процессы – ключевую часть Erlang, которая опирается на модель передачи сообщений для поддержки параллелизма и устойчивости.

Когда у вас есть базовый набор, вы можете поближе взглянуть на отладку и хранение данных, а затем быстро взглянуть на набор инструментов, который, вероятно, лежит в основе вашей долгосрочной разработки с Erlang: Open Telecom Platform (OTP), которая может быть применимы не только в телекоммуникационных системах.

Некоторые люди хотят изучать языки программирования, как словарь. Вот список операторов, вот список структур управления, это типы данных – и затем используйте всё это вместе. Всё это есть в Приложении А, а не в основной части книги.

Многие примеры построены на одних и тех же данных. Я думаю, что так, вы сможете лучше сосредоточится на концепциях, чем на особенностях примеров.

Главное, что вы должны вынести из этой книги – это то, что вы можете программировать на Erlang. Если вы этого не получите, дайте мне знать!

Этюды на Erlang

Пока я писал эту книгу, Дж. Дэвид Эйзенберг разрабатывал широкий набор упражнений, чтобы сопровождать ее. Они оказались достаточно комплексными, чтобы стать отдельным проектом, который вы можете найти (бесплатно в Интернете).

Этюды построены так, чтобы соответствовать этой книге, но, надеюсь, со временем они будут расширяться и охватывать более широкий круг тем, чем эта книга. Вы, вероятно, получите максимальную отдачу от них, если будете изучать их каждый раз, когда заканчиваете главу, но они также отлично подходят для общего обзора и проверки вашего понимания.

Почему я написал эту книгу

Я не эксперт по Erlang, надеющийся воспитать побольше экспертов по Erlang, чтобы завершить какой-то большой проект.

Я писатель и разработчик, который столкнулся с Erlang, поняв, что именно это язык программирования, искал в течение долгого времени, и чувствующий себя вынужденным поделиться своими знаниями. Я надеюсь, что путь, которым я следовал, будет работать для других, возможно, с вариациями, и что книга, написанная с точки зрения начинающего (и проверенная экспертами), поможет большему количеству людей насладиться работой с Erlang.

Другие источники

Эта книга, возможно, не лучший способ изучить Erlang. Все зависит от того, что вы хотите узнать и почему.

Если ваш основной интерес в изучении Erlang состоит в том, чтобы вырваться из рутины программирования, вы должны прочесть поверхностный тур Брюса Тейта «Семь языков

за семь недель» (Seven Languages in Seven Weeks (Pragmatic Publishers)), который исследует Ruby, Io, Prolog, Scala, Erlang, Clojure и Haskell. Erlang он отводит только 37 страниц, но это может быть то, что вам надо.

Чтобы получить опыт работы (в настоящее время также в печатном виде от No Starch Books) с более яркими и забавными иллюстрациями, вам следует изучить книгу Фреда Хеберта «Изучите Erlang во имя добра!» http://learnyousomeerlang.com/. Хотя она гораздо подробнее расскрывает возможности языка, чем рассказывает Тейт, она, безусловно, больше походит на руководство для опытного программиста по Erlang.

Есть две классические книги по Erlang – это «Programming Erlang» (Pragmatic Publishers), написанную одним из создателей Erlang Джо Армстронгом, и «Erlang Programming» (O'Reilly) написанную Франческо Чезарини и Саймоном Томпсона. Они охватывают множество схожих и перекрывающихся областей, и обе могут стать хорошими книгами для начачинающих изучения Erlang. Если вам покажется, что эта книга движется слишком медленно или вам нужно больше справочного материала – переходите на страницы их книг. «Erlang Programming» даст вам информацию о том, что вы можете делать на Erlang, тогда как «Programming Erlang» предоставим вам подробности о настройке среды программирования Erlang.

Что касается более продвинутой стороны, то «Erlang and OTP in Action» (Manning) Мартина Логана, Эрика Мерритта и Ричарда Карлссона открывают высокоскоростное 72-страничное введение в Erlang, а затем проводят большую часть своего времени, применяя OTP – фреймверк Erlang для создания обновляемых и поддерживаемых параллельных приложений. Недавно Франческо Чезарини и Стив Виноски написали книгу «Проектирование для масштабируемости с помощью Erlang/OTP» (O'Reilly). Эта книга сосредоточена на описание создания больших и устойчивых приложений использующих библиотеки OTP.

В конце каждой главы этой книги вы найдете примечание, указывающее на соответствующую информацию о содержании главы в других книгах, посвященных Erlang. Надеюсь, они помогут вам быстро перемещаться между ними, если вы будете использовать эту книгу в качестве дополнения к остальной части растущей библиотеки Erlang.

Если вы хотите сконцентрироваться на использования Erlang для собздания веб-приложений, вам определенно следует изучить «Создание веб-приложений на Erlang» (O'Reilly) от Захария Кессина.

Вы также захотите посетить основной веб-сайт Erlang, https://www.erlang.org/, для получения обновлений, загрузок, документации и многого другого.

Вы уверены, что хотите изучать Erlang?

Возможно, вы заметили, что функциональных языков становится всё больше и больше.

В частности, пять из них – Clojure, Scala, F#, Haskell и Elixir – могут быть более привлекательными, чем Erlang, если у вас есть особые потребности.

- Clojure и Scala работают на виртуальной машине Java (JVM), что делает их безумно переносимыми, и в результате они получают доступ к библиотекам Java.

ClojureScript делает то же самое с JavaScript. (Erjang позволяет запускать Erlang на JVM, но это не основная часть языка.)

- F# работает в среде .NET Common Language Runtime (CLR), что делает его очень переносимым в экосистеме Microsoft и, опять же, имеет доступ к библиотекам .NET.
- Haskell не работает на виртуальной машине, но также предлагает более сильную систему типов и другой тип дисциплины (и лень).
- Elixir построен на тех же основах, что и Erlang, и хорошо работает с Erlang, но имеет Ruby-подобный синтаксис с сильной поддержкой метапрограммирования.

Лично я начал изучать концепции функционального программирования в XSLT. Это совершенно другой тип языка, предназначенный для конкретной области преобразования документов, но многие применимо и там.

Вам, конечно, не нужно решать, является ли Erlang мечтой вашей жизни сейчас. Вы можете изучать концепции в Erlang и применять их в других местах, если это окажется лучшей идеей для вашей работы.

Erlang изменит вас

Прежде чем углубляться, вы должны знать, что работа в Erlang может безвозвратно изменить ваш взгляд на программирование. Сочетание функционального кода, процессорной ориентации и распределенной разработки на первый взгляд может показаться чуждым. Тем не менее, после его внедрения Erlang может изменить способ решения проблем и потенциально затруднить возврат к другим языкам, средам и программным культурам.

Соглашения, используемые в этой книге

В этой книге используются следующие типографские обозначения:

курсивный
: Указывает новые термины, URL-адреса, адреса электронной почты, имена файлов и расширения файлов.

`Моноширинный`
: Используется для текстов программ, а также в абзацах для ссылки на элементы программы, такие как имена переменных или функций, операторы и ключевые слова.

`Моноширинный жирным шрифтом`
: Показывает команды или другой текст, который должен быть набран буквально.

`Моноширинный курсивом`
: Показывает текст, который должен быть заменен предоставленными пользователем значениями или значениями, определенными контекстом.

Этот значок обозначает подсказку, условие или общую заметку.

Этот значок указывает на предупреждение или предостережение.

Замечание по синтаксису Erlang

Синтаксис Erlang, кажется, является камнем преткновения для многих людей. Это не похоже на семейство языков C. Пунктуация отличается, и заглавные буквы имеют значение. Даже точки используются в качестве вывода, а не соединителей!

Для меня синтаксис Erlang в основном выглядит естественным, и я особенно рад, что он отличается от других языков, которые я обычно использую. Я делаю намного меньше путаниц. Вместо того чтобы останавливаться на синтаксисе, я решил просто представить его как есть. Сравнение его с другими языками не кажется полезным, особенно когда разные читатели могут быть из разных областей программирования. Надеюсь, вы найдете синтаксис Erlang таким же приятным, как и я. Если вы просто не сможете принять его, вы можете попробовать Elixir.

Использование примеров кода

Примеры в этой книге предназначены для обучения базовым понятиям небольшими порциями. Хотя вы, безусловно, можете заимствовать код и повторно использовать его по своему усмотрению, вы не сможете сразу же взять код этой книги и создать потрясающее приложение (если, возможно, у вас нет необычной привязанности к расчету скорости падающих объектов).

Примеры в этой книге намеренно просты и, возможно, даже глупы. Они не предназначены для того, чтобы ослеплять или поражать, а для того чтобы вы могли понять, как части сочетаются друг с другом самым простым способом. Однако вы должны быть в состоянии определить, какие шаги необходимо предпринять для создания отличного приложения.

Вы можете загрузить код по ссылке «Примеры» на странице книги по адресу http://shop.oreilly.com/product/0636920056690.do.

Эта книга здесь, чтобы помочь вам выполнить свою работу. В общем, вы можете использовать код из этой книги в своих программах и документации. Вам не нужно обращаться к нам за разрешением, если вы не воспроизводите значительную часть кода. Например, написание программы, которая использует несколько фрагментов кода из этой книги, не требует разрешения. Продажа или распространение CD-ROM с примерами из книг O'Reilly требует разрешения. Чтобы ответить на вопрос, сославшись на эту книгу и приведя пример кода, разрешения не требуется. Включение значительного количества примеров кода из этой книги в документацию вашего продукта требует разрешения.

Мы ценим, но не требуем, атрибуции. Атрибуция обычно включает название, автора, издателя и ISBN. Например: «Introducing Erlang, by Simon St.Laurent (O’Reilly). Copyright 2013 Simon St.Laurent, 9781449331764».

Если вы считаете, что использование примеров кода выходит за рамки добросовестного использования или разрешения, указанного выше, свяжитесь с нами по адресу permissions@oreilly.com.

Помогите улучшить книгу

Я надеюсь, что вам понравится читать эту книгу и учиться у нее, но я также надеюсь, что вы сможете помочь другим читателям изучать здесь Erlang. Вы можете помочь своим читателям несколькими способами:

- Если вы обнаружите конкретные технические проблемы, плохие объяснения или вещи, которые можно улучшить, сообщите о них через систему исправлений.
- Если вам нравится (или не нравится) книга, пожалуйста, оставляйте отзывы. Наиболее заметные места для этого – на Amazon.com (или на его международных сайтах) и на странице O'Reilly для книги по адресу http://shop.oreilly.com/product/0636920056690.do. Подробные объяснения того, что сработало и что не сработало для вас (и более широкой целевой аудитории программистов, плохо знакомых с Erlang), полезны для других читателей и для меня.
- Если вы обнаружите, что у вас есть еще что-то, что вы хотите сказать об Erlang, рассмотрите возможность поделиться этим. Будь то в Интернете, в вашей собственной книге, на учебных занятиях или в любой другой форме, которую вы считаете наиболее простой.

Я обновлю книгу, внеся исправления ошибок и попытаюсь решить проблемы, поднятые в обзорах. Даже после того, как книга «закончена», я все же могу добавить к ней несколько дополнительных частей. Если вы приобрели ее как электронную книгу, вы получите эти обновления бесплатно, по крайней мере, до того момента, когда наступит время для целого нового издания. Однако я не ожидаю, что новая редакция выйдет быстро, если, конечно, экосистема Erlang существенно не изменится.

Надеюсь, что эта книга заинтересует вас настолько, что вы захотите поделиться своими знаниями.

Пожалуйста, используйте полученные знания во имя добра

Я позволю себе уточнить, что значит «добро». Подумайте об этом. Пожалуйста, попробуйте использовать силу Erlang для проектов, которые сделают мир лучше или, по крайней мере, не хуже.

Safari® Books Online

Safari Books Online – это электронная библиотека по запросу, которая предоставляет качественные материалы в виде книг, видео от ведущих мировых авторов в области технологий и бизнеса.

Специалисты по технологиям, разработчики программного обеспечения, веб-дизайнеры, а также профессионалы в области бизнеса и творчества используют Safari Books Online в качестве основного ресурса для исследований, решения проблем, обучения и сертификации.

Safari Books Online предлагает ряд планов и цен для предприятий, правительства, образования и частных лиц.

Участники имеют доступ к тысячам книг, обучающих видео и рукописей для предварительной публикации в одной полностью доступной для поиска базе данных от таких издателей, как O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology и многим другим. Для получения дополнительной информации о Safari Books Online, пожалуйста, посетите нас онлайн.

Как с нами связаться

Пожалуйста, направляйте комментарии и вопросы относительно этой книги издателю:

O’Reilly Media, Inc.
1005 Gravenstein Highway North
PrefaceSebastopol, CA 95472
800-998-9938 (в США или Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс)

У нас есть веб-страница для этой книги, где мы перечисляем ошибки, примеры и любую дополнительную информацию. Вы можете получить доступ к этой странице по адресу https://learning.oreilly.com/library/view/introducing-erlang-2nd/9781491973363/.

Чтобы прокомментировать или задать технические вопросы об этой книге, отправьте электронное письмо по адресу bookquestions@oreilly.com.

Для получения дополнительной информации о наших книгах, курсах, конференциях и новостях посетите наш веб-сайт по адресу http://www.oreilly.com.

Найдите нас на Facebook: http://facebook.com/oreilly

Подпишитесь на нас в Твиттере: http://twitter.com/oreillymedia

Смотрите нас на YouTube: http://www.youtube.com/oreillymedia

Выражение признательности

Большое спасибо Захарию Кессину за то, что он заинтересовал меня Erlang, а также ему и Франческо Чезарини за то, что они побудили меня написать эту книгу. Подробные отзывы от Стива Виноски и Фреда Хеберта позволили, я надеюсь, этой книге, чтобы помочь читателям начать работу в правильном направлении. J. David Eisenberg и Chuck На сделали возможным помочь новичкам начать правильно, указав на пробелы и проблемы в моей прозе.

В частности, спасибо моей жене Анжелике за то, что она подтолкнула меня к этому, моему сыну Конраду за то, что он не слишком много разбрасывал распечатки, и моей дочери Сунгиве за понимание того, что после того, как я рассказал ей историю о Неде и Эрни, приключениях змей, мне нужно было вернуться вниз и поработать над книгой.

Глава 1. Осваиваивайтесь!

Erlang имеет забавную кривую обучения для многих людей. Она начинается плавно на некоторое время, затем становится намного круче, когда вы понимаете, в чем заключается дисциплина, а затем некоторое время становится почти вертикальным, когда вы пытаетесь понять, как эта дисциплина влияет на выполнение работы, а затем она внезапно становится спокойной и мирной с легкой оценкой в течение длительного времени, когда вы повторно применяете то, что вы узнали в разных контекстах.

Перед этим восхождением лучше всего освоиться на солнечных лугах внизу кривой обучения. Оболочка Erlang, ее интерфейс командной строки – это удобное место для начала работы и хорошее место, чтобы начать выяснять, что работает, а что нет в Erlang. Его функции избавят вас от головной боли позже, так что соглашайтесь!

Установка

Файлы установки Erlang можно скачать по адресу https://www.erlang.org/downloads. Для этого издания я использовал Erlang/OTP 19, но любая версия Erlang, более поздняя, чем 17, должна работать.

Если вы используете Windows, то установка Erlang пройдёт легко. Загрузите установочный дистрибутив для Windows, запустите установщик, и все готово. Если вы храбрый новичок, взявшись за свой первый язык программирования, это, безусловно, ваш лучший выбор.

В Linux или Mac OS X вы можете загрузить исходные файлы и скомпилировать их. Если подход к компиляции не работает или не подходит вам, Erlang Solutions предлагает несколько установок по адресу https://www.erlang-solutions.com/resources/download.html. Кроме того, многие различные менеджеры пакетов (Debian, Ubuntu, MacPorts, brew и другие) включают Erlang. Возможно, это будет не самая последняя версия, но наличие Erlang намного лучше, чем его отсутствие.

Во многом благодаря распространению CouchDB, Erlang все чаще становится частью установки по умолчанию на многих системах, включая Ubuntu.

Вперед!

В Mac OS X или Linux перейдите в командную строку и введите `erl`. В Windows перейдите в в командной строке и введите `werl`.

Вы увидите что-то вроде следующего примера кода, вероятно, с курсором рядом с подсказкой `1>`.

```
Erlang R15B (erts-5.9) [source] [smp: 2: 2] [async-threads: 0] [hipe] [kernel-poll: false]
Eshell V5.9 (отмена с ^ G)
1>
```

Добро пожаловать в Erlang!

Первые шаги: оболочка

Прежде чем перейти к восхищению от программирования на Erlang, всегда стоит узнать, как же прекратить работу. Оболочка предлагает использовать ^G, Ctrl-G, что приведет вас к таинственной (пока что) команде переключения команд. (Ctrl-C приведет вас к меню.) Самый простой способ выйти, позволяющий всему, что работает в оболочке, нормально завершиться, это вызов метода `q()`.

```
1> q().
ok
2> SimonMacBook:~ simonstl$
```

Так что вы здесь сделали? Вы ввели команду оболочки, вызвав функцию q, которая сама вызывает функцию `init:stop()`, встроенную в Erlang. Точка после команды говорит Erlang, что вы закончили ввод строки. Команда сообщает `ok`, оболочка печатает новый номер строки (она всегда делает это после почки), и возвращает вас обратно в обычную командную строку, в этом случае оболочку bash на вашем ноутбуке.

Если бы вы не ввели точку после `q()`, то результаты будет выглядеть немного иначе. Вы бы начали новую строку, но счетчик команд не обновился бы, поэтому строка все равно начиналась бы с 1>. Когда это произойдет, вы можете просто напечатать точку (.) и нажать Enter, чтобы завершить команду.

```
1> q()
1> .
ok
2> SimonMacBook:~ simonstl$
```

Включение точки в конце строки скоро станет второй вашей натурой, но если оставить её её не установить, это может привести к путанице.

Выход из Erlang с помощью q(). завершает все, что делает Erlang. Точка. Это хорошо, когда вы работаете локально, но станет плохой идеей, когда вы подключаетесь к удаленной оболочке. Чтобы выйти из оболочки без риска завершить работу среды Erlang в другой системе, вызовите Ctrl-G и затем введите q, а затем клавишу Enter.

Перемещение по тексту

Если вы исследуете работу с оболочкой, то обнаружите, что многое работает так же, как и другие (подобные) оболочки. Клавиши со стрелками влево и вправо перемещают вас назад и вперед. Некоторые из привязок клавиш повторяют текстовый редактор emacs. Ctrl-A перенесет курсор в начало строки, а Ctrl-E перенесет курсор в конец строки. Если вы ввели два символа в неправильной последовательности, нажатие Ctrl-T поменяет их местами.

Как и большинство оболочек Unix, нажатие клавиши Tab заставит оболочку попытаться автоматически завершить то, что вы написали, хотя в этом случае (это является особенность Erlang-оболочки) она ищет имена модулей или функций (вы скоро с ними познакомимся), а не имена файлов.

Кроме того, при вводе закрывающих скобок курсор выделит соответствующую открывающую скобку.

Повторное использование результатов работы

Клавиши со стрелками вверх ↑ и вниз ↓ пролистывают историю ввода, облегчая повторный ввод.

Особенностью использования стрелок вверх и вниз является перебор введённых строк, разделитель, которых является перевод на новую строку. То есть если вы ввели одно выражение на нескольких строках, то вы будете получать не выражение целиком, а её составные части. Если вы хотите увидеть, что было сохранено в истории, наберите команду "h().". Вы также можете указать, сколько строк хранить в истории с помощью функций history(N) и results(N), где N – это номер строки. Вы можете указать Erlang выполнить заданную строку снова с помощью функции e(N), а получить результат выполненного выражения – с помощью функции v(N). Эти номера строк могут быть полезны!

Использоавние символов Unicode

Начиная с Erlang/OTP R16B, вы можете выбрать диапазон ISO Latin-1 или весь диапазон Unicode, указав флаг запуска +pc latin1 или +pc unicode соответственно.

```
$ erl +pc unicode
Erlang R16B (erts-5.10.1) [source] [async-threads:0] [hipe] [kernel-
poll:false]

Eshell V5.10.1  (abort with ^G)
1> [1024].
"È"
2> [1070,1085,1080,1082,1086,1076].
"Юникод"
3> [229,228,246].
"åäö"
4> <<208,174,208,189,208,184,208,186,208,190,208,180>>.
<<"Юникод"/utf8>>
5> <<229/utf8,228/utf8,246/utf8>>.
<<"åäö"/utf8>>
```

Перемещение по каталогам

Оболочка Erlang в некоторой степени понимает файловые системы, потому что вам, возможно, придется перемещаться по каталогам, чтобы получить доступ к файлам. Команды оболочки имеют те же имена, что и команды Unix, но выражаются в виде функций. Оболочка Erlang может быть запущена в любом каталоге. Что понять какой каталог оболочка в данный момент считает текущей, запустите команду `pwd()`:

```
4> pwd().
/Users/simonstl
ok
5>
```

Для того, чтобы другой каталог стал текущим используйте команду cd(), но вам нужно будет заключить аргумент не только в скобки, но в кавычки, предпочтительно в двойные кавычки.

```
5> cd(..).
* 1: syntax error before: '..'
5> cd("..").
/Users
ok
6> cd("simonstl").
/Users/simonstl
ok
7>
```

Вы можете также запустить команду `ls()`, которая выведет список файлов и каталогов в текущем каталоге. Если вы не укажете ему аргумент (путь к каталогу), то получите список файлов в указанном каталоге.

```
7> ls().
erlang
ok
7> ls("erlang/fizzbuzz").
8> ls().
README.md         fizzbuzz.beam     fizzbuzz.erl      rosettacode
run
```

Использование оболочки

Один из самых простых способов начать изучение возможностей оболочки Erlang – это начать её использование, как калькулятора. Вы можете ввести математические выражение и получить результат:

```
Eshell V5.9 (abort with ^G)
1> 2+2.
4
2> 27-14.
13
3> 35*42023943.
1470838005
4> 200/15.
13.333333333333334
5> 200 div 15.
13
6> 200 rem 15.
5
7> 3*(4+15).
57
```

Первые три оператора – это сложение (+), вычитание (-) и умножение (*), которые работают одинаково, независимо от того, работаете ли вы с целочисленными значениями или числами с плавающей запятой. Четвертый, /, поддерживает деление, где вы ожидаете результат с плавающей запятой (число с десятичной частью). Если вы хотите получить целочисленный результат (и иметь целочисленные аргументы), используйте вместо этого оператор div (строка 5). Используйте rem для получения остатка от деления, как показано в строке 6. Скобки позволяют изменить порядок обработки операторов, как показано в строке 7. (Обычный порядок операций указан в Приложении А.)

Erlang будет использование целых чисел вместо дробных не является ошибкой. Обратное утверждение ложно. Если вам нужно преобразовать число с плавающей запятой в целое, вы можете использовать встроенную функцию `round()`:

```
8> round(200/15).
13
```

Функция `round()` удаляет дробную часть числа. Если эта часть была больше или равна 0.5, функция увеличит целую часть до 1, округляя в большую сторону. Если вы

предпочитаете просто отбрасывать десятичную часть полностью, используйте функцию `trunc()`.

Вы также можете отобразить на предыдущий результат по номеру строки, используя функцию v(). Например:

```
9> 4*v(8).
52
```

Результат вычислений в строке 8 был числом 13, следовательно 4 * 13 = 52.

Для удобства, параметр функции v() может быть отрицательным, чтобы ссылаться на предыдущие результаты. v(-1).
Например, предыдущий результат, v(-2) – это результат до него и так далее.

Вызов функций

Если вы хотите выполнять более сложные вычисления, модуль `math` предлагает в значительной степени классический набор функций, поддерживаемых научным калькулятором. Они возвращают значения с плавающей запятой. Константа `pi` доступна как функция, `math:pi()`. Тригонометрические, логарифмические, экспоненциальные, квадратный корень и (кроме ОС Windows) функции распределения Гаусса легко доступны. (Тригонометрические функции берут свои аргументы в радианах, а не в градусах, поэтому будьте готовы преобразовать их в случае необходимости.) Использование этих функций немного многословно из-за необходимости ставить перед ними префикс `math:`, но все же достаточно разумно.

Например, чтобы получить синус нулевого радиана, вы должны написать:

```
1> math:sin(0).
0.0
```

Обратите внимание, что это 0.0, а не просто 0, что означает, что число является дробным.

Чтобы рассчитать косинус Пи и 2pi радиан, вы должны написать:

```
2> math:cos(math:pi()).
-1.0
3> math:cos(2*math:pi()).
1.0
```

Чтобы вычислить 2, взятые в 16-й степени, вы должны написать:

```
4> math:pow(2,16).
65536.0
```

Полный набор математических функций, поддерживаемых модулем math, приведен в Приложении А.

Числа в Erlang

Erlang распознает два вида чисел: целые числа (integer) и числа с плавающей запятой (часто называемые float). Легко представить integer, как «целые числа» без десятичной

части и float, как «десятичные числа» с десятичной точкой и некоторым значением (даже если это 0) справа от десятичной дроби. 1 – это целое число, 1.0 – это число с плавающей запятой.

Тем не менее, есть некоторые особенности. Erlang хранит integer и float по-разному. Erlang позволяет хранить большие числа в виде целых чисел, но независимо от того, большие они или маленькие.

float, с другой стороны, охватывают широкий диапазон чисел, но с ограниченной точностью. Erlang использует 64-битное представление IEEE 754-1985 с «двойной точностью». Это означает, что он отслеживает около 15 десятичных цифр плюс показатель степени. Он также может представлять некоторые большие числа – доступны степени до положительного или отрицательного 308, но поскольку он отслеживает только ограниченное количество цифр, результаты будут отличаться немного больше, чем может показаться удобным, особенно если вы хотите сделать сравнение.

```
1> 34875983475983475984375834758934758437492450.0.
3.4875983475983474e42
2> 2343243.34543589385023454333954 5.
2343243.3454358936
3> 0.00000000000000000000000000000023432432432432234232324.
2.3432432432432235e-30
```

Как вы можете видеть, некоторые цифры остались скрылись, а общая величина числа представлена экспонентой.

Когда вы вводите числа с плавающей запятой, у вас всегда должен быть хотя бы одно значение слева от десятичной запятой, даже если число равно нулю. В противном случае Erlang сообщает о синтаксической ошибке – он не может корректно интерпретировать данный синтаксис.

```
4> .00000000000000000000000000000023432432432432234232324.
* 1: syntax error before: '.'
```

Вы также можете написать число с плавающей запятой, используя цифры плюс обозначение экспоненты:

```
7> 2.923e127.
2.923e127
8> 7.6345435e-231.
7.6345435e-231
```

Отсутствие точности у float может привести к аномальным результатам. Например, синус ноля равен нулю, а синус Пи также равен нулю. Однако, если вы вычислите это в Erlang, вы не достигнете нуля с помощью десятичного приближения, которое Erlang предоставляет для pi:

```
1> math:sin(0).
0.0
2> math:sin(math:pi()).
1.2246467991473532e-16
```

Если бы представление числа пи было длиннее, а его вычисления были продолжены, результат для строки 2 был бы ближе к нулю.

Если вам нужно следить за точность при денежных расчётах, лучше подойдут целые числа. Используйте наименьшую доступную единицу, например, центы для расчётов в долларах США. Помните, что эти центы составляют 1/100 доллара. (Финансовые

транзакции могут идти в гораздо меньших долях, но вы все равно захотите представлять их как целые числа с известным множителем.) Однако для более сложных вычислений вам, возможно, понадобится использовать числа с плавающей запятой, а это может снизить точность вычислений.

Если вам нужно выполнить вычисления для целых чисел, используя основание, отличное от 10, вы можете использовать нотацию Base#Value. Например, если вы хотите указать двоичное значение 1010111, вы можете написать:

```
3> 2#1010111.
87
```

Erlang возвращает число по основанию 10. Точно так же вы можете указать шестнадцатеричные числа, используя 16 вместо 2:

```
4> 16#cafe.
51966
```

Erlang позволяет использовать верхний или нижний регистр для шестнадцатеричных чисел – `16#CAFE` и `16#CaFe` также выдают 51966. Вы не ограничены традиционным двоичным (основание 2), восьмеричным (основание 8) и шестнадцатеричным (основание) 16) выбор. Если вы хотите работать на базе 18 или на любой базе до 36, вы это можете:

```
5> 18#gaffe.
1743080
```

Зачем вам может понадобиться основание 36? Это чрезвычайно простой способ создания ключей, которые выглядят как комбинация букв и цифр, но аккуратно преобразуются в цифры. Шестизначные коды, используемые авиакомпаниями для идентификации билетов, такие как G6ZV1N, легко воспринимаются как база 36. (Однако они обычно пропускают некоторые цифры и буквы, которые легко перепутать, например, 0 и O, 1 и l.)

Чтобы сделать любое из этих чисел отрицательным, просто поставьте перед ними знак минус (-). Это работает как с обычными целыми числами в нотации Base#Value и с плавающей точкой:

```
6> -1234.
-1234
7> -16#cafe.
-51966
8> -2.045234324e6.
-2045234.324
```

Работа с переменными в оболочке Erlang

Хотя функция v() позволяет ссылаться на результаты предыдущих выражений, но это отслеживание результатов не совсем удобно. Кроме того, функция v() работает только в оболочке. Это не универсальный механизм. Более разумное решением было бы хранить значения в текстовых именах, создавая переменные.

Имена переменных Erlang начинаются с заглавной буквы или подчеркивания. Нормальные переменные начинаются с заглавной буквы, в то время как подчеркивания начинаются со специального маркера переменных, означающее упрощенный режим

анализа её использования. Мы сосредоточим свое внимание на использовании нормальных переменных. Итак, для присваивания значения переменной, используйте синтаксис, который должен быть вам знаком по алгебре или другим языкам программирования. Здесь N является переменной:

```
1> N=1.
1
```

Чтобы вывести значение переменной, просто введите ее имя:

```
2> N.
1
```

Чтобы увидеть, как Erlang протестует против вашего грубого, некорректного поведения, попробуйте присвоить переменной новое значение:

```
3> N=2.
** exception error: no match of right hand side value 2
4> N=N+1.
** exception error: no match of right hand side value 2
```

Что тут происходит? Erlang ожидает, что правая часть выражения после = будет соответствовать левой стороне. Он готов сделать это, если переменная слева еще не связана, как это было в случае с N = 1 в первой строке. Однако, как только переменная N установлена в 1, Erlang интерпретирует N = 2 как 1 = 2, что является ошибкой. N = N + 1 также оценивается как 1 = 2, и не работает. Модель единственного назначения Erlang, в которой каждой переменной может быть присвоено значение только один раз в данном контексте, налагает дисциплину, значение которой вы увидите в последующих главах.

Выражения Erlang работают как алгебра, где N никогда не равно N + 1. Такого просто не может быть. Однако, как только вы установите N на 1, можно попробовать выражения, которые также равняются единице:

```
5> N=2-1.
1
6> N=15 div (3*5).
1
```

Это станет намного важнее, когда вы начнете использовать возможности Erlang для сопоставления с образцом. Вы также можете написать следующее:

```
7> 1=N.
1
```

Erlang не будет пытаться связать какие-либо переменные, когда они появляются с правой стороны от знака равенства. В это выражение Erlang просит сравнить 1 и 1. Однако попробуйте это с 2. Erlang выдаст сообщение об ошибке: нет совпадения, так как 2 не равно 1:

```
8> 2=N.
** exception error: no match of right hand side value 1
```

Вы также можете использовать связанные переменные в вычислениях, например, для создания новых связанных переменных. Например, создание переменной `Number`:

```
9> Number=N*4+N.
5
10> 6*Number.
30
```

Когда вы присваиваете значение переменной, вы должны убедиться, что все вычисления находятся справа от знака равенства. Если мне известно, что M должно быть 6, когда 2 * M = 3 * 4, Erlang не может выполнить привязку переменной:

```
11> 2*M=3*4.
* 1: illegal pattern
```

Оболочка будет помнить ваши переменные до тех пор, пока вы не выйдете или не скажете забыть их. Код в функциях Erlang не забывает о созданных переменных, до тех пор пока функции не завершит свою работу.

Вывод информации о всех связанных переменных

Через некоторое время работы в оболочке, используя её в качестве калькулятора (попробуйте это!), вы можете забыть, какие переменные вы уже связали. Если вам нужно напоминание, команда оболочки b() в этом поможет:

```
11> b().
N = 1
Number = 5
ok
```

Очистка информации о связанных переменных

В оболочке и только в оболочке вы можете уничтожить все переменные или какою-либо определённую переменную. Это может оказаться полезным после досадной опечатки или при необходимости сброса состояния вашей консоли для новых вычислений, но этого у вас не получится, при работе ваших программ в обычном режиме.

Чтобы очистить определенную переменную, удалив ее привязку и позволив установить новое значение, используйте функцию `f()`, указав имя переменной в качестве аргумента:

```
12> f(N).
ok
13> Number=5.
ok
14> N=2.
2
```

Чтобы очистить все связанные переменные в оболочке, просто вызовите `f()` без аргументов.

```
15> b().
N = 2
Number = 5
ok
16> f().
ok
17> b().
ok
```

Как вы видите переменные отсутствуют.

Прежде чем перейти к следующей главе, в которой будут представлены модули и функции, проведите некоторое время, работая в оболочке Erlang. Опыт, даже на этом простом уровне, поможет вам двигаться вперед. Используйте переменные и посмотрите,

что происходит при работе с большими целыми числами. Erlang очень хорошо поддерживает большие числа. Попробуйте смешать числа с десятичными значениями (числами с плавающей запятой) и целыми числами в вычислениях и посмотрите, что произойдет. У вас не должно возникнуть каких-либо сложностей, хотя я подозреваю, что идея переменных, которые не меняют значения, даст вам подсказку о том, во что она может развиться.

Подробнее об установке и работе с оболочкой вы можете узнать в главе 2 «Erlang Programming» (O'Reilly); В главах 2 и 6 книги «Programming Erlang» (Pragmatic); в разделе 2.1 «Erlang and OTP in Action» (Manning); из главы 1 «Learn You Some Erlang For Great Good!» (No Starch Press).

Глава 2. Функции и модули

Как и большинство языков программирования, Erlang позволяет вам определять функции, которые помогут вам выполнять повторяющиеся вычисления. Хотя функции Erlang могут усложняться, они начинаются достаточно просто.

Анонимные функции

Вы можете создавать функции в оболочке Erlang, используя подходящее имя fun. Например, чтобы создать функцию, которая вычисляет скорость падающего объекта на основе расстояния, которое он падает в метрах, вы можете создать следующее:

```
1> FallVelocity = fun(Distance) -> math:sqrt(2 * 9.8 * Distance) end.
#Fun <erl_eval.6.111823515>
```

Вы можете прочитать это как сопоставление с образцом, который связывает переменную `FallVelocity` с функцией, которая принимает аргумент `Distance`. Функция возвращает (мне нравится читать -> в качестве выходных данных) квадратный корень в 2 раза больше гравитационной постоянной для Земли, равной 9,8 м / с, умноженной на Расстояние (в метрах). Затем функция заканчивается, и точка закрывает оператор.

Если вы хотите включить несколько утверждений в функцию, разделите их запятыми, например
FallVelocity = fun(Distance) -> X = (2 * 9.8 * Distance), math:sqrt(X) end.

Возвращаемое значение в оболочке, #Fun<erl_eval.6.111823515>, само по себе не имеет особого значения, но говорит о том, что вы создали функцию. Если вам нужнf более подробная информация о том, что Erlang получил от вас, вы можете использовать команду оболочки b(), чтобы увидеть, что у него есть:

```
2> b().
FallVelocity =
        fun(Distance) ->
                math:sqrt(2 * 9.8 * Distance)
        end
ok
```

Весьма удобно, привязав функцию к переменной `FallVelocity`, можете использовать эту переменную для расчета скорости падения объектов на Землю:

```
3> FallVelocity(20).
19.79898987322333
4> FallVelocity(200).
62.609903369994115
5> FallVelocity(2000).
197.9898987322333
```

Если вы хотите получить скорость не в метрах в секунду, а в милях в час, просто создайте другую функцию. Вы можете скопировать и вставить предыдущие результаты в него (как я сделал здесь), или выбрать другие:

```
6> Mps_to_mph = fun(Mps) -> 2.23693629 * Mps end.
#Fun<erl_eval.6.111823515>
```

```
7> Mps_to_mph(19.79898987322333).
44.289078952755766
8> Mps_to_mph(62.609903369994115).
140.05436496173314
9> Mps_to_mph(197.9898987322333).
442.89078952755773
```

Думаю, я буду держаться подальше от падений с высоты 2000 метров. А Вы предпочитаете скорость падения в километрах в час?

```
10> Mps_to_kph = fun(Mps) -> 3.6 * Mps end.
#Fun<erl_eval.6.111823515>
11> Mps_to_kph(19.79898987322333).
71.27636354360399
12> Mps_to_kph(62.609903369994115).
225.3956521319788
13> Mps_to_kph(197.9898987322333).
712.76363543604
```

Вы также можете перейти к нужному измерению, вложив следующие вызовы функций одну в другую:

```
14> Mps_to_kph(FallVelocity(2000)).
712.76363543604
```

Как бы вы это ни представляли, это действительно быстро, хотя сопротивление воздуха сильно замедлит их в реальности.

Это удобно для повторных вычислений, но вы, вероятно, не захотите слишком часто использовать такого рода функций в оболочке, так как сброс переменных или выход из сеанса оболочки удалит все ваши функции.

Если вы получили ошибку, похожую на `** exception error: no function clause matching erl_eval:'-inside-an-interpreted-fun-'(value)`, проверьте правильность написания переменной. Может потребоваться некоторое время, чтобы привыкнуть к языковой семантике.

Объявление модулей

Большинство программ Erlang определяют свои функции в скомпилированных модулях, а не в оболочке. Модули являются более формальным местом для размещения программ, и они дают вам возможность более эффективно хранить, инкапсулировать, обмениваться и управлять вашим кодом.

Код каждого модуля должен храниться в своем собственном файле с расширением `.erl`. Имя файла должно иметь вид `name_of_module.erl`, где `name_of_module` - это имя, которое вы указываете внутри файла модуля. Пример 2-1, который вы можете найти в архиве примеров по адресу *ch02/ex1-drop*, показывает, как может выглядеть модуль `drop.erl`, содержащий ранее определенные функции.

Пример 2-1. Модуль для расчета и преобразования скоростей падения

```
-module(drop).
-export([fall_velocity/1, mps_to_mph/1, mps_to_kph/1]).

fall_velocity(Distance) -> math:sqrt(2 * 9.8 * Distance).
```

```
mps_to_mph(Mps) -> 2.23693629 * Mps.

mps_to_kph(Mps) -> 3.6 * Mps.
```

В этом модуле есть два ключевых вида информации. Вверху директивы `-module` и `-export` сообщают компилятору о модуле – его имени и функциях, которые он должен сделать видимыми для другого кода, использующего этот модуль. Директива `-export` предоставляет список функций, которые следует сделать видимыми не только их имена, но и их арность, количество аргументов, которые они принимают. Erlang рассматривает функции с одинаковым именем, но разной арностью как разные функции.

Весь код в модуле должен содержаться в функциях.

Ниже директив представлен набор выражений, определяющих функции, которые похожи на объявление анонимных функций, которые использовались ранее, но не совсем так. Имена функций начинаются со строчных, а не прописных букв, а синтаксис немного отличается. Ключевые слова `fun` и `end` не появляются, и сразу после имени функции следуют скобки, содержащие набор аргументов.

Если вы получаете такие ошибки, такие как "`drop.erl:2: bad function arity drop.erl:6: syntax error before: Fall_velocity`", возможно, вы не конвертировали имена из ваших анонимных функций и поэтому они начинаются со строчной буквы

Как же со всем этим сделать что-либо полезное?

Пришло время начать компилировать код Erlang. Оболочка позволит вам скомпилировать модули и сразу использовать их. Функция `c()` позволяет вам компилировать код. Вы должны находиться в том же каталоге, что и файл, независимо от того, запустили ли вы оттуда оболочку Erlang или перешли туда с помощью команд, показанных в предыдущей главе. Вам не нужно включать расширение файла `.erl` в имя, которое вы передаете функции `c()`, хотя вы можете указать пути к каталогам.

```
1> ls().
drop.erl
ok
2> c(drop).
{ok,drop}
3> ls().
drop.beam drop.erl
ok
```

В строке 1 проверяется, есть ли исходный файл drop.erl, и вы видите список каталогов. Строка 2 фактически компилирует его, а строка 3 показывается, что теперь доступен новый файл drop.beam. Теперь, когда у вас есть drop.beam, вы можете вызывать функции из модуля. Вам нужно поставить префикс перед этими вызовами, как будет показано в строках 4 и 5 следующего кода.

```
4> drop:fall_velocity(20).
19.79898987322333
5> drop:mps_to_mph(drop:fall_velocity(20)).
44.289078952755766
```

Эти функции работают так же, как и их предшественники, но теперь вы можете выйти из оболочки, вернуться сюда и по-прежнему использовать скомпилированные функции.

```
6> q().
ok
$ erl
Erlang R15B (erts-5.9) [source] [smp:8:8] [async-threads:0] [hipe] [kernel-poll:false]
Eshell V5.9 (abort with ^G)
1> drop:mps_to_mph(drop:fall_velocity(20)).
44.289078952755766
```

Основная работы при программировании на Erlang (помимо работы с оболочкой) сводится к созданию функций в модулях и соединениях их в более крупные программы.

Erlang Компиляция и система времени выполнения

Когда вы пишете Erlang в оболочке, она должна интерпретировать каждую команду, независимо от того, написали ли вы её раньше или нет. Когда вы говорите Erlang скомпилировать файл, он преобразует ваш текст во что-то, что он может обработать, без необходимости повторной интерпретации всего текста, что значительно повышает эффективность при запуске кода. Это «то, что он может обработать» в случае Erlang, является файлом BEAM. Он содержит код, который может запустить процессор BEAM, ключевой элемент Erlang Runtime System (ERTS). BEAM – это абстрактная машина Богдана для абстрактной машины Erlang, которая интерпретирует оптимизированный код BEAM. Это может звучать несколько менее эффективно, чем традиционная компиляция в машинный код, который выполняется непосредственно на компьютере, но напоминает другие виртуальные машины. (Oracle Virtual Machine (JVM) Oracle и Common Language Runtime, используемые Microsoft .NET Framework, являются двумя наиболее распространенными виртуальными машинами.)

Наличие собственной виртуальной машины и системы времени выполнения позволяет Erlang оптимизировать некоторые ключевые вещи, упрощая создание надежных масштабируемых приложений. Его планировщик процессов упрощает распределение работы между несколькими процессорами на одном компьютере. Вам не нужно думать о том, сколько процессоров может использовать ваше приложение – вы просто пишете независимые процессы, и Erlang распространяет их. Erlang также управляет вводом и выводом по-своему, избегая стилей подключения, которые блокируют другую обработку. Виртуальная машина также использует стратегию сбора мусора, которая соответствует ее стилю обработки, что позволяет делать более короткие паузы при выполнении программы. (Сборка мусора освобождает память, которая в определенный момент обрабатывается, но больше не используется.)

При создании и доставке программ Erlang вы будете распространять их как набор скомпилированных файлов BEAM. Вам не нужно компилировать каждый файл из оболочки, как мы здесь делаем. `erlc` позволит вам напрямую скомпилировать файлы Erlang и объединить эту компиляцию в задачи `make` и аналогичные вещи, тогда как `escript` может скомпилировать или интерпретировать и запускать код Erlang извне оболочки Erlang.

Комбинирование модулей и анонимных функций

Если хотите совместить всё лучшее от анонимный функции и функций модуля, то вы можете использовать функции модуля при объявлении переменных. Для этого используйте имя функции и её арность при объявлении переменой. Например:

```
1> F_v = fun drop:fall_velocity/1.
#Fun<drop.fall_velocity.1>
```

```
2> F_v(20).
19.79898987322333
```

Вы также можете сделать это и в коде модуля. Если вы ссылаетесь на код в том же модуле, вы можете пропустить префикс имени модуля. (В этом случае это означало бы, что можно было бы не писать префикс drop, а просто использовать `fall_velocity/1`).

Область действия функций и переменных

Хотя Erlang позволяет вам связывать переменную только один раз, вы можете вызывать функцию много раз в коде программы. Но разве это не означает, что одна и та же переменная будет связана много раз?

Да, она будет связана много раз, но всегда в разных контекстах. Erlang не считает несколько вызовов одной и той же функции одним и тем же. Всё начинается с создания нового набора неназначенных переменных каждый раз, когда вы вызываете эту функцию.

Точно так же Erlang не беспокоится, если вы используете одно и то же имя переменной в разных функциях или функциональных предложениях. Они не будут вызываться в одном и том же контексте в одно и то же время, поэтому ошибки не происходит.

Место, где вам нужно избегать повторного присвоения значений уже связанной переменной, находится в заданном пути к функцию. Пока вы не пытаетесь повторно использовать переменную в данном контексте, вам не о чем беспокоиться.

Директивы модуля

По умолчанию модули имеют очень тонкую оболочку, и все внутри неё считается закрытым. Всё, что входит в модуль или выходит из него, нуждается в явном объявлении и вы предоставляете их через директивы модуля (иногда называемые атрибутами модуля).

В приведенном выше примере показаны две директивы модуля `-module` и `-export`. Директива `-module` устанавливает имя для модуля, который нужно знать внешнему коду для вызова функций. Директива `-export` указывает, к каким функциям может обращаться внешний код.

Модуль drop в настоящее время смешивает два разных вида функций. Функция `fall_velocity/1` очень хорошо вписывается в название модуля `drop`, обеспечивая расчет на основе высоты, с которой падает объект. Функции `mps_to_mph/1` и `mps_to_kph/1`, однако, не предназначены для равчётов. Они являются общими функциями преобразования измерений, которые полезны и в других контекстах, принадлежащих их собственному модулю. В Примере 2-2 и Примере 2-3, в папке *ch02/ex2-combined*, показано, как этого можно достичь.

Пример 2-2. Модуль для расчета скорости падения (drop.erl)

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

fall_velocity(Distance) -> math:sqrt(2 * 9.8 * Distance).
```

Пример 2-3. Модуль для преобразования скоростей падения (convert.erl)

```
-module(convert).
-export([mps_to_mph/1, mps_to_kph/1]).

mps_to_mph(Mps) -> 2.23693629 * Mps.

mps_to_kph(Mps) -> 3.6 * Mps.
```

Далее вы можете объединить их, и тогда отдельные функции станут доступны для использования:

```
Eshell V5.9 (abort with ^G)
1> c(drop).
{ok,drop}
2> c(convert).
{ok,convert}
3> ls().
convert.beam convert.erl drop.beam drop.erl
ok
4> convert:mps_to_mph(drop:fall_velocity(20)).
44.289078952755766
```

Этот код более аккуратный, но как он мог бы работать, если бы третий модуль нуждался в вызове этих функций? Модули, которые вызывают код из других модулей, должны указывать это явно. В примере 2-4, в *ch02/ex3-comb*, показан модуль, который использует функции как из модулей `drop`, так и из модулей `convert`.

Пример 2-4. Модуль для объединения логики модулей drop и covert (combined.erl)

```
-module(combined).
-export([height_to_mph/1]).

height_to_mph(Meters) -> convert:mps_to_mph(drop:fall_velocity(Meters)).
```

Это очень похоже на то, как вы вызывали его из оболочки Erlang, но если у вас много обращений к внешним модулям, это может быстро привести к многословности. Директива -import, показанная в Примере 2-5, позволяет упростить ваш код, хотя и может привести к путанице у людей, которые считают, что импортированные функции должны быть определены в этом модуле. (Вы можете найти это в *ch02/ex4-combined*.)

Пример 2-5. Модуль для объединения логики модулей drop и covert с использованием импорта

```
-module(combined).
-export([height_to_mph/1]).
-import(drop, [fall_velocity/1]).
-import(convert, [mps_to_mph/1]).

height_to_mph(Meters) -> mps_to_mph(fall_velocity(Meters)).
```

На данный момент, вероятно, лучше знать о директиве -import, чтобы вы могли читать чужой код, но я не рекомендую использовать его, если на то нет веских причин. Это может затруднить определение источника ошибок.

Erlang включает еще одну директиву, которая весьма удобна, но её не рекомендуется использовать: `-compile(export_all)`. Эта директива разрушает стенку модуля, делая все функции доступными для внешних вызовов. В модуле, где все должно быть общедоступным, это может спасти вас, печатая все функции и все арности вашего модуля. Тем не менее, это также означает, что любой может вызывать что угодно в вашем коде, предоставляя гораздо большую площадь для недопонимания и сложной отладки. Если вы просто не можете сопротивляться, используйте её, но всё-таки попытайтесь сопротивляться.

Вы также можете составить свои собственные директивы пользователя. -author(имя) и -date(дата) обычно используются. Если вы составляете свои собственные директивы, они могут иметь только один аргумент. Если вы проведете достаточно времени в Erlang, вы также столкнетесь со следующими директивами: `-behaviour(Behaviour)`, `-record(Name, Fields)`, и `-vsn(Version)`.

Документирование исходного кода

Ваши программы могут прекрасно работать и без документации. Однако пользоваться вашим проектам будет намного сложнее.

В то время как программистам нравится думать, что они пишут код, который каждый может просмотреть и разобраться, болезненная реальность заключается в том, что код, даже немного более сложный, чем код, показанный в предыдущих примерах, может оказаться загадочным для других разработчиков. Если вы на какое-то время отойдете от кода, понимание, которое вы реализовали во время его программирования, может исчезнуть, и даже ваш собственный код может показаться непонятным.

Самый простой способ добавить более явные объяснения к вашему коду – это вставить комментарии. Вы можете начать комментарий с символа % (он действует до конца строки). Некоторые комментарии занимают целую строку, в то время как другие представляют собой короткие фрагменты в конце строки.
Пример 2-6 показывает оба варианта комментариев.

Пример 2-6. Комментарии в действии

```
-module(combined).
-export([height_to_mph/1]). % there will be more soon!

%%% combines logic from other modules into a convenience function.
height_to_mph(Meters) -> convert:mps_to_mph(drop:fall_velocity(Meters)).
```

Компилятор Erlang будет игнорировать весь текст между знаком % и концом строки, но те, что будет изучать ваш код, смогут его прочитать.

Почему в начале строки несколько знаков процента? Режим Erlang Emacs и многие другие инструменты Erlang ожидают, что число знаков процента будет указывать на уровни отступа. Три знака процента (%%%) означают, что комментарий будет отформатирован по левому краю, два знака процента (%%) означают, что комментарий имеет отступ от окружающего кода, а один знак процента (%) используется для комментариев в конце строки.

Неформальные комментарии полезны, но разработчики имеют привычку включать комментарии, которые помогают им отслеживать, что они делают, пока пишут код. Эти комментарии могут и не быть понятными другим разработчикам(и даже вам самим). Более формальные структуры комментариев могут быть более трудоемкими, чем вы хотели бы выполнять в пылу сеанса программирования, но они создаются для документации.

Erlang включает в себя систему документации EDoc, которая преобразует комментарии, размещенные в коде, в доступную HTML документацию. Он опирается на специально отформатированные комментарии, директивы и иногда дополнительный файл для предоставления структурированной информации о ваших модулях и приложении.

Правила документирования модулей

Модули в этой главе пока очень просты, но их достаточно для начала документирования, как показано в файлах в *ch02/ex5-docs*. Пример 2-7 представляет модуль drop с дополнительной информацией о том, кто его создал и почему.

Пример 2-7. Документированный модуль для расчета скорости падения

```
%% @author Simon St.Laurent <simonstl@simonstl.com> [http://simonstl.com]
%% @doc Functions calculating velocities achieved by objects
%% dropped in a vacuum.
%% @reference from <a href= "http://shop.oreilly.com/product/0636920025818.do" >Introducing
Erlang</a>
%% O'Reilly Media, Inc., 2012.
%% @copyright 2012 by Simon St.Laurent
%% @version 0.1
-module(drop).
-export([fall_velocity/1]).

fall_velocity(Distance) -> math:sqrt(2 * 9.8 * Distance).
```

Erlang может создавать файлы документации для вас, используя функционал EDoc:

```
Eshell V5.9 (abort with ^G)
1> edoc:files(["drop.erl"], [{dir, "doc"}]).
ok
```

Теперь у вас будет набор файлов в подкаталоге doc. Если вы откроете drop.html в браузере, вы увидите что-то вроде рисунка 2-1.

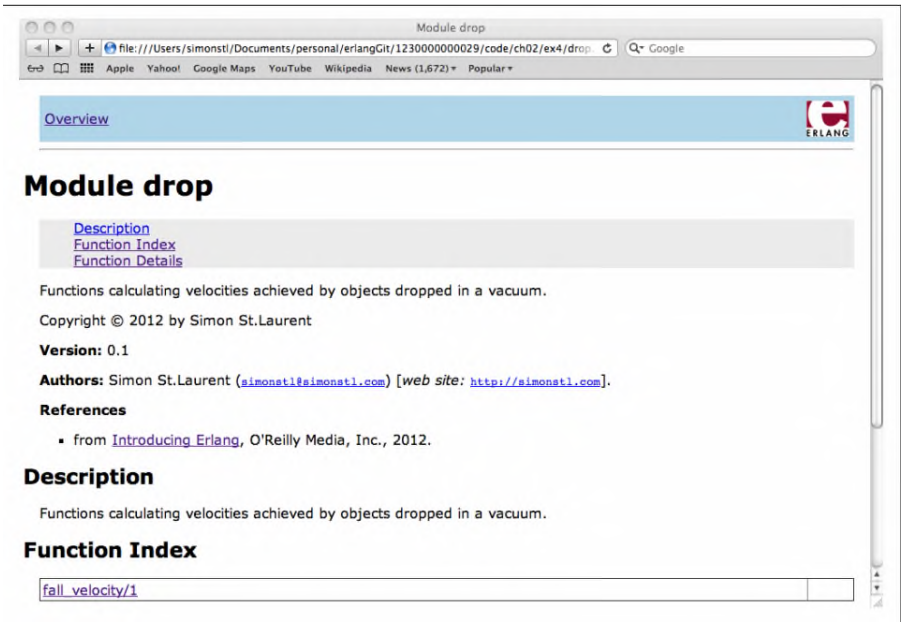


Рисунок 2-1. Документация модуля, созданного с помощью edoc

Все эти метаданные великолепны. Однако, если у вас нет сложной истории, чтобы рассказать о вашем модуле в целом, вполне вероятно, что главная информация будет сосредоточена над функциями.

Документирование функций

Модуль удаления содержит одну функцию: `fall_velocity/1`. Вы, наверное, знаете, что для расстояния, падающего в вакууме на Земле, требуется расстояние в метрах и скорость в метрах, но код не говорит об этом. В примере 2-8 показано, как это исправить с помощью EDoc-комментариев и тега @doc.

Пример 2-8. Документирование функции расчёта расстояния

```
%% @doc Calculates the velocity of an object falling on Earth
%% as if it were in a vacuum (no air resistance). The distance is
%% the height from which the object falls, specified in meters,
%% and the function returns a velocity in meters per second.

fall_velocity(Distance) -> math:sqrt(2 * 9.8 * Distance).
```

На рисунке 2-2 показан результат, который значительно полезнее, чем предыдущий. Он аккуратно берет первое условие, следующее за @doc, и помещает ее в указатель, используя полное описание для раздела «Сведения о функции». Вы также можете использовать разметку XHTML в разделе @doc.

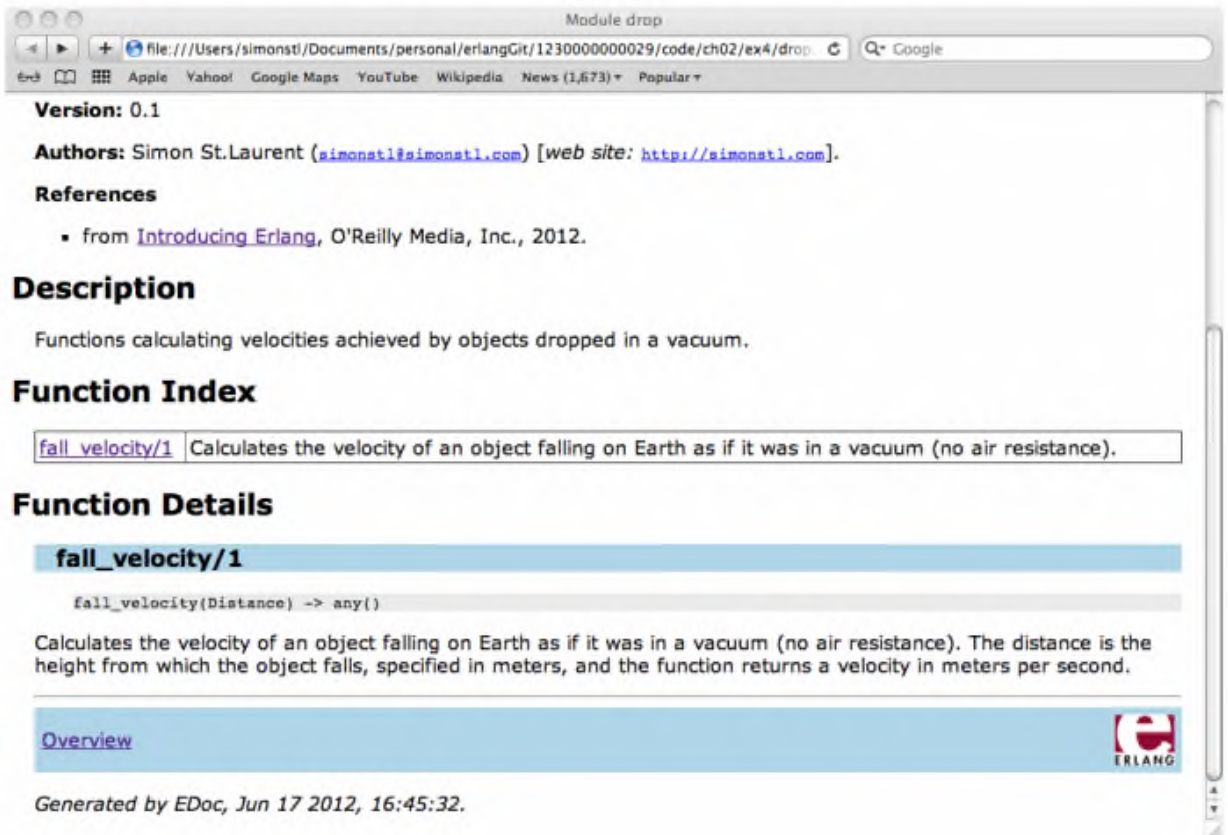


Рисунок 2-2. Документация по функциям, сгенерированная из файла drop.erl

Это серьезное улучшение, но что, если пользователь введёт текст «двадцать» метров вместо 20 метров? Поскольку Erlang не особо беспокоится о типах, код Erlang не говорит о том, что значение для Distance должно быть числом, иначе функция вернет ошибку.

Вы можете добавить директиву -spec, чтобы добавить эту информацию. Это немного странно, так как в некотором смысле это похоже на дубликат объявления метода. Её использование показано в примере 2-9.

Пример 2-9. Документированная функция для расчёта скорости падения

```
%% @doc Calculates the velocity of an object falling on Earth
%% as if it was in a vacuum (no air resistance). The distance is
%% the height from which the object falls, specified in meters,
%% and the function returns a velocity in meters per second.

-spec(fall_velocity(number()) -> number()).

fall_velocity(Distance) -> math:sqrt(2 * 9.8 * Distance).
```

EDoc объединил типы, указанные в директиве -spec, с именами параметров в фактическом объявлении функции, чтобы получить документацию, показанную на рисунке 2-3.

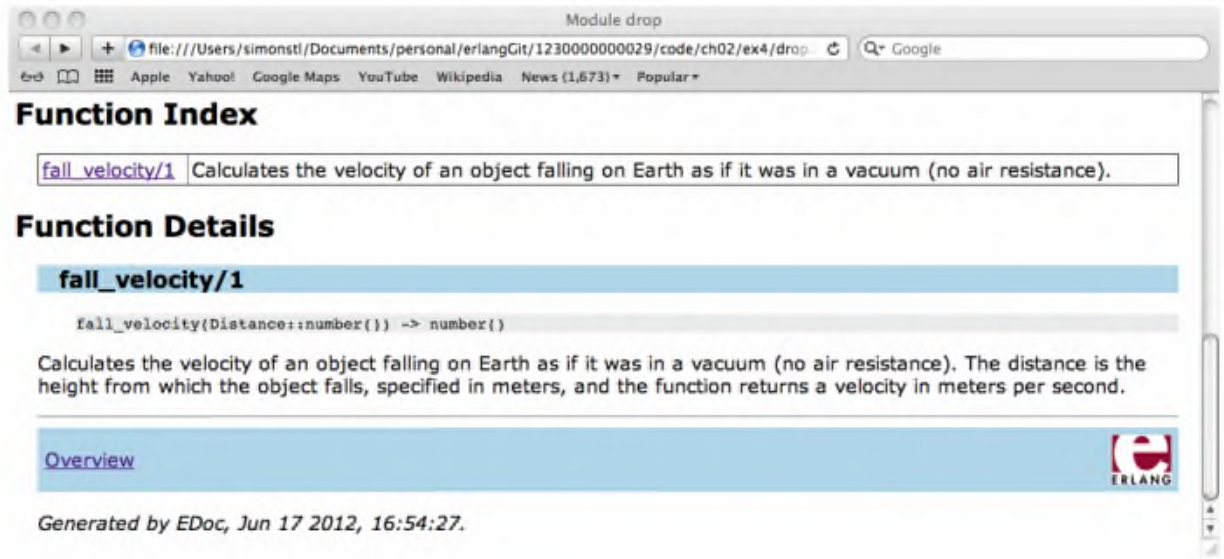


Рисунок 2-3. Документация EDoc с информацией о типе

Эта глава уже демонстрировала тип данных number(), которое сочетает в себе integer() и float(). Приложение А включает содержит полный список типов.

Документирование вашего приложения

Иногда вы хотите, чтобы информация, такая как автор и данные об авторских правах, появлялась в каждом модуле, часто, когда она варьируется от модуля к модулю. В других случаях это становится беспорядочным, и легче поместить его в одно место, где он применяется ко всем вашим модулям.

Для этого необходимо создать файл overview.edoc в *подкаталоге doc* вашего проекта. Его содержимое очень похоже на разметку, используемую в модулях, но, поскольку оно не смешано с кодом, вам не нужно вводить каждую строку с %%. Файл overview.edoc для этого проекта может выглядеть как в Пример 2-10.

Пример 2-10. Документированный модуль для расчета скорости падения

```
@author Simon St.Laurent <simonstl@simonstl.com> [http://simonstl.com]
@doc Functions for calculating and converting velocities.
@reference from <a href= "http://shop.oreilly.com/product/0636920025818.do"
>Introducing Erlang</a>, O'Reilly Media, Inc., 2012.
@copyright 2012 by Simon St.Laurent
@version 0.1
```

Теперь, если вы заново сгенерируете документацию и нажмете на ссылку Overview, вы увидите то, что показано на Рисунок 2-4.

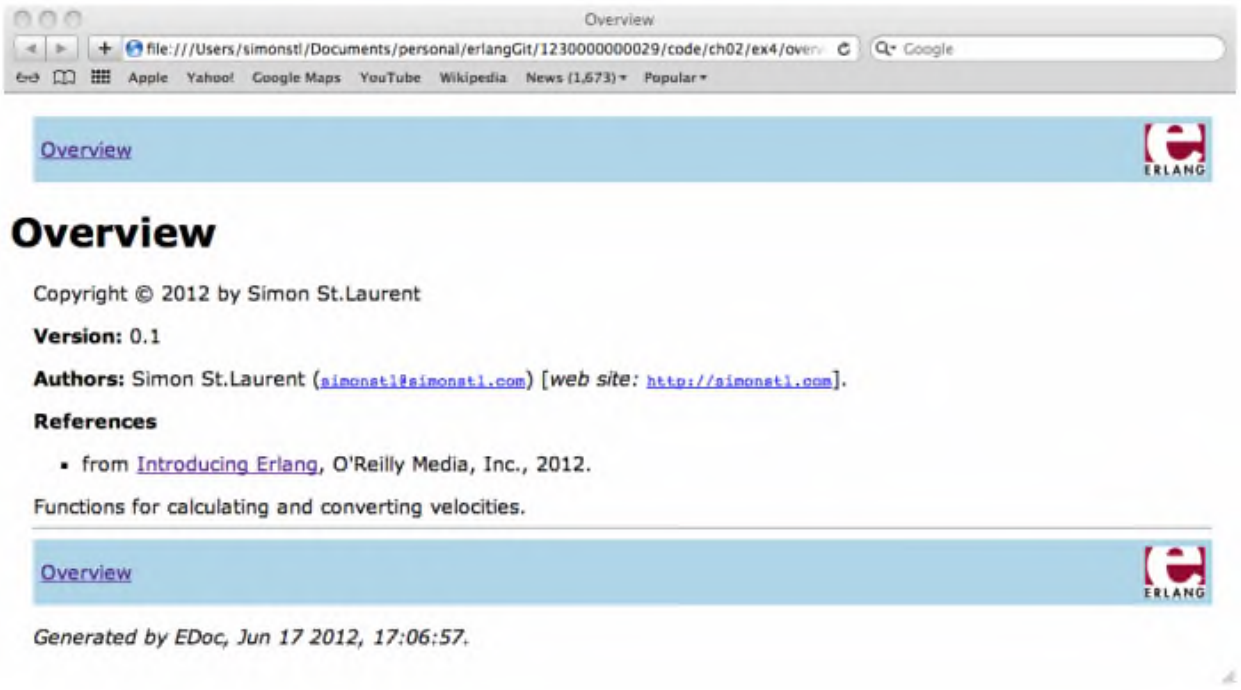


Рисунок 2-4. Обзор приложения, созданный с помощью EDoc

Если вы создаете аналогичную документацию в каждом из файлов Erlang и запускаете `edoc:files(["drop.erl", "convert.erl", "combined.erl"], [{dir, "doc"}]).` в оболочке Erlang, EDoc создаст аккуратный, хотя и несколько простой набор основанной на фреймах документации для вашего приложения, как показано на рисунке 2-5.

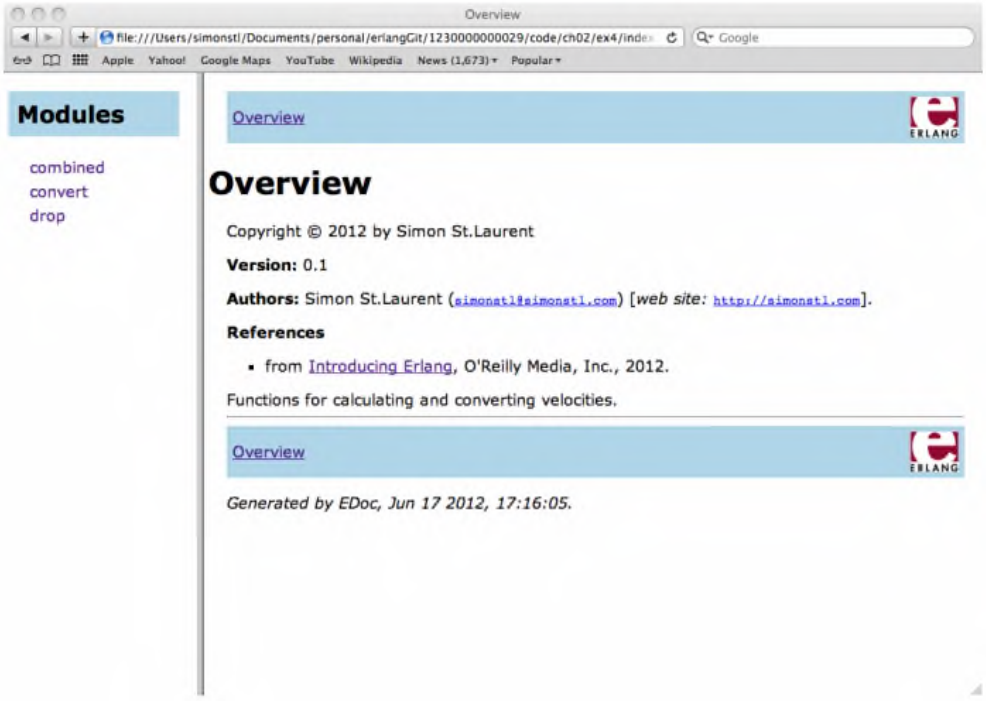


Рисунок 5. Вид комплекта модульной документации

Это всего лишь введение в EDoc. Для получения дополнительной информации см. Главу 18 книги «Erlang Programming», где вы можете узнать о таких интересных конструкциях, как тег @todo.

Подробнее о работе с функциями и модулями вы можете узнать в главах 2, 3 и 9 «Erlang Programming» (O'Reilly); Глава 3 «Programming Erlang» (Pragmatic); Разделы 2.3, 2.5 и 2.7 «Erlang and OTP in Action» (Manning); и главы 2 и 3 «Learn You Some Erlang For Great Good!» (No Starch Press). Более подробную информацию о документации можно найти в главе 18 «Erlang Programming» и о типах в главе 30 «Learn You Some Erlang For Great Good!».

Глава 3. Атомы, Кортежи и сопоставление с образцом

Программы Erlang – это набор запросов и инструментов для их обработки. Erlang предоставляет инструменты, которые упрощают эффективную обработку этих сообщений, позволяя вам создавать код, который будет читабелен (по крайней мере, для программистов), но при этом будет работать эффективно, когда вам нужна скорость.

Атомы

Атомы являются ключевой структурой в Erlang. Технически, это просто тип данных, но трудно переоценить их влияние на стиль программирования Erlang.

Обычно атомы – это блоки текста, которые начинаются со строчной буквы, например ok или earth. Они также могут содержать (хотя и не начинаться) с подчеркивания (_) и символ (@), например this_is_a_short_sentence или me@home. Если вы хотите больше свободы, чтобы начать с заглавных букв или использовать пробелы, вы можете поместить их в одинарные кавычки, например, 'Today is a good day'. Как правило, текст в строчной форме из одного слова легче читается.

Атомы имеют своё значение – это то же самое, что и их текст. (Помните, что точка после hello не является частью атома – она просто заканчивает выражение.)

```
1> hello.
hello
```

Всё это очевидно. Что делает атомы особенными, так это то, что они могут сочетаться с другими типами и методами сопоставления с образцами Erlang для создания простых, но мощных логических структур.

Использование атомов при сопоставлении и образцом

Erlang использовал сопоставление с образцом, чтобы примеры из Главы 2 работали, но они были очень простые. Имя функции было одним ключевым элементом, который менялся, и, пока вы указали числовой аргумент, Erlang знал, что вы имели в виду. Однако сопоставление с образцом в Erlang предлагает гораздо более сложные возможности, позволяя сопоставлять как аргументы, так и имена функций.

Например, предположим, что вы хотите вычислить скорость падающих объектов не только на Земле, где гравитационная постоянная составляет 9,8 метра в секунду в квадрате, но и на поверхности Луны, где она составляет 1,6 метра в секунду в квадрате, и на Марсе, где она составляет 3,71 метра в секунду в квадрате. Пример 3-1, который вы можете найти в *ch03/ex1-atom.erl*, показывает один из способов создания кода, который это поддерживает.

Пример 3-1. Сопоставление с образцом с помощью атомов и имён функций

```
-module(drop).
-export([fall_velocity/2]).

fall_velocity(earth, Distance) -> math:sqrt(2 * 9.8 * Distance);
fall_velocity(moon, Distance) -> math:sqrt(2 * 1.6 * Distance);
fall_velocity(mars, Distance) -> math:sqrt(2 * 3.71 * Distance).
```

Код выглядит так, как будто функция fall_velocity определяется здесь три раза и, безусловно, предоставляет три пути обработки для одной и той же функции. Однако поскольку эти определения разделяются точкой с запятой, они рассматриваются как варианты, выбранные путем сопоставления с образцом, а не как дубликаты определений. Как и в английской грамматике, эти структуры называются *условиями*.

Если вы будете использовать точки вместо точек с запятой, то вы получите такие ошибки, как drop.erl:5: function fall_velocity/2 already defined.

Имея эти условия, вы можете рассчитать скорости для объектов, падающих на определенное расстояние на Земле, на поверхности Луны и на Марсе:

```
1> c(drop).
{ok,drop}
2> drop:fall_velocity(earth,20).
19.79898987322333
3> drop:fall_velocity(moon,20).
8.0
4> drop:fall_velocity(mars,20).
12.181953866272849
```

Вы быстро обнаружите, что атомы являются критически важным компонентом для написания читаемого Erlang кода.

Атомарные значения логических данных

Два атома в Erlang обладают особыми свойствами: `true` и `false`. Они представляют логические значения с одноимёнными именами. Erlang вернет эти атомы, если вы попросите его сравнить две величины:

```
1> 3<2.
false
2> 3>2.
true
3> 10 == 10.
true
```

У Erlang также есть специальные операторы, которые работают с этими атомами (и сравнениями, которые используют эти атомы):

```
1> true and true.
true
2> true and false.
false
3> true or false.
true
4> false or false.
false
5> true xor false.
true
6> true xor true.
false
7> not true.
false
```

Операторы `and`, `or`, и `xor` принимают два аргумента. Для `and` результат `true` тогда и только тогда, когда два аргумента `true`. Для `or` результат равен `true`, если хотя бы один из аргументов имеет значение `true`. Для `xor`, исключающее или, результат равен `true`, если

один, но не оба аргумента имеют значение `true`. Во всех остальных случаях они возвращают `false`. Если вы сравниваете выражения более сложные, чем `true` и `false`, было бы разумно поместить их в скобки.

Есть два дополнительных оператора для ситуаций, когда вы не хотите или не должны проверять все аргументы. Оператор `andalso` ведет себя так, что анализ второго аргумента будет происходить если только первый являлся истинным. Оператор `orelse` будет анализировать второй аргумент, только если первый аргумент имеет значение `false`.

Оператор `not` проще, так как принимает только один аргумент. Оператор превращает `true` в `false` и `false` в `true`. В отличие от других логических операторов, которые находятся между своими аргументами, этот оператор ставится перед своим единственным аргументом.

Если вы попытаетесь использовать эти операторы с любыми другими атомами, результатом будет исключение (неправильный аргумент).

Существуют и другие атомы, которые часто имеют общепринятое значение, например, такие как `ok` и `error`, но это скорее соглашения, чем формальная часть языка.

Охранные выражения

Расчеты fall_velocity работают довольно хорошо, но есть одина странность: если функция получает отрицательное значение для расстояния, функция квадратного корня (sqrt) в расчете выбрасывает исключение:

```
5> drop:fall_velocity(earth,-20).
** exception error: bad argument in an arithmetic expression
       in function math:sqrt/1
              called as math:sqrt(-392.0)
       in call from drop:fall_velocity/2 (drop.erl, line 4)
```

Поскольку вы не можете вырыть яму на 20 метров вниз, бросить объект в низ и удивляться результату. Тем не менее, может быть более элегантно, по крайней мере, вызвать другой вид ошибки.

В Erlang вы можете указать, какие данные функция должна принимать называется *охранными выражениями*. В коде они начинаются с ключевого слова when, позволяют точно настроить сопоставление с образцом на основе содержимого аргументов, а не только их формы. Охранные выражения должны оставаться простыми. Могут использовать только очень немного встроенных функций и они ограничены требованием, что они анализируют только данные, без каких-либо побочных эффектов, но они все же могут преобразовать ваш код.

Вы можете найти список функций, которые могут быть использованы в охранных выражениях в Приложении А.

Охранные выражения выполняют свои выражения и проверяют их на истинность, как описано ранее, и выбирается первый, который имеет положительный результат. Это означает, что вы можете написать, `when true`, для охранного выражения из списка подобных или заблокировать некоторый код, который вы не хотите вызывать (пока), с помощью `when false`.

С помощью охранного выражения вы можете оградить тело функции от попаданий в него отрицательных чисел при вычислении квадратного корня, добавив охранные выражения в условия функции `fall_velocity`, как показано в Примере 3-2, который вы можете найти в *ch03/ex2-guards*.

Пример 3-2. Добавление охранных выражений в условия функций

```
-module(drop).
-export([fall_velocity/2]).

fall_velocity(earth, Distance) when Distance >= 0 -> math:sqrt(2 * 9.8 * Distance);
fall_velocity(moon, Distance) when Distance >= 0 -> math:sqrt(2 * 1.6 * Distance);
fall_velocity(mars, Distance) when Distance >= 0 -> math:sqrt(2 * 3.71 * Distance).
```

В Erlang оператор «больше или равно» >=, а «меньше чем равно» =<. Не делайте их похожими на стрелки!

Выражение `when` описывает условие или набор условий в заголовке функции. В этом случае условие простое: расстояние должно быть больше или равно нулю. Если вы скомпилируете этот код и вызовите функцию с отрицательным значением расстояния, результат будет другим:

```
5> drop:fall_velocity(earth,-20).
** exception error: no function clause matching
       drop:fall_velocity(earth,-20) (drop.erl, line 12)
```

Благодаря охранным выражениям Erlang не находит условие, которое подходило бы для использования отрицательного входного значения. Конечно, сообщение об ошибке может и не показаться существенным улучшением, но всё такая реакция «нет подходящего условия» лучше, чем «ошибка в формуле».

Более понятным, хотя и все же простым, использованием охранных выражений может быть код, который возвращает абсолютное значение. Да, Erlang имеет встроенную функцию `abs/1` для этого и Пример 3-3 поясняет, как она работает.

Пример 3-3. Расчет абсолютной величины с помощью охранных выражений

```
-module(mathdemo).
-export([absolute_value/1]).

absolute_value(Number) when Number < 0 -> -Number;

absolute_value(Number) when Number == 0 -> 0;

absolute_value(Number) when Number > 0 -> Number.
```

Когда `mathdemo:absolute_value` вызывается с отрицательным (меньше нуля) аргументом, Erlang вызывает первое условие, которое возвращает полозительное значение

отрицательного аргумента. Когда аргумент равен (==) нулю, Erlang вызывает второе условие, возвращая 0. Наконец, когда аргумент положительный, Erlang вызывает третье условие, просто возвращая число. (Первые два предложения обработали все, что не является положительным, поэтому защита в последнем предложении не нужна и исчезнет в примере 3-4.)

```
1> c(mathdemo).
{ok,mathdemo}
2> mathdemo:absolute_value(-20).
20
3> mathdemo:absolute_value(0).
0
4> mathdemo:absolute_value(20).
20
```

Это может показаться громоздким способом расчета. Но не волнуйтесь в Erlang есть более простые логические переключатели, которые вы можете использовать внутри функций. Однако такая защита крайне важна для выбора между условиями функций, что будет особенно полезно, когда вы начнете работать с рекурсией в Главе 4.

Erlang просматривает условия функций в том порядке, в котором вы их перечислите и останавливается на первом, который ему соответствует. Если вы обнаружите, что ваша вводимая информация срабатывает на не том условии, вы можете изменить порядок условий или откорректировать охранные выражения.

Кроме того, когда ваше охранное выражение проверяет только одно значение, вы можете легко переключиться на использование сопоставления с образцом вместо него. Эта функция `absolute_value` в примере 3-4 делает то же самое, что и в примере 3-3.

Пример 3-4. Вычисление абсолютного значения с охранным выражение и сопоставлением с образцом

```
absolute_value(Number) when Number < 0 -> - Number;
absolute_value(0) -> 0;
absolute_value(Number) -> Number.
```

В этом случае вам решать, предпочитаете ли вы более простую форму или сохраняете параллельный подход.

У вас может быть несколько логических выражений в одном охранном выражении. Если вы разделяете их точками с запятой (`;`), это работает как оператор ИЛИ, и завершается успешно, если любое из этиз сравнений завершается успешно. Если же вы разделяете их запятыми (`,`), это работает как оператор AND, и все они должны быть положительными, чтобы условие сработало.

Используйте подчеркивая, чтобы игнорировать значение

Охранные выражения позволяют вам указать более точную обработку входящих аргументов. Иногда вам может понадобиться обработка, которая менее точна. Не каждый аргумент необходим для каждой операции, особенно когда вы начинаете передавать сложные структуры данных. Вы можете создавать переменные для аргументов, а затем никогда не использовать их, но вы получите предупреждения от компилятора (который подозревает, что вы, должно быть, допустили ошибку), и вы можете быть сбиты с толку других людей, использующих ваш код, которые удивятся, обнаружив, что ваш код заботится только о половине аргументов ими были получены.

Например, вы можете решить, что вас не интересует, какая планета (для объекта планетарной массы, включая планеты, планеты-карлики и спутники) указывает пользователь вашей функции скорости, и вы просто используете значение гравитации для планеты Земля. Затем вы можете написать что-то вроде Примера 3-5 из *ch03/ex3-underscore*.

Пример 3-5. Объявление переменной, а затем ее игнорирование

```
-module(drop).
-export([fall_velocity/2]).

fall_velocity(Planemo, Distance) -> math:sqrt(2 * 9.8 * Distance).
```

Этот код скомпилируется, но вы получите предупреждение, и если вы попытаетесь использовать его, скажем, для Марса, вы получите неправильный ответ для Марса.

Для Марса результат должен быть больше 12, а не 19, так что компилятор был прав, чтобы вас предупреждал.

Однако в других случаях вас действительно могут волновать только некоторые аргументы. В этих случаях вы можете использовать простое подчеркивание (_). Подчеркивание выполняет два действия: оно сообщает компилятору не беспокоить вас и сообщать любому, кто читает ваш код, что вы не собираетесь использовать этот аргумент. На самом деле, Erlang даже не позволит вам это сделать. Вы можете попытаться присвоить значения подчеркиванию, но Erlang не вернет их вам. Он считает, что подчеркивание не связано ни с одним аргументом:

```
3> _ = 20.
20
4> _.
* 1: variable '_' is unbound
```

Если вы действительно хотите, чтобы ваш код был ориентирован на работу с значением гравитации Землю и игнорировал любые иные значения (других планет), вы могли бы вместо этого написать что-то вроде Примера 3-6.

Пример 3-6. Умышленное игнорирование аргумента с подчеркиванием

```
-module(drop2).
-export([fall_velocity/2]).

fall_velocity(_, Distance) -> math:sqrt(2 * 9.8 * Distance).
```

На этот раз не будет никакого предупреждения компилятора, и любой, кто взглянет на код, будет знать, что первый аргумент не будет использоваться внутри тела функции.

Вы можете использовать подчеркивание несколько раз, чтобы игнорировать несколько аргументов. Это соответствует любому значению для сопоставления с образцом и никогда не связывается с входными данными, поэтому никогда не возникает конфликта.

Вы также можете использовать подчеркивание, в качестве первого символа переменной - например, `_Planemo` – и компилятор не предупредит, если вы никогда не будете использовать эти переменные. Эти переменные будут связаны с передаваемыми в функцию даными и вы можете ссылаться на них позже в своем коде, если передумаете захотие. Однако если вы используете одно и то же имя переменной более одного раза в

наборе аргументов, даже если имя переменной начинается с подчеркивания, вы получите сообщение об ошибке от компилятора за попытку дважды связать одно и то же имя.

Усложняем код – добавляем структуру. Кортежи

Кортежи Erlang позволяют объединять несколько элементов в один составной тип данных. Это упрощает передачу сообщений между компонентами, позволяя создавать собственные сложные типы данных по мере необходимости. Кортежи могут содержать любые данные Erlang, включая числа, атомы, другие кортежи, а также списки и строки, с которыми вы познакомитесь в последующих главах.

Сами кортежи – это просто группа предметов, окруженая фигурными скобками:

Кортежи могут содержать 1 элемент, или они могут содержать 100. Два-пять кажутся типичными (и полезными, и читаемыми). Часто (но не всегда) атом в начале кортежа указывает, для чего он на самом деле, предоставляя неформальный идентификатор сложной информационной структуры, хранящейся в кортеже.

Erlang включает редко используемые встроенные функции, которые дают вам доступ к содержимому кортежа для каждого элемента в отдельности. Вы можете получить значения элементов с помощью функции элемента, установить значения в новом кортеже с помощью функции `setelement` и узнать, сколько элементов в кортеже с помощью функции `size` или `tuple_size`.

```
2> Tuple = {earth, 20}.
{earth,20}
3> element(2, Tuple).
20
4> NewTuple = setelement(2, Tuple, 40).
{earth,40}
5> tuple_size(NewTuple).
2
```

Ваш код станет более понятным, если вы будете использовать кортежи в шаблонах при сопоставлении с образцом.

Кортежи в шаблонах при сопоставлении с образцом

Кортежи упрощают упаковку нескольких аргументов в один контейнер и позволяют принимающей функции решать, что с ними делать. Сопоставление с образцом в кортежах очень похоже на сопоставление с образцом с атомами, за исключением того, что, конечно, есть пара фигурных скобок вокруг каждого набора аргументов. Пример 3-7, который вы найдете в *ch03/ex4-tuples*, демонстрирует это.

Пример 3-7. Инкапсуляция аргументов в кортеже

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

fall_velocity({earth, Distance}) -> math:sqrt(2 * 9.8 * Distance);

fall_velocity({moon, Distance}) -> math:sqrt(2 * 1.6 * Distance);

fall_velocity({mars, Distance}) -> math:sqrt(2 * 3.71 * Distance).
```

Обратите внимание, что мы изменили арность функций. `fall_velocity/1` вместо `fall_velocity/2`, потому что кортеж является одним аргументом. Такая версия работает так же, как предыдущая, но требует дополнительных фигурных скобок, когда вы вызывате её.

Зачем вам использовать эту форму, когда она требует дополнительного набора текста? Использование кортежей открывает больше возможностей. Другой код ваших программ может упаковывать разные структуры в кортежи – больше аргументов, разные атомы, даже функции, созданные с помощью анонимных функций `fun()`. Передача одного кортежа, а не множества аргументов, дает Erlang большую гибкость, особенно когда вы передаете сообщения между различными процессами.

Обработка кортежей

Существует много способов обработки кортежей, а не только простое сопоставление с образцом, показанное в примере 3-7. Если вы получаете кортеж, как переменную, вы можете делать с ним много разных манипуляций. Например, использовать кортеж для перехода к закрытой версии функции. Эта часть примера 3-8 может показаться вам знакомой, поскольку она аналогична функции `fall_velocity/2` в Примере 3-2 (полную версию кода примера вы можете найти в *`ch03/ex5-tuplesMore`*).

Пример 3-8. Инкапсуляция аргументов в кортеже и передача их частной функции

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

fall_velocity({Planemo, Distance}) -> fall_velocity(Planemo, Distance).

fall_velocity(earth, Distance) when Distance >= 0 -> math:sqrt(2 * 9.8 * Distance);
fall_velocity(moon, Distance) when Distance >= 0 -> math:sqrt(2 * 1.6 * Distance);
fall_velocity(mars, Distance) when Distance >= 0 -> math:sqrt(2 * 3.71 * Distance).
```

Директива `-export` предоставляет доступ извне модуля только функцию `fall_velocity/1`. Однако внутри модуля доступна функция `fall_velocity/2`. Это может быть удобно, если вы не хотите представлять доступ к внутренним функциям вашего модуля.

Если вы вызываете эту функцию, то, обратите внимание, необходимо использовать фигурные скобки для описания кортежа. Функция `fall_velocity/1` вызывает закрытую функцию `fall_velocity/2` внутри модуля. Она возвращает результат в `fall_velocity/1`, которая, в свою очередь, возвращает результирующее значение.

```
1> c(drop).
{ok,drop}
2> drop:fall_velocity({earth,20}).
19.79898987322333
3> drop:fall_velocity({moon,20}).
8.0
4> drop:fall_velocity({mars,20}).
12.181953866272849
```

Есть несколько разных способов извлечь данные из кортежа. Вы можете ссылаться на компоненты кортежа по номеру, используя встроенную функцию `element`, которая принимает позицию элемента в кортеже и кортеж в качестве аргументов. Первый компонент кортежа можно получить начиная с 1, второй в позиции 2 и так далее.

```
fall_velocity(Where) -> fall_velocity(element(1,Where) , element(2,Where)).
```

Вы также можете немного разделить кортеж на составляющие с помощью сопоставления с образцом после получения переменной:

```
fall_velocity(Where) ->
       {Planemo, Distance} = Where,
       fall_velocity(Planemo, Distance).
```

Эта функция имеет более одной строки. Обратите внимание, что действия разделены запятыми, и только последняя строка заканчивается точкой. Результатом этой последней строки будет значение, которое возвращает функция.

Сопоставление с образцом немного отличается. Функция принимает кортеж в качестве аргумента и присваивает кортеж имеющих туже структуру, что и данные в переменной `Where`. (Если `Where` не является кортежем, функция завершится с ошибкой). Извлечение содержимого этого кортежа (поскольку мы знаем его структуру), может быть выполнено с помощью сопоставления с образцом внутри функции. Переменные `Planemo` и `Distance` будут привязаны к значениям, содержащимся в кортеже `Where`, и затем могут быть использованы при вызове fall_velocity/2.

Подробнее о работе с атомами, кортежами и сопоставлением с образцом вы можете узнать в Главе 2 «Erlang Programming» (O’Reilly); Глава 2 «Programming Erlang» (Pragmatic); Разделы 2.2 и 2.4 Erlang и «Erlang and OTP in Action» (Manning); и в главах 1 и 3 «Learn You Some Erlang For Great Good!» (No Starch Press).

Глава 4. Управление и рекурсия

Пока что язык Erlang кажется логичным, но довольно простым. Сопоставление с образцом контролирует поток через программу, а запросы, соответствующие форме, возвращают определенные ответы. Хотя этого достаточно для выполнения многих задач, бывают ситуации, когда вам нужны более мощные опции, особенно когда вы начинаете работать с более крупными и сложными структурами данных.

Логика внутри функций

Сопоставление с образцом и охранные выражения – это мощные инструменты, но бывают случаи, когда гораздо проще проводить некоторые сравнения внутри тела функции вместо создания новых функций. Разработчики Erlang согласились и создали две конструкции для анализа условий внутри функций: выражение `case` и менее часто используемое выражение `if`.

Конструкция `case` позволяет вам использовать сопоставление с образцом и охранные выражения внутри тела функции. Она наиболее ясно читается, когда одно значение (или набор значений) необходимо сравнить с несколькими возможными вариантами. Конструкция `if` анализирует только одно или серию охранных выражений без сопоставления с образцом. Конструкция `if` может создавать более читаемый код в ситуациях, когда множественные возможности определяются комбинациями различных значений.

Обе конструкции возвращают значение, которое может присвоить переменной.

Использование `case`

Конструкция case позволяет выполнять сопоставление с образцом внутри тела функции. Если вы обнаружили, что код примера 3-2 трудно читать, вы можете предпочесть создать версию, похожую на пример 4-1, которую вы можете найти в *`ch04/ex1-case`*.

Пример 4-1. Перемещение сопоставления с образцом внутрь функции

```
-module(drop).
-export([fall_velocity/2]).

fall_velocity(Planemo, Distance) when Distance >= 0 ->

        case Planemo of
                earth -> math:sqrt(2 * 9.8 * Distance);
                moon  -> math:sqrt(2 * 1.6 * Distance);
                mars  -> math:sqrt(2 * 3.71 * Distance) % no closing period!
        end.
```

Конструкция `case` будет сравнивать атом в `Planemo` со значениями, перечисленными в списке, двигаясь последовательно по списку. Анализ прекратится после нахождения соответствующего атома. Конструкция `case` вернет результат различных вычислений, основанных на том, какой атом используется, и поскольку конструкция `case` возвращает последнее значение в предложении функции, функция также вернет это значение.

Вы можете использовать подчеркивание (_) для сопоставления с образцом, если вы хотите, чтобы выбор соответствовал «всему остальному». Однако вы всегда должны ставить его последним в списке сопоставлений с образцом, иначе следующие за ним атому не будут игнорированы.

Результаты работы кода должны быть вам уже знакомы:

```
1> c(drop).
{ok,drop}
2> drop:fall_velocity(earth,20).
19.79898987322333
3> drop:fall_velocity(moon,20).
8.0
4> drop:fall_velocity(mars,20).
12.181953866272849
5> drop:fall_velocity(mars,-20).
** exception error: no function clause matching
drop:fall_velocity(mars,-20) (drop.erl, line 5)
```

Конструкция `case` переключается между космическими объектами, в то время как охранные выражения в определениях функций отбрасывают отрицательные расстояния, что приводит (правильно) к ошибке в строке 5. Таким образом, охранное выражение только одно.

Вы также можете использовать возвращаемое значение из конструкции `case`, чтобы уменьшить количество дублирующегося кода и сделать логику вашей программы более понятной. В этом случае единственная разница между вычислениями для Земли, Луны и Марса – это гравитационная постоянная. В примере 4-2, который вы можете найти в *`ch04/ex2-case`*, показано, как заставить конструкцию case возвращать гравитационную постоянную для использования в конце одного вычисления.

Пример 4-2. Использование возвращаемого значения конструкции case для создания более ясной логики функции

```
-module(drop).
-export([fall_velocity/2]).

fall_velocity(Planemo, Distance) when Distance >= 0 ->
       Gravity = case Planemo of
              earth -> 9.8;
              moon -> 1.6;
              mars -> 3.71 % note comma - function isn't done yet
              end,
       math:sqrt(2 * Gravity * Distance).
```

На этот раз переменная Gravity устанавливается на возвращаемое значение конструкции case. Не забудьте о запятую после end! Эта функция еще не выполнена! Запятые позволяют разделять конструкции внутри объявлений функций. Далее следует более читаемая формула `math:sqrt(2 * Gravity * Distance)`. Это последняя строка функции и полученное значение будет возвращаемым значением.

Вы также можете использовать охранные выражения в оператором case, как показано, возможно, менее элегантно, как в примере 4-3, который находится в *`ch04/ex3-case`*. Это могло бы иметь больше смысла, если бы существовали разные планеты с разными правилами анализа расстояний.

Пример 4-3. Размещение охранных выражений внутри case

```
-module(drop).
-export([fall_velocity/2]).

fall_velocity(Planemo, Distance)  ->
       Gravity = case Planemo of
              earth when Distance >= 0 -> 9.8;
              moon when Distance >= 0 -> 1.6;
              mars when Distance >= 0 -> 3.71 % note comma - function isn't done yet
       end,
       math:sqrt(2 * Gravity * Distance).
```

Это приводит к аналогичным результатам, за исключением того, что сообщение об ошибке в конце изменяется с `no function clause matching drop:fall_velocity(mars, -20)` на `no case clause matching mars in function drop:fall_velocity/2`:

```
1> c(drop).
{ok,drop}
2> drop:fall_velocity(mars,20).
12.181953866272849
3> drop:fall_velocity(mars,-20).
** exception error: no case clause matching mars
in function drop:fall_velocity/2 (drop.erl, line 6)
```

Ошибка весьма точно сообщает о проблеме. Конструкция case пытается сопоставить `mars`, но вводит в заблуждение, потому что проблема не в атоме `mars`, а в охранном выражении, который проверяет переменную `Distance`. Если сообщение Erlang сообщит, что case не имеет нужного образца для сопоставления, проверьте ваши охранные выражения.

Если так, иначе так

Конструкция `if`, в целом, похожа на инструкцию `case`, но без сопоставления с образцом. Упрощения этого оператора (чтобы условие срабатывало при любых значениях – введите охранное выражение `true` в конце.

Предположим, например, что точность функции `fall_velocity` слишком велика. Вместо реальной скорости вы хотели бы описать скорость, получаемую при падении с башни заданной высоты. Вы можете добавить конструкцию `if`, которая делает это, к более раннему коду из примера 4-2, как показано в примере 4-4, в *`ch04/ex4-if`*.

Пример 4-4. Добавление конструкции if для преобразования чисел в атомы

```
-module(drop).
-export([fall_velocity/2]).

fall_velocity(Planemo, Distance) when Distance >= 0 ->
       Gravity = case Planemo of
              earth -> 9.8;
              moon -> 1.6;
              mars -> 3.71
              end,
       Velocity = math:sqrt(2 * Gravity * Distance),

       if
              Velocity == 0 -> 'stable';
              Velocity < 5 -> 'slow';
              Velocity >= 5, Velocity < 10 -> 'moving';
              Velocity >= 10, Velocity < 20 -> 'fast';
              Velocity >= 20 -> 'speedy'
       end.
```

На этот раз конструкция `if` возвращает значение (атом, описывающий скорость) на основе множества охранных выражений, которые оно включает. Поскольку это значение является

последним значением, возвращаемым в функции, оно становится возвращаемым значением функции.

Запятые в охранных выражениях `if` ведут себя как оператор and.

Результаты немного отличаются от прошлых:

```
1> c(drop).
{ok,drop}
2> drop:fall_velocity(earth,20).
fast
3> drop:fall_velocity(moon,20).
moving
4> drop:fall_velocity(mars,20).
fast
5> drop:fall_velocity(earth,30).
speedy
```

Вы можете использовать значение, возвращаемое оператором `if`, сохранив это значение в переменной. Пример 4-5, в *`ch04/ex5-if`*, отправляет предупреждение (в данном случае в оболочку Erlang), если вы уронили объект слишком быстро.

Пример 4-5. Отправка дополнительного предупреждения, если скорость слишком высока

```
-module(drop).
-export([fall_velocity/2]).

fall_velocity(Planemo, Distance) when Distance >= 0 ->
        Gravity = case Planemo of
                earth -> 9.8;
                moon -> 1.6;
                mars -> 3.71
        end,
        Velocity = math:sqrt(2 * Gravity * Distance),
        Description = if
                Velocity == 0 -> 'stable';
                Velocity < 5 -> 'slow';
                (Velocity >= 5) and (Velocity < 10) -> 'moving';
                (Velocity >= 10) and (Velocity < 20) -> 'fast';
                Velocity >= 20 -> 'speedy'
                end,

if
        (Velocity > 40) -> io:format("Look out below!~n");
        true -> true
end,
Description.
```

Новое (второе) условие `if` проверяет переменную `Velocity`, чтобы увидеть, не превышает ли она `40`. Если это так, она вызывает `io:format`, что создает побочный эффект: сообщение на экране. Однако каждый if должен найти какое-то истинное утверждение, иначе он сообщит об ошибке в тех случаях, когда ничего не совпадает. Здесь вы можете добавить явное соответствие случая, когда скорость меньше или равна `40`. Однако во многих случаях это не имеет значения. Строка `true -> -` это универсальное средство, которое возвращает `true` независимо от того, что достигнуто. После завершения if в единственная строка `Description`. возвращает содержимое переменной `Description`.

Подход с использованием ловушек работает в тех случаях, когда вы хотите протестировать только подмножество случаев среди сложного набора возможностей. Однако в таких простых случаях, как в этом примере, возможно, будет удобнее создать более явный тест.

Функция выдает дополнительный результат - сообщение - когда расстояние достаточно велико (а сила тяжести `Planemo` достаточно сильна), чтобы создать скорость, превышающую 40 метров в секунду:

```
> c(drop).
{ok,drop}
2> drop:fall_velocity(earth,10).
fast
3> drop:fall_velocity(earth,200).
Look out below!
speedy
```

Присвоение переменной в case и if

Каждый значение, возвращаемое выражениями case или if, имеет возможность привязать значения к переменным. Обычно это замечательно, но может позволить создавать нестабильные программы, назначая разные переменные в разных предложениях. Это может выглядеть примерно так, как в примере 4-6, который вы можете найти в *ch04/ex6-broken*.

Пример 4-6. Некомплируемый код

```
-module(broken).
-export([bad_if/1]).

bad_if(Test_val) ->
        if
                Test_val < 0 -> X = 1;
                Test_val >= 0 -> Y = 2
        end,
        X+Y.
```

Теоретически, после выхода из case или if программа может аварийно завершить работу из-за несвязанных переменных. Тем не менее, Erlang не позволит вам зайти так далеко:

```
1> c(broken).
broken.erl:11: variable 'X' unsafe in 'if' (line 6)
broken.erl:11: variable 'Y' unsafe in 'if' (line 6)
error
```

Ошибки компиляции появляются там, где ваша программа использует переменные. Компилятор Erlang дважды проверяет правильность определения переменных, которые он собирается использовать. Это не позволит вам скомпилировать что-то неработающее.

Вы *можете* связать переменные в конструкции `if` или `case`. Однако вы должны определить все переменные в каждом предложении. Если вы определяете только одну переменную, гораздо проще связать возвращаемое значение предложения `if` или `case` с переменной, а не определять эту переменную в каждом теле выражения.

Самый корректный побочный эффект: io:format

Вплоть до Примера 4-5 все примеры Erlang, которые вы видели, фокусировались на одном пути через группу функций. Вы вводите аргумент или аргументы и получаете возвращаемое значение. Этот подход – самый простой способ сделать что-то: вы можете рассчитывать на то, что раньше работало, будет снова работать, потому что нет возможности испортить систему.

Пример 4-5 вышел за пределы этой модели, создав побочный эффект, который сохранится после завершения функции. Побочным эффектом является просто сообщение, которое появляется в оболочке (или в стандартном выводе, когда вы запускаете Erlang вне оболочки). Приложения, которые обмениваются информацией с несколькими пользователями или хранят информацию дольше, чем короткий цикл обработки, будут нуждаться в более мощных побочных эффектах, таких как хранение информации в базах данных.

Лучшие практики Erlang предполагают использование побочных эффектов только тогда, когда это действительно необходимо. Например, приложению, предоставляющему интерфейс к базе данных, действительно нужно читать и писать эту базу данных. Приложение, которое взаимодействует с пользователями, должно будет отображать информацию на экране (или другом интерфейсе), чтобы пользователи могли понять, что они должны делать.

Побочные эффекты также чрезвычайно полезны для отслеживания логики, когда вы только начинаете. Самый простой способ увидеть, что делает программа, прежде чем вы научитесь использовать встроенные в процессы инструменты отслеживания и отладки Erlang – это заставить программу сообщать о своем состоянии в моменты, которые вы считаете интересными. Это не та функция, которую вы хотите оставить в коде доставки, но когда вы начинаете, она может дать вам понятное представление о поведении вашего кода.

Функция `io:format` позволяет вам отправлять информацию на консоль или, когда вы в конечном итоге выполняете код вне консоли, в другие места. А пока вы будете использовать его для отправки сообщений из программы на консоль. В примере 4-5 показан простейший способ использования `io:format` просто напечатать сообщение в двойных кавычках:

```
io:format("Look out below!~n");
```

~n представляет новую строку, сообщающую консоли, что нужно начинать любые новые сообщения, которые она отправляет в начале следующей строки. Это делает ваши результаты более аккуратными.

Более типичный способ использования `io:format` включает два аргумента: строку форматирования в двойных кавычках и список значений, которые могут быть включены в строку. `~w` позволяет включать содержание без отступов и форматирования. В этом случае (который вы можете увидеть в примере *`ch04/ex7`*) это может выглядеть следующим образом:

```
io:format("Look out below! ~w is too high.~n", [Distance]);
```

или:

```
io:format("Look out below! ~w is too high on ~w.~n", [Distance, Planemo]);
```

`io:format/2` предлагает много вариантов форматирования, кроме ~w и ~n. Вы столкнетесь с ними по мере необходимости, но если вы нетерпеливы, в Приложении A есть список. Вы также можете изучить раздел регистрации ошибок в Главе 9, если вы обнаружите, что используете `io:format` для задач, которые могут помочь решить более подходящие инструменты ведения журналов.

Erlang категорически запрещает операции, которые могут вызывать побочные эффекты в охранных выражениях. Если бы побочные эффекты были разрешены для охранных выражений, то всякий раз, когда оно выполнялось – возвращало ли оно истину или ложь, то мог бы происходить побочный эффект. `io:format` вряд ли сделает что-то ужасное, но эти правила означают, что он также заблокирован для использования в выражениях.

Простая рекурсия

Поскольку переменные не могут изменять значения, основным инструментом, который вы будете использовать для повторения действий, является рекурсия: наличие самого вызова функции до тех пор, пока она (надеюсь) не достигнет своего завершения. Это может показаться сложным, но это не обязательно.

Есть два основных вида полезной рекурсии. В некоторых ситуациях вы можете рассчитывать на рекурсию, чтобы достичь естественного конца. В процессе не хватает элементов для работы или он достигает естественного предела. В других ситуациях нет естественного конца, и вам нужно отслеживать результат, чтобы завершить процесс. Если вы сможете овладеть этими двумя основными формами, вы сможете создавать гораздо более сложные варианты.

Существует третья форма, в которой рекурсивные вызовы никогда не заканчиваются. Это называется бесконечным циклом и лучше всего известно как ошибка, которую вы хотите избежать. Однако, как вы увидите в Главе 8, даже бесконечные циклы могут быть полезны.

Обратный отсчет

Самая простая модель рекурсии с естественным пределом – это обратный отсчет, аналогичный модели, используемой для ракет. Вы начинаете с большого числа и отсчитываете до нуля. Когда вы достигаете нуля, все готово (и ракета взлетает, если она есть).

Чтобы реализовать это в Erlang, вы передадите начальный номер функции Erlang. Если число больше нуля, оно объявляет номер и вызывает себя с номером минус один в качестве аргумента. Если число равно нулю (или меньше), возвращается blastoff!, что является завершением функции. Пример 4-7, в *`ch04/ex8-countdown`*, показывает один из способов сделать это.

Пример 4-7. Обратный отсчет

```
-module(count).
-export([countdown/1]).
```

```
countdown(From) when From > 0 ->
                  io:format("~w!~n", [From]),
                  countdown(From-1);

countdown(From) ->
                  io:format("blastoff!~n").
```

Последнее предложение может иметь охранное выражение `From =< 0`, но оно было бы полезно только для читателей кода. Ненужные охранные выражения могут привести к странным ошибкам, поэтому краткость, вероятно, является лучшим вариантом здесь, хотя вы получите предупреждение о том, что `From` не используется в последнем предложении. Вот пример работы этой функции:

```
1> c(count).
count.erl:9: Warning: variable 'From' is unused
{ok,count}
2> count:countdown(2).
2!
1!
blastoff!
ok
```

В первый раз Erlang выбрал первое предложение обратного отсчета (`From`), передав ему значение 2. Это предложение напечатало 2, плюс восклицательный знак и символ новой строки, а затем снова вызвало функцию обратного отсчета, передав ей значение 1. Это снова вызвало первый пункт. Он напечатал 1, плюс восклицательный знак и символ новой строки, а затем снова вызвал функцию обратного отсчета, на этот раз передав значение 0.

Значение `0` вызвало второе предложение, которое напечатало `blastoff!` и работа функций закончилась. После запуска трех значений через один и тот же набор кода, функция приходит к четкому выводу.

Вы также можете реализовать этот вывод с помощью оператора `if` внутри одного из условий `countdown(From)`. Это нетипично в Erlang. Я нахожу охранные выражения более читабельными в этих случаях, но у вас может быть иное мнение.

Прямой отсчёт

Подсчет сложнее, потому что нет естественной конечной точки, поэтому вы не можете смоделировать свой код в примере 4-7. Подход Erlang к переменным с одним присваиванием исключает некоторые подходы, но есть и другой способ сделать это, используя аккумулятор. Аккумулятор – это дополнительный аргумент, который отслеживает текущий результат прошлой работы, передавая его обратно в рекурсивную функцию. (У вас может быть несколько аргументов-накопителей, если вам нужно, хотя их часто достаточно.) Пример 4-8, который вы можете найти в *`ch04/ex9-countup`*, показывает, как добавить функцию отсчета в модуль `count`, который позволяет Erlang сосчитать числа.

Пример 4-8. Прямой отсчет

```
-module(count).
-export([countdown/1, countup/1]).

countup(Limit) ->
```

```
        countup(1, Limit).

countup(Count, Limit) when Count =< Limit ->
        io:format("~w!~n", [Count]),
        countup(Count+1, Limit);

countup(Count, Limit) ->
        io:format("Finished.~n").
...
```

Это приводит к следующим результатам:

```
1> c(count).
{ok,count}
2> count:countup(2).
1!
2!
Finished.
ok
```

Директива `export` делает видимой функцию `countup/1` (так же как и более ранний `countdown/1`, который вы найдете в примере кода).

Функция `countup/2`, которая выполняет большую часть работы, остается закрытой и не экспортируется. Это не обязательно. Вы можете сделать её публичным, если хотите поддерживать подсчет между произвольными значениями, но это обычная практика Erlang. Хранение рекурсивных внутренних функций в тайне снижает вероятность того, что кто-то будет злоупотреблять ими в целях, для которых они не очень подходит. В этом случае это не имеет значения, но может иметь большое значение в других более сложных ситуациях, особенно при изменении данных.

Когда вы вызываете `countup/1`, она вызывает `countup/2` с аргументом `1` (для текущего счетчика) и значением `Limit`, которое вы указали для верхнего предела.

Если текущий счетчик меньше или равен верхнему пределу, первое предложение функции `countup/2` сообщает текущее значение счетчика в с помощью `io:format`. Затем он снова вызывает себя, увеличивая количество на единицу, но оставляя предел в покое.

Если текущий счет превышает верхний предел, он не проходит защиту первого предложения, поэтому второе предложение включается, сообщает «Finished» и завершается.

Охранных выражений здесь достаточно, чтобы избежать бесконечных петель. Вы можете ввести ноль, отрицательные числа или десятичные числа в качестве аргументов для `countup/1`, и он будет аккуратно завершен. Однако вы можете столкнуться с серьезными проблемами, если ваш тест завершения основан на == или =:= для более точного сравнения, а не на >= или =< для грубого сравнения.

Рекурсия с возвращаемыми значениями

Примеры подсчета просты – они демонстрируют, как работает рекурсия, просто отбрасывая возвращаемые значения. Существуют возвращаемые значения – вызовы `io:format` возвращают атом в порядке, но они не очень полезны. Более типично, рекурсивный вызов функции будет использовать возвращаемое значение.

Классический рекурсивный вызов вычисляет факториалы. Факториал – это произведение всех натуральных чисел, равных или меньших аргумента. Факториал 1 равен 1; 1 само по себе дает 1. Факториал 2 равен 2; 2 × 1 дает 2. Это начинает становиться интересным в 3, где 3 × 2 × 1 - шесть. При 4, 4 × 3 × 2 × 1 равно 24, и результаты быстро увеличиваются с увеличением аргументов.

Впрочем, в этом была есть закономерность. Вы можете вычислить любой факториал, умножив целое число на факториал на единицу меньше. Это делает его идеальным вариантом для использования рекурсии, использования результатов меньших целых чисел для вычисления больших. Этот подход похож на логику обратного отсчета, но вместо простого подсчета программа собирает вычисленные результаты. Это может выглядеть как Пример 4-9, который вы найдете в *ch04/ex10-factorialdown*.

Пример 4-9. Факториал, написанный с помощью обратного отсчета

```
-module(fact).
-export([factorial/1]).

factorial(N) when N > 1->
       N * factorial(N-1);

factorial(N) when N =< 1 ->
       1.
```

Первый пункт факториала использует шаблон, описанный ранее. Первое предложение, используемое для чисел выше единицы, возвращает значение, которое является числом N, умноженным на факториал следующего целого числа вниз. Второе предложение возвращает значение 1, когда оно достигает 1. Использование = < в этом сравнении, а не ==, дает функции большую устойчивость к нецелым или отрицательным аргументам, хотя ответы, которые она возвращает, не совсем верны: факториалы действительно работают только для целых чисел от 1 или выше. Результаты:

```
1> c(fact).
{ok,fact}
2> fact:factorial(1).
1
3> fact:factorial(3).
6
4> fact:factorial(4).
24
5> fact:factorial(40).
815915283247897734345611269596115894272000000000
```

Это работает, но может быть неясно, почему это работает. Функция ведет обратный отсчет и собирает значения, но если вы хотите увидеть как это происходит, вам нужно добавить в код несколько вызовов `io:format`, как показано в примере 4-10. (Вы можете найти это в `ch04/ ex11-factorial-down-instrumented`.)

Пример 4-10. Изучение факториальных рекурсивных вызовов

```
-module(fact).
-export([factorial/1]).

factorial(N) when N > 1->
       io:format("Calling from ~w.~n", [N]),
       Result = N * factorial(N-1),
       io:format("~w yields ~w.~n", [N, Result]),
       Result;

factorial(N) when N =< 1 ->
       io:format("Calling from 1.~n"),
       io:format("1 yields 1.~n"),
       1.
```

Здесь немного больше накладных расходов. Чтобы представить результат рекурсивного вызова и все еще вернуть это значение следующему рекурсивному вызову, необходимо сохранить его в переменной, которая здесь называется `Result`. Вызов `io:format` делает видимым, какое значение дало результат. Затем, поскольку последнее значение выражения в предложении функции является возвращаемым значением, результат появляется снова. Второе предложение для `1` аналогично, за исключением того, что оно может просто сообщить, что `1` дает `1.`, потому что так будет всегда.

Когда вы скомпилируете это и запустите, вы увидите что-то вроде следующего:

```
7> fact:factorial(4).
Calling from 4.
Calling from 3.
Calling from 2.
Calling from 1.
1 yields 1.
2 yields 2.
3 yields 6.
4 yields 24.
24
```

Хотя вызовы отсчитывают значения, как подсказывает логика, сообщения о результатах не появляются до тех пор, пока не завершится обратный отсчет, а затем все они отображаются по порядку.
Причина, по которой это происходит, заключается в том, что функция вызывает не возвращаемые значения до завершения обратного отсчета. До этого времени среда исполнения Erlang создает стек кадров, соответствующих вызовам функций. Вы можете думать о кадрах, как о приостановленных версиях логики функции, ожидая ответа. Как только вызов с аргументом `1` возвращает простое значение, больше не вызывая, Erlang может разматывать эти кадры и вычислять результат. Это раскручивание представляет результаты – «X дает Y» – в том порядке, в котором кадры раскручиваются.

Это «раскручивание» также означает, что код в Примере 4-9 и Примере 4-10 не является хвостовой рекурсией. Когда Erlang встречает код, который заканчивается простым рекурсивным вызовом, он может оптимизировать обработку, чтобы избежать сохранения этого стека вызовов. Вероятно, это не имеет значения для одноразовых вычислений, но это имеет огромное значение, когда вы пишете код, который будет работать долго.

Вы можете получить хвостовую рекурсию для факториалов, применив подход к подсчету факториалов. Вы получите те же результаты (по крайней мере, для целочисленных значений), но вычисления будут работать немного по-другому, как показано в примере 4-11, в *`ch04/ex12-factorial-up`*.

Пример 4-11. Факториал, написанный с помощью метода подсчета

```
-module(fact).
-export([factorial/1]).

factorial(N) ->
        factorial(1, N, 1).

factorial(Current, N, Result) when Current =< N ->
        NewResult = Result*Current,
        io:format("~w yields ~w!~n", [Current, NewResult]),
        factorial(Current+1, N, NewResult);

factorial(Current, N, Result) ->
        io:format("Finished.~n"),
        Result.
```

Как и в примере с подсчетом, вызов основной функции, здесь `factorial/1`, вызывает закрытую функцию, `factorial/3`. У неё есть два аккумулятора. `Current` сохраняет текущую позицию в счетчике, а `Result` – ответ из предыдущего умножения. Когда значение `Current` поднимается выше предельного значения `N`, защита выходит из строя, вызывается второе предложение, и функция завершается, возвращая результат. (Вы получите предупреждения компилятора, потому что в последнем предложении не используются переменные-накопители `Current` и `N`. Вы можете игнорировать их.)

Поскольку последний вызов `factorial/3` в рекурсивном разделе сам по себе, без каких-либо сложностей для отслеживания, он хвостово рекурсивен. Erlang может минимизировать объем информации, которую он должен хранить, пока происходят все вызовы.

Расчет дает те же результаты, но вычисления выполняются в другом порядке:

```
9> fact:factorial(4).
1 yields 1!
2 yields 2!
3 yields 6!
4 yields 24!
Finished.
24
```

Хотя код отслеживает больше значений, среда выполнения Erlang работает менее интенсивно. Когда результат окончательно достигнут, дальнейших вычислений не требуется. Этот результат и является конечным результатом работы программы, и он возвращается к исходному вызову. Это также упрощает структурирование `io:format`. Если вы удалите их или закомментируете, остальная часть кода останется прежней.

Вы можете узнать больше о работе с логикой потока выполнения и рекурсией в Главе 3 «Erlang Programming» (O’Reilly); в Главе 3 «Programming Erlang» (Pragmatic); Разделах 2.6 и 2.15 «Erlang and OTP in Action» (Manning); и в Главах 3 и 5 «Learn You Some Erlang For Great Good!» (No Starch Press).

Глава 5. Взаимодействие с человеком

По причине происхождения Erlang для нужд телекоммуникационной коммутации он имеет довольно минимальный набор инструментов для взаимодействия с людьми, но и этого достаточно. Вы уже использовали некоторые из них (`io:format/1` и `io:format/2`), но есть еще несколько функций, которые вам необходимо изучить, чтобы справляться с общением с людьми, а иногда и с другими приложениями. Эта глава позволит вам создать более удобные интерфейсы для тестирования вашего кода, чем вызов функций из оболочки Erlang.

Если вас интересует рекурсией, которую вы изучили в Главе 4, вы можете перейти к Главе 6, где эта рекурсия снова будет в центре внимания.

Строки

Атомы отлично подходят для отправки сообщений внутри программы, даже сообщений, которые программист может запомнить, но на самом деле они не предназначены для взаимодействия вне контекста процессов Erlang. Если вам нужно собирать предложения или даже представлять информацию, вам нужно что-то более гибкое. Строки, последовательности символов – это структура, которая вам нужна. Вы уже немного использовали строки в качестве аргументов в двойных кавычках для функции `io:format` в Главе 4:

```
io:format("Look out below!~n");
```

Содержимое в двойных кавычках (`Look out below!~n`) является строкой. Строка – это последовательность символов. Если вы хотите включить двойную кавычку в строку, вы можете экранировать ее обратной косой чертой, например `\"`. Чтобы включить обратную косую черту, вы должны использовать `\\`. В Приложении А содержится полный список escape-символов и других параметров. Если вы создадите строку в оболочке, Erlang сообщит строку с escape-символами. Чтобы увидеть, что она должна содержать, используйте `io:format`:

```
1> X = "Quote - \" in a string. \n Backslash, too: \\ . \n".
"Quote - \" in a string. \n Backslash, too: \\ ."
2> io:format(X).
Quote - " in a string.
Backslash, too: \ .
ok
```

Если вы начинаете вводить строку и не закрываете кавычки, при нажатии клавиши Enter оболочка Erlang просто выдаст вам новую строку с тем же номером. Это позволяет вам включать строки в строки, но это может быть очень запутанным. Если вы думаете, что застряли, введите ". (двойные кавычки и точку) и это поможет.

Исторически Erlang не уделял особого внимания тексту, но если ваши программы предполагают обмен информацией с людьми, вы захотите ознакомиться с тем, как вводить и выводить информацию из строк. Это область, где вы можете потратить достаточное количество времени в оболочке, работая с различными инструментами.

Технически, строки не существуют в Erlang как тип, поскольку строки представляют собой списки символов. Однако думать о строках как о списках символов полезно лишь в нескольких ситуациях, обычно когда вы хотите обработать строку от начала до конца. Вы узнаете о списках в Главе 6, и в этом коде будет использоваться встроенная функция списков, но сейчас вы должны просто думать о строковых операциях, а не о списках.

Самым простым, как правило, является конкатенация, когда вы объединяете две строки в одну. Erlang предлагает два простых способа сделать это. Первый использует оператор ++:

```
1> "erl" ++ "ang".
"erlang"
2> A="ang".
"ang"
3> "erl" ++ A.
"erlang"
```

Другой подход использует явную строку: функцию concat/2:

```
4> string:concat("erl", "ang").
"erlang"
5> N="ang".
"ang"
6> string:concat("erl", N).
"erlang"
```

Оператор ++ обычно более удобен, потому что позволяет работать с более чем двумя аргументами без вложенных функций.

Erlang имеет возможность сокращенной записи для объединения строк. Вы можете объединить две строки, просто поместив их рядом друг с другом: `"erl"` и `"ang"` будут объединены в `"erlang"`. Однако, если вы попытаетесь объединить переменные, вы получите синтаксическую ошибку. Эта сокращенная запись имеет ограниченное применение, за исключением тех случаев, когда вы вырезаете и вставляете значения в кавычках, когда пишете свой код, и он работает не во всех контекстах.

Erlang также предлагает три варианта сравнения равенства строк: оператор ==, оператор `=:=` (точное равенство) и `string:equal/2`. Оператор ==, как правило, самый простой для этого, хотя остальные дают те же результаты:

```
7> "erl" == "erl".
true
8> "erl" == "ang".
false
9> G ="ang".
"ang"
10> G == "ang".
true
```

Erlang не предлагает функций для изменения строк, так как это плохо работает с моделью, в которой содержимое переменных не меняется. Тем не менее, он предлагает набор функций для поиска содержимого в строках и разделения или заполнения этих строк, которые вместе позволяют извлекать информацию из строки (или нескольких строк) и рекомбинировать ее в новую строку.

Если вы хотите сделать больше со своими строками, вам определенно следует изучить документацию для строки и повторных (регулярных выражений) модулей Erlang. Если строки, с которыми вы хотите работать, представляют имена файлов или каталогов, определенно изучите модуль имени файла. Если вам необходимо выполнить преобразование кодировки Unicode для строк Erlang, вам также нужно изучить модуль Unicode. (По умолчанию Erlang представляет символы с использованием значений UTF-8.)

Я работаю над созданием единого модуля-обертки, который собирает инструменты Erlang для работы со строками в одном месте. Для получения дополнительной информации посетите https://github.com/simonstl/erlang-simple-string.

Получение информации от пользователей

Многие приложения Erlang работают как оптовики – в фоновом режиме, предлагая товары и услуги розничным продавцам, которые напрямую взаимодействуют с пользователями. Однако иногда приятно иметь прямой интерфейс для кода, который немного более индивидуален, чем консоль Erlang. Вы, вероятно, не будете писать много приложений Erlang, основным интерфейсом которых является командная строка, но этот интерфейс может оказаться очень полезным, когда вы впервые попробуете свой код. (Скорее всего, если вы работаете с Erlang, вы не против использовать интерфейс командной строки.)

Вы можете смешивать ввод и вывод с логикой вашей программы, но для такого простого фасада, вероятно, имеет смысл поместить его в отдельный модуль. В этом случае модуль `ask` будет работать с удаленным модулем из примера 3-8.

Функции ввода-вывода Erlang для ввода имеют множество странных взаимодействий с оболочкой Erlang, как обсуждалось в следующем разделе. Вам повезет работать с ними в других контекстах.

Условия получения доступа

Самый простой способ создать интерфейс – интерфейс, вероятно, только для программистов – это создать для пользователей способ ввода термов Erlang с помощью `io:read/1`. Это позволяет пользователям вводить полный терм Erlang, например, атом, число или кортеж. Начальная версия этого может выглядеть как Пример 5-1, который вы можете найти в *`ch05/ex1-ask`*.

Пример 5-1. Функция получения запрашиваемого пользователем расстояние

```
-module(ask).
-export([term/0]).

term() ->
        Input = io:read("What {planemo, distance} ? >>"),
        Term = element(2,Input),
        drop:fall_velocity(Term).
```

Переменная `Input` будет установлена вызовом `io:read/1`, получая терм Erlang. Если все идет хорошо, он будет содержать кортеж типа `{ok, {mars, 20}}`, где первое значение атом `ok`, а второе значение – это терм, введенный пользователем. Извлечение этого значения, в данном случае кортежа, требует вызова метода `element/2`. Наконец, код вызывам метод `drop:fall_velocity` с этим значением.

Если вы хотите, вы можете поместить все это в одну строку как `drop:fall_velocity(element(2,io:read("What {planemo, distance} ? >>")))`., Но эту строку кода трудно читать, а также непросто вносить измения.

Для вашего собственного использования это может быть совершенно нормально. Простой сеанс работы в оболочке может выглядеть следующим образом:

```
1> c(drop).
{ok,drop}
2> c(ask).
{ok,ask}
3> ask:term().
What {planemo, distance} ? >>{mars,20}.
12.181953866272849
```

Если вы пропустите точку в конце сроки, Erlang будет повторять запрос, но не покажет, где вы были, давая вам прочитать строку выше. Кроме того, данные, которые вы вводите в приглашении `io:read/1`, становятся частью истории команд консоли, и вы можете повторить их со стрелкой вверх. (Эти проблемы связаны с оболочкой Erlang, а не с самой функцией.)

Однако все может быстро стать некорректным, если пользователь вводит некорректные термы – число вместо кортежа или термы с неправильным синтаксисом.

```
4> ask:term().
       What {planemo, distance} ? >>20.
       ** exception error: no function clause matching
       drop:fall_velocity(20) (drop.erl, line 4)
5> ask:term().
       What {planemo, distance} ? >>.
       ** exception error: no function clause matching
       drop:fall_velocity({1,erl_parse,
       ["syntax error before: ","'.'"]}) (drop.erl, line 4)
```

В любом случае передача извлеченного терма непосредственно в `fall_velocity/1` – плохая идея. В первом случае мы получаем ошибку потому, что `fall_velocity/1` ожидает кортеж, а не число. Во втором случае `fall_velocity/1` имеем похожую проблему, так как ему отправляется сообщение об ошибке, а не терм, который он может обработать. Пример 5-2 в *`ch05/ex2-ask`* показывает лучший способ решения подобных проблем. Мы предоставляем пользователю сообщение об ошибке, когда программа получает неправильный тип информации или ошибочную информацию. (Используется сопоставление с образцом вместо функции `element/2`.)

Пример 5-2. Запрашивать у пользователя терм Erlang и обрабатывать некорректный ввод данных

```
-module(ask).
-export([term/0]).

term() ->
       Input = io:read("What {planemo, distance} ? >>"),
       process_term(Input).
```

```
process_term({ok, Term}) when is_tuple(Term) ->
       drop:fall_velocity(Term);

process_term({ok, _}) -> io:format("You must enter a tuple.~n");

process_term({error, _}) -> io:format("You must enter a tuple with correct syntax.~n").
```

К сожалению, это не решает все возможные проблемы. Пользователи могут по-прежнему вводить кортежи с неправильным содержимым, и `drop:fall_velocity` сообщит об ошибке. Глава 9 исследует, как решить эту проблему гораздо более подробно.

Однако, когда вы приступаете к созданию такого интерфейса, скорее всего, это происходит не потому, что ввод названия функции `ask:term()` короче, чем `drop:fall_velocity`. Весьма вероятно, что вы захотите попробовать ввести ряд значений, поэтому вы захотите, чтобы вопрос повторился. Пример 5-3 в *`ch05/ex3-ask`* представляет результат (правильно отформатированного) вызова пользователю, а затем снова вызывает `term()`, настраивая рекурсивный цикл. (Он также предлагает способ выйти из цикла.)

Пример 5-3. Запросить у пользователя терм Erlang и обработать некорректный результат

```
-module(ask).
-export([term/0]).

term() ->
       Input = io:read("What {planemo, distance} ? >>"),
       process_term(Input).

process_term({ok, Term}) when is_tuple(Term) ->
       Velocity = drop:fall_velocity(Term),
       io:format("Yields ~w. ~n",[Velocity]),
       term();

process_term({ok, quit}) ->
       io:format("Goodbye.~n");
       % does not call term() again

process_term({ok, _}) ->
       io:format("You must enter a tuple.~n"),
       term();

process_term({error, _}) ->
       io:format("You must enter a tuple with correct syntax.~n"),
       term().
```

Когда вы скомпилируете модуль `ask` и вызовите `ask:term/0`, вы увидите, что вопрос повторяется до тех пор, пока вы продолжаете вводить соответствующие кортежи. Чтобы выйти из этого цикла, просто введите атом `quit`, а затем точку.

```
6> c(ask).
{ok,ask}
7> ask:term().
What {planet, distance} ? >>{mars,20}.
Yields 12.181953866272849.
What {planet, distance} ? >>20.
You must enter a tuple.
What {planet, distance} ? >>quit.
Goodbye.
ok
```

Считывание символов

Функция `io:get_chars/2` позволит вам получить всего несколько символов от пользователя. Это кажется удобным, если, например, у вас есть список опций. Представьте параметры пользователю и дождитесь ответа. В этом случае список `planmos`

является опцией, и их легко нумеровать с 1 по 3, как показано в коде для примера 5-4, который вы можете найти в *ch05/ex4-ask*. Это означает, что вам просто нужен односимвольный ответ.

Пример 5-4. Представление меню и ожидание односимвольного ответа

```
-module(ask).
-export([chars/0]).

chars() ->
       io:format("Which planemo are you on?~n"),
       io:format(" 1. Earth ~n"),
       io:format(" 2. Earth's Moon~n"),
       io:format(" 3. Mars~n"),
       io:get_chars("Which? > ",1).
```

Большая часть этого представляет меню, и вы можете объединить все эти вызовы `io:format/1` в один вызов, если хотите. Ключевым элементом является вызов `io:get_chars/2` в конце. Первый аргумент – это приглашение, а второй – количество возвращаемых символов. Функция по-прежнему позволяет пользователям вводить все, что они хотят, пока они не нажмут Enter, но она покажет вам только первое, сколько бы символов вы не указали.

```
1> c(ask).
{ok,ask}
2> ask:chars().
Which planemo are you on?
1. Earth
2. Earth's Moon
3. Mars
Which? > 3
"3"
3>
3>
```

Функция io:get_chars возвращает строку «3», символ, введенный пользователем, после нажатия клавиши Enter. Тем не менее, может быть введено и большее количество символов. Если было введено большее количетсов символов, то они будут игнорироваться:

```
4> ask:chars().
Which planemo are you on?
1. Earth
2. Earth's Moon
3. Mars
Which? > 222222
"2"
5> 22222
5>
```

Могут быть случаи, когда `io:get_chars` – это именно то, что вам нужно.

Считывание текстовых строк

Erlang предлагает несколько различных функций, которые приостанавливают запрос информации у пользователей. Функция `io:get_line/1` ожидает, пока пользователь введет полную строку текста, оканчивающуюся новой строкой. Затем вы можете обработать строку, чтобы извлечь нужную информацию, и в буфере ничего не останется. Пример 5-5 в *ch05/ex5-ask* показывает, как это может работать, хотя извлечение информации несколько сложнее, чем хотелось бы.

Пример 5-5. Сбор пользовательских ответов из строки

```
-module(ask).
-export([line/0]).

line() ->
        Planemo = get_planemo(),
        Distance = get_distance(),
        drop:fall_velocity({Planemo, Distance}).

get_planemo() ->
        io:format("Where are you?~n"),
        io:format(" 1. Earth ~n"),
        io:format(" 2. Earth's Moon~n"),
        io:format(" 3. Mars~n"),
        Answer = io:get_line("Which? > "),
        Value = hd(Answer),
        char_to_planemo(Value).

char_to_planemo(Char) ->
        if
          [Char] == "1" -> earth;
          Char == $2 -> moon;
          Char == 51 -> mars
        end.

get_distance() ->
        Input = io:get_line("How far? (meters) > "),
        Value = string:strip(Input, right, $\n),
        {Distance, _} = string:to_integer(Value),
        Distance.
```

Функция `line/0` просто вызывает три другие функции. Она вызывает `get_planemo/0`, чтобы представить меню пользователю и получить ответ, и она также вызывает `get_distance/0`, чтобы спросить пользователя расстояние падения. Далее вызывается функция `drop:fall_velocity/1`, чтобы получить скорость, с которой объект без трения ударится о поверхность при падении с этой высоты.

Функция `get_planemo/0` представляет собой комбинацию вызовов `io:format/1` для представления информации. Вызов `io:get_line/1` используется для получения данных от пользователя. В отличие от `io:get_chars/1`, `io:get_line/1` возвращает все введенное пользователем значение, включая перевод строки и ничего не оставляет в буфере.

```
get_planemo() ->
        io:format("Where are you?~n"),
        io:format(" 1. Earth ~n"),
        io:format(" 2. Earth's Moon~n"),
        io:format(" 3. Mars~n"),
        Answer = io:get_line("Which? > "),
        Character = hd(Answer),
        char_to_planemo(Character).
```

Последние две строки – это фактическая обработка строки. Единственный фрагмент ответа, который имеет значение для этой функции – это первый символ строки. Простой способ получить это с помощью встроенной функции `hd/1`, которая извлекает первый элемент из строки или списка.

Поскольку строки на самом деле являются списками чисел, вы можете вместо этого вызывать `lists:nth(1, Answer)`. Первый аргумент, 1, это позиция, которую вы хотите получить, а второй аргумент, Ответ, это список, в данном случае строка, из которой вы хотите ее получить. Для этой функции первый символ в строке Erlang находится в позиции 1, а не 0, как во многих других языках. Это дает смысл названию функции nth, когда пришло время получить 4-е, 5-е, 6-е и т. д.

Функция drop:fall_velocity/1 не знает, что делать с значениями, указанным как 1, 2 или 3; он ожидает атомы earth, moon или mars. Функция get_planemo/0 завершается возвратом значения этого преобразования, выполненного функцией char_to_planemo/1:

```
char_to_planemo(Char) ->
if
        [Char] == "1" -> earth;
        Char == $2 -> moon;
        Char == 51 -> mars
end.
```

Оператор if показывает три разных способа тестирования вводимых данных. Если вы предпочитаете анализировать символ в виде текста, вы можете заключить его в квадратные скобки и сравнить его со строкой, например, "1". Вы также можете проверить соответствие с символами Erlang, в которых $2 является значением для символа два. Наконец, если вам удобны символьные значения, вы можете сравнить их с этими значениями, например, 51, что соответствует 3. Атом, возвращенный оператором case, будет возвращен функции get_planemo/0, которая, в свою очередь, вернет его к функции line/0 для использования в расчете.

Вы также можете переписать эту функцию, чтобы пропустить оператор case и просто использовать сопоставление с образцом:

```
char_to_planemo($1) -> earth;
char_to_planemo($2) -> moon;
char_to_planemo($3) -> mars.
```

Система обозначений Erlang также понимает Юникод. Если вы попробуете $☃, то Юникод Снеговик, Erlang поймет, что это символ 9731, hex 2603. Он также понимает символы Emoji, которые часто бывают трудными для простых реализаций Юникода.

Получить расстояние можно ещё проще:

```
get_distance() ->
        Input = io:get_line("How far? (meters) > "),
        {Distance, _} = string:to_integer(Value),
        Distance.
```

Переменная Input получает ответ пользователя на вопрос «Как далеко?», А функция string:to_integer/1 извлекает целое число из этого ответа. Совпадение с шаблоном слева захватывает первый фрагмент возвращаемого им кортежа, который является целым числом, в то время как подчеркивание отбрасывает остальную часть того, что он отправляет, что является чем-то еще в строке. Это будет включать новую строку, но также и любые десятичные части, которые вводят пользователи. Вы можете использовать string:to_float/1 для большей точности, но это не примет целое число. Использование string:to_integer/1 не идеально, но для этих целей это, вероятно, приемлемо.

Это не обязательное преобразование, но если вы просто хотите удалить лишние символы из ответов пользователя, то вы можете использовать string:strip(Input, right, $\n), где Input – это то, что только что пришло от пользователя.

Этот пример демонстрирует, как получить правильные ожидаемые результаты при правильном вводе данных.

```
1> c(ask).
{ok,ask}
2> ask:line().
Where are you?
1. Earth
2. Earth's Moon
3. Mars
Which? > 1
How far? (meters) > 20
19.79898987322333
3> ask:line().
Where are you?
1. Earth
2. Earth's Moon
3. Mars
Which? > 2
How far? > 20
8.0
```

В Главе 9 мы вернемся к этому коду и рассмотрим более эффективные способы обработки ошибок, которые пользователи могут спровоцировать, введя некорректные ответы.

Работа со строками, наверное, не самая сильная сторона Erlang, но у него есть все возможности, чтобы сделать практически все, что вам нужно. Читая следующие две главы о списках, помните, что строки на самом деле представляют собой списки символов, и вы можете использовать любой из инструментов списков для строк.

Вы можете узнать больше о работе со строками в главе 2 «Erlang Programming» (O'Reilly); Разделы 2.11 и 5.4 «Programming Erlang» (Pragmatic); Раздел 2.2.6 «Erlang and OTP in Action» (Manning); и 1-я глава «Learn You Some Erlang For Great Good!» (No Starch Press).

Глава 6. Списки

Erlang отлично справляется со списками, длинными сериями похожих (или нет) значений. Обработка списков позволяет легко увидеть ценность рекурсии и дает возможность выполнить большую работу за минимальные усилия.

Основы работы со списками

Список Erlang – это упорядоченный набор элементов. Обычно вы обрабатываете список по порядку, от первого элемента (заголовка) до последнего элемента, хотя бывают случаи, когда вы можете захотеть извлечь конкретный элемент из списка. Erlang также предоставляет встроенные функции для управления списками, когда вы не хотите обработать всю последовательность.

Синтаксис Erlang заключает списки в квадратные скобки и разделяет элементы запятыми. Список чисел может выглядеть следующим образом:

```
[1,2,4,8,16,32]
```

Элементы могут быть любого типа, включая числа, атомы, кортежи, строки и другие списки. Когда вы начинаете, определенно проще работать со списками, которые содержат только один тип элемента, а не смешанные, хотя сам Erlang не имеет такого ограничения. Также нет ограничений на количество элементов, которые может содержать список, хотя в конечном итоге вы можете найти практические ограничения памяти.

Вы можете сопоставлять шаблоны со списками так же, как и с другими структурами данных Erlang:

```
1> [1,X,4,Y] = [1,2,4,8].
[1,2,4,8]
2> X.
2
3> Y.
8
```

Хотя можно использовать списки вместо кортежей, ваш код будет иметь больше смысла, если вы будете использовать кортежи для обработки структур данных, содержащих различные типы данных в известной структуре, а списки для обработки структур данных, содержащих менее разнообразные данные в заранее неизвестном количестве. (Ожидается, что кортежи будут приходить в определенном порядке и могут также содержать списки, поэтому, если у вас есть структура данных, которая в основном известна, за исключением расширяющейся части или двух, включение списка внутри кортежа может быть приемлемым решением.)

Списки могут содержать списки, и иногда это может привести к неожиданным результатам. Если, например, вы хотите добавить список в список, у вас может получиться больше уровней списка, чем вы планировали:

```
4> Insert=[2,4,8].
[2,4,8]
5> Full = [1, Insert, 16, 32].
[1,[2,4,8],16,32]
```

Вы можете исправить это (если захотите) с помощью функции `lists:flatten/1`:

```
6> Neat = lists:flatten(Full).
[1,2,4,8,16,32]
```

Это также означает, что если вы хотите добавить списки, вам нужно решить, создаете ли вы список списков или один список, содержащий содержимое списков компонентов. Чтобы создать список списков, вы просто помещаете списки в списки.

```
7> A = [1,2,4].
[1,2,4]
8> B = [8,16,32].
[8,16,32]
9> ListOfLists = [A,B].
[[1,2,4],[8,16,32]]
```

Чтобы создать один список из нескольких списков, вы можете использовать функцию `lists:append/2` или эквивалентный оператор `++`.

```
10> Combined1 = lists:append(A,B).
[1,2,4,8,16,32]
11> Combined2 = A ++ B.
[1,2,4,8,16,32]
```

Оба дают один и тот же результат: комбинированный и сплющенный список.

Оператор `++` является ассоциативным справа. Он может изменять порядок получаемого списка при добавлении нескольких списков.

Если у вас есть набор списков, который вы хотите объединить, вы можете использовать функцию `lists:append/1`, которая принимает список списков в качестве аргумента и возвращает один список, содержащий их содержимое:

```
12> C = [64,128,256].
[64,128,256]
13> Combined4 = lists:append([A,B,C]).
[1,2,4,8,16,32,64,128,256]
```

Если вы хотите создать список последовательных целых чисел (или символов), то для этого лучше всего подходит функция `lists:seq/2`. Его аргументы – начало и конец списка значений. Например, `lists:seq(-2,8)` создает `[-2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8]`, а `lists:seq($A, $z)` создает строку (список) `"ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuvwxyz"`.

Разделение списков на головы и хвосты

Списки – это не только удобный способ хранения кусочков похожих данных. Их сильная сторона Erlang – это то, как они (списки) упрощают рекурсию. Списки естественным образом соответствуют логике «обратного отсчета», рассмотренной в Главе 4: вы можете перемещаться по списку, пока у вас не закончатся элементы. Во многих языках пробежка по списку означает выяснение, сколько элементов в нем содержится и последовательное их прохождение. Erlang использует другой подход, позволяя вам обрабатывать первый элемент списка, заголовок, извлекая остальную часть списка, хвост, чтобы вы могли рекурсивно передать его другому вызову.

Чтобы извлечь голову и хвост, вы используете сопоставление с шаблоном со специальной формой синтаксиса списка слева:

```
[Head | Tail] = [1,2,4].
```

Две переменные, разделенные вертикальной чертой (|), или минусы, для конструктора списка, будут связаны с заголовком и хвостом списка справа. В консоли Erlang просто сообщит содержимое правой части выражения, а не фрагменты, созданные при сопоставлении с образцом, но если вы работаете со списком, вы можете увидеть результаты:

```
1> List = [1,2,4].
[1,2,4]
2> [H1 | T1] = List.
[1,2,4]
3> H1.
1
4> T1.
[2,4]
5> [H2 | T2] = T1.
[2,4]
6> H2.
2
7> T2.
[4]
8> [H3 | T3] = T2.
[4]
9> H3.
4
10> T3.
[]
11> [H4 | T4] = T3.
** exception error: no match of right hand side value []
```

Строка 2 копирует исходный список, разделяя его на две меньшие части. H1 будет содержать первый элемент списка, тогда как T1 будет содержать список, в котором есть все, кроме первого элемента. Строка 5 повторяет процесс в меньшем списке, разбивая T1 на H2 и T2. На этот раз T2 все еще является списком, как показано в строке 7, но содержит только один элемент. Строка 8 снова разбивает этот список из одного элемента, помещая значение в H3, а пустой список в T3.

Что происходит, когда вы пытаетесь разделить пустой список, как показано в строке 11? Erlang сообщает об ошибке «нет совпадения…». К счастью, это не означает, что рекурсия в списках создаёт ошибку. Отсутствие совпадения естественным образом остановит рекурсивный процесс, что, вероятно, то, что вы хотите.

Разделение данных на голову и хвост работает только для списков. Если порядок имеет значение, и вам действительно нужно просмотреть список в обратном порядке, вам нужно будет использовать функцию `lists:reverse`, а затем пройтись по этому новому списку.

Обработка содержания списка

Абстракции голова и хвост были созданы специально для рекурсивной обработки списка. Обычно, список поступает в качестве аргумента, а затем передается другой (обычно закрытой) функции с аргументом-накопителем. В простом случае можно выполнить анализ содержимого списка. Пример 6-1, в *`ch06/ex1-product`*, показывает этот используемый шаблон, для перемножения элементы списка.

Пример 6-1. Расчет произведения значений в списке

```
-module(overall).
-export([product/1]).

product([]) -> 0; % in case the list is empty, return zero
product(List) -> product(List,1).

product([], Product) -> Product; % when list empty, stop, report
product([Head|Tail], Product) -> product(Tail, Product * Head).
```

В этом модуле функция `product/1` является шлюзом, который передает список (если список содержит элементы) плюс накопитель в `product/2`, который выполняет реальную работу. Если вы хотите проверить список получателей, чтобы убедиться, что он соответствует вашим ожиданиям, возможно, имеет смысл выполнить эту работу в `product/1`, а `product/2` пусть сосредоточится на рекурсивной обработке.

Функция `product/2` имеет два входных параметра. Первый соответствует пустому списку и вызывается в конце рекурсивного процесса, когда больше нет записей для обработки или если список прибывает пустым. Возвращает второй аргумент - аккумулятор.

Второе условие делает больше работы, если поступающий список не пуст. Сначала сопоставление с образцом (`[Head | Tail]`) отделяет первое значение списка от остальной части списка. Затем он снова вызывает `product/2` с оставшейся (если есть) частью списка и новым аккумулятором, который умножается на значение первой записи в списке. Результатом будет произведение значений, включенных в список:

```
1> c(overall).
{ok,overall}
2> overall:product([1,2,3,5]).
30
```

Всё прошло гладко, но что случилось? После того, как `product/1` вызвал `product/2`, он сделал пять итераций по списку, заканчивая пустым списком, как показано в Таблице 6-1.

Таблица 6-1. Рекурсивная обработка простого списка с помощью функции product/2

Входной параметр (список)	Значение входного параметра (Product)	Голова	Хвост
[1,2,3,5]	1	1	[2,3,5]
[2,3,5]	1(1*1)	2	[3,5]
[3,5]	2(1*2)	3	[5]
[5]	6(2*3)	5	[]
[]	30(6*5)	Ничего	Ничего

Последнее значение полученное накопителем Product равен 30, будет передан в функцию с одним параметром и передан как возвращаемое значение. Когда `product/1` получит это значение, он также сообщит 30 как свое возвращаемое значение и работа функции будет завершена

.

Поскольку строки Erlang представляют собой списки символов, представленных в виде чисел, вы можете делать манипуляции, такие как overall:product("funny"). Функция product/1 будет интерпретировать значения символов, как числа и возвратит число 17472569400.

Создание списков с помощью абстракций «голова» и «хвост»

Хотя бывают случаи, когда вы хотите вычислить одно значение из списка, большая часть обработки списка включает в себя изменение списков или преобразование списка в другой список. Поскольку вы не можете изменить список, изменение или преобразование списка означает создание нового списка. Чтобы сделать это, вы используете тот же синтаксис «голова / хвост» вертикальной черты, но с правой стороны шаблона, а не слева. Вы можете попробовать это в консоли, хотя это более полезно в модуле:

```
1> X=[1|[2,3]].
[1,2,3]
```

Erlang интерпретирует это выражение [1 | [2,3]], как создание списка. Если значение справа от вертикальной панели является списком, оно добавляется к заголовку в виде списка. В этом случае результатом является аккуратный список чисел. Есть несколько других форм, о которых вы должны знать:

```
2> Y=[1,2 | [3]].
[1,2,3]
3> Z=[1,2 | 3].
[1,2|3]
```

В строке 2 нет списка, обернутого вокруг двух элементов в голове, но конструктор по-прежнему плавно смешивает голову и хвост. (Если вы заключите их в квадратные скобки, конструктор списка предполагает, что вы хотите, чтобы список был первым элементом в списке, поэтому [[1,2] | [3]] создаст [[1,2], 3] .)

Тем не менее, строка 3 демонстрирует, что происходит, если вы не заключаете хвост в квадратные скобки - вы получаете список, называемый неправильным списком, который все еще содержит конструктор со странным хвостом. До тех пор, пока вы не освоитесь в Erlang, вам, вероятно, следует избегать этого, так как это приведет к ошибкам во время выполнения, если вы попытаетесь обработать его как обычный список. В конце концов вы можете найти причины для этого или встретить код, который его использует.

В более общем случае вы будете использовать конструкторы списков для создания списков внутри рекурсивных функций. Пример 6-2, который вы можете найти в *ch06/ex2-drop*, начинается с набора кортежей, представляющих плоскость и расстояния. С помощью модуля `drop` из Примера 3-8 он создает список скоростей для соответствующих падений.

Пример 6-2. Вычисление ряда скоростей падения с ошибкой

```
-module(listdrop).
-export([falls/1]).

falls(List) -> falls(List,[]).

falls([], Results) -> Results;
falls([Head|Tail], Results) -> falls(Tail, [drop:fall_velocity(Head) | Results]).
```

Многое из этого известно из Примера 6-1, за исключением того, что переменная Results получает список вместо числа, а последняя строка fall/2 создает список вместо одного значения. Однако, если вы запустите его, вы увидите одну небольшую проблему:

```
1> c(drop).
{ok,drop}
2> c(listdrop).
{ok,listdrop}
3> listdrop:falls([{earth,20},{moon,20},{mars,20}]).
[12.181953866272849,8.0,19.79898987322333]
```

Результирующие скорости меняются местами: у Земли больше гравитации, чем у Марса, и объекты должны падать быстрее на Земле. Что случилось? Последняя ключевая строка в `fall/2` читает список от начала до конца и создает список от конца до начала. Это ставит значения в неправильном порядке. К счастью, как показывает пример 6-3, который вы можете найти в *ch06/ex3-drop*, это легко исправить. Вам необходимо вызвать `lits:reverse/1` в предложении функции `fall/2`, которая обрабатывает пустой список.

Пример 6-3. Вычисление ряда скоростей падения с исправленной ошибкой последовательности элементов в списке

```
-module(listdrop).
-export([falls/1]).

falls(List) -> falls(List,[]).

falls([], Results) -> lists:reverse(Results);
falls([Head|Tail], Results) -> falls(Tail, [drop:fall_velocity(Head) | Results]).
```

Теперь элементы списка имеют правильный порядок:

```
4> c(listdrop).
{ok,listdrop}
5> listdrop:falls([{earth,20},{moon,20},{mars,20}]).
[19.79898987322333,8.0,12.181953866272849]
```

Вы, конечно могли бы поместить вызов функции `lists:reverse/1` в результирующую открытую функцию `fall/1`. В любом случае вам выбирать предпочтительное решение. Я предпочитаю проводить необходимые преобразования в функции `falls/2`. В этом случае результирующей функции необходимо только получить и передать результат вычислений.

Совместное использование списков и кортежей

По мере углубления в Erlang и обхода более сложных структур данных вы можете обнаружить, что вы обрабатываете списки кортежей, или что было бы удобнее упорядочиваете два списка в один список кортежей или наоборот. Модуль списков включает в себя простые решения для таких преобразований и поиска.

Простейшим набором инструментов являются функции `lists:zip/2` и `lists:unzip/1`. Они могут превратить два списка одинакового размера в список кортежей или список кортежей в два списка.

```
1> List1=[1,2,4,8,16].
[1,2,4,8,16]
2> List2=[a,b,c,d,e].
[a,b,c,d,e]
3> TupleList=lists:zip(List1,List2).
[{1,a},{2,b},{4,c},{8,d},{16,e}]
4> SeparateLists=lists:unzip(TupleList).
{[1,2,4,8,16],[a,b,c,d,e]}
```

Два списка, List1 и List2, имеют разное содержимое, но одинаковое количество элементов. Функция `lists:zip/2` возвращает список, содержащий кортеж для каждого элемента в исходном списке. Функция `lists:unzip/1` берет этот список двухкомпонентных кортежей и разбивает его на кортеж, содержащий два списка.

Erlang также предоставляет `lists:zip3/3` и `lists:unzip3/1`, которые выполняют одинаковое комбинирование и разделение наборов из трех списков или значений кортежей.

Вы также, вероятно, столкнетесь со случаями, когда вам понадобится обработать другой тип списка, содержащего кортежи, набор значений, идентифицируемых ключами. Многие языки включают в себя ассоциативные массивы, где доступ обеспечивается через значения ключей, но списки Erlang всегда последовательны и не имеют встроенной концепции получения информации с помощью ключа. Однако модуль `lists` предоставляет функции, которые поддерживают обработку списка кортежей, как если бы он был хранилищем ключей/значений, таких как хеш-таблица, хеш-дерево или ассоциативный массив.

Хотя большая часть этой работы переходит на использование карт, описанных в Главе 10, вы можете найти старый код, который требует от вас работы с ними. Ключ должен находиться в согласованном месте в кортежах, хранящихся в списке. Поскольку функции позволяют вам указать местоположение, вы можете разместить их в любом месте кортежа, если вы будете последовательны, но в этом примере они будут на первой позиции.

Первая функция для изучения – `lists:keystore/4`. Она принимает значение ключа, позицию, список, который является предыдущим состоянием хранилища ключ/значение (определено в строке 1 следующего примера кода), и кортеж. Если ни один кортеж не имеет этого значения ключа, то новый кортеж просто добавляется в список, как показано в строке 2 следующего примера кода. Если, как и в строке 3, кортеж уже имеет это значение ключа, функция вернет список, который заменяет соответствующий кортеж новым.

```
1> Initial=[{1,tiger}, {3,bear}, {5,lion}].
[{1,tiger},{3,bear},{5,lion}]
2> Second=lists:keystore(7,1,Initial,{7,panther}).
[{1,tiger},{3,bear},{5,lion},{7,panther}]
3> Third=lists:keystore(7,1,Second,{7,leopard}).
[{1,tiger},{3,bear},{5,lion},{7,leopard}]
```

Вы также можете передать lists:keystore/4 пустой список для массива, и он просто вернет список, содержащий новый кортеж.

Иногда вы можете захотеть заменить значение, *только* если оно присутствует, а не добавлять новое значение в список. `lists:keyreplace/4` делает именно это.

```
4> Fourth=lists:keyreplace(6,1,Third,{6,chipmunk}).
[{1,tiger},{3,bear},{5,lion},{7,leopard}]
```

В предыдущем списке не было элемента со значением ключа 6, поэтому `lists:keyreplace/4` только что вернул копию исходного списка.

Все эти функции копируют списки или создают новые измененные версии списка. Как и следовало ожидать в Erlang, первоначальный список не тронут.

Если вы хотите получить информацию о списке, функция `lists:keyfind/3` данные, соответствующие данному ключу:

```
5> Animal5=lists:keyfind(5,1,Third).
{5,lion}
```

Однако, если ключ отсутствует, вы просто получите значение false вместо кортежа.

```
6> Animal6=lists:keyfind(6,1,Third).
false
```

Построение списка списков

Хотя простая рекурсия не слишком сложна, обработка списков может превращаться в списки списков на разных этапах. Треугольник Паскаля – классический математический инструмент. Он относительно прост в создании, но демонстрирует более сложную работу со списками. Он начинается с `1` в верхней части, а затем каждая новая строка состоит из суммы двух чисел над ней:

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
  ...
```

Если эти числа кажутся знакомыми, возможно, это потому, что они представляют собой биномиальные коэффициенты, которые появляются, когда вы возводите (x + y) в степень. Это только начало этого математического чуда, более подробно описанного по адресу https://ru.wikipedia.org/wiki/Треугольник_Паскаля.

Треугольник Паскаля легко реализовать с Erlang несколькими способами. Вы можете применить методы списков, которые уже обсуждались в этой главе, рассматривая каждую строку как список, а треугольник как список списков. Код будет заполнен первой строкой – верхней 1, представленной как [0,1,0]. Дополнительные нули значительно упрощают сложение.

Представленное решение не является самым эффективным, элегантным или максимально компактной реализацией. Эта реализация помогает лучше объяснить работу со списками. Как только это станет понятным, и вы узнаете о списках в Главе 7, вы можете исследовать, как может выглядеть гораздо более компактная версия. см. http://rosettacode.org/wiki/Pascal's_triangle#Erlang.

Сначала в примере 6-4 вычисляем строки раздельно. Это простой рекурсивный процесс, обход старого списка и добавление его содержимого для создания нового списка.

Пример 6-4. Расчет строки

```
-module(pascal).
-export([add_row/1]).

add_row(Initial) -> add_row(Initial, 0, []).

add_row([], 0, Final) -> [0 | Final];

add_row([H | T], Last, New) -> add_row(T, H, [Last + H | New]).
```

Функция `add_row/1` устанавливает все, отправляя текущую строку, 0, чтобы начать расчёты, и пустой список, который вы можете рассматривать как «куда идут результаты», хотя на самом деле это аккумулятор. Функция `add_row/3` имеет два условия. Первое проверяет, является ли добавляемый список пустым. Если это так, то он возвращает последнюю строку, добавляя 0 в начале.

Большая часть работы выполняется во втором разделе `add_row/3`. Когда он получает свои аргументы, `[H | T]` сопоставление с образцом разбивает начало списка на значение `H` (число), а хвост – на `T` (список, который может быть пустым, если это был последний номер). Он также получает значения для последнего обработанного числа и текущего создаваемого нового списка.

Затем он делает рекурсивный вызов `add_row/3`. В этом новом вызове хвост старого списка, `T`, является новым списком для обработки, значение `H` становится последним обработанным числом, и третий аргумент, список, открывается с фактическим выполнением добавления, которое затем объединяется с остальной частью нового списка строится.

Поскольку списки в треугольнике симметричны, нет необходимости использовать `lists:reverse/1`, чтобы перевернуть их. Вы можете, конечно, если хотите.

Вы можете легко проверить это из консоли, но помните, что ваши тестовые списки должны быть обернуты в нули:

```
1> c(pascal).
{ok,pascal}
2> pascal:add_row([0,1,0]).
[0,1,1,0]
3> pascal:add_row([0,1,1,0]).
[0,1,2,1,0]
4> pascal:add_row([0,1,2,1,0]).
[0,1,3,3,1,0]
```

Теперь, когда вы можете создать новую строку из старой, вам нужно иметь возможность создать набор строк в верхней части треугольника, как показано в примере 6-5, который вы можете найти в *`ch06/ex4-pascal`*. Функция `add_row/3` эффективно ведет обратный отсчет до конца списка, но `triangle/3` нужно будет считать до заданного числа строк. Функция `triangle/1` устанавливает все, определяя начальную строку, устанавливая счетчик в 1 (потому что эта начальная строка является первой строкой), и передавая количество строк, которые будут созданы.

Функция `triangle/3` имеет два условия. Во-первых, условие остановки рекурсии, когда создано достаточное количество строк, и переворачивает список. (Отдельные строки могут быть симметричными, но сам треугольник - нет.) Второе условие выполняет фактическую работу по генерации новых строк. Оно получает предыдущую строку, сгенерированную из списка, а затем передает ее в функцию `add_row/1`, которая возвращает новую строку. Затем оно вызывает себя с новым списком, увеличенным значением `Count` и значением Rows, необходимым для остановки.

Пример 6-5. Вычисление всего треугольника с обеими функциями

```
-module(pascal).
-export([triangle/1]).

triangle(Rows) -> triangle([[0,1,0]],1,Rows).

triangle(List, Count, Rows) when Count >= Rows -> lists:reverse(List);
triangle(List, Count, Rows) ->
       [Previous | _] = List,
       triangle([add_row(Previous) | List], Count+1, Rows).

add_row(Initial) -> add_row(Initial, 0, []).

add_row([], 0, Final) -> [0 | Final];
add_row([H | T], Last, New) -> add_row(T, H, [Last + H | New]).
```

К счастью, всё это работает.

```
5> c(pascal).
{ok,pascal}
6> pascal:triangle(4).
[[0,1,0],[0,1,1,0],[0,1,2,1,0],[0,1,3,3,1,0]]
7> pascal:triangle(6).
[[0,1,0],
[0,1,1,0],
[0,1,2,1,0],
[0,1,3,3,1,0],
[0,1,4,6,4,1,0],
[0,1,5,10,10,5,1,0]]
```

Треугольник Паскаля может быть немного более аккуратным набором списков, чем большинство, которые вы будете обрабатывать, но этот вид многоуровневой обработки списков является очень распространенной тактикой обработки и генерации списков данных.

Вы можете узнать больше о работе со списками в Главе 2 «Erlang Programming» (O'Reilly); Разделы 2.10 и 3.5 «Programming Erlang» (Pragmatic); Раздел 2.2.5 «Erlang and OTP in Action» (Manning); и 1-я глава «Learn You Some Erlang For Great Good!» (No Starch Press).

Глава 7. Функции высшего порядка и генераторы списков

Функции высшего порядка – это функции, которые принимают другие функции в качестве аргументов. Они являются ключевой частью, в которой мощь Erlang действительно проявляется наилучшим образом. В отличие от многих других языков, Erlang рассматривает функции высшего порядка как неотъемлемую и естественную часть языка, а не как что-то дополнительное.

Простые функции высшего порядка

Еще в главе 2 вы увидели, как использовать анонимные функции для создания функции:

```
1> Fall_velocity = fun(Distance) -> math:sqrt(2 * 9.8 * Distance) end.
#Fun<erl_eval.6.111823515>
2> Fall_velocity(20).
19.79898987322333
3> Fall_velocity(200).
62.609903369994115
```

Erlang позволяет не только помещать функции в переменные, но и передавать функции в качестве аргументов. Это означает, что вы можете создавать функции, поведение которых вы изменяете во время вызова, гораздо более сложными способами, чем это обычно возможно с параметрами. Очень простая функция, которая принимает в качестве аргумента другую функцию, может выглядеть как Пример 7-1, который вы можете найти в *ch07/ex1-hof*.

Пример 7-1. Чрезвычайно простая функция высшего порядка

```
-module(hof).
-export([tripler/2]).

tripler(Value, Function) -> 3 * Function(Value).
```

Имена аргументов ничего не значат (просто взяты для примера). `tripler/2` примет значение и функцию в качестве аргументов. Он передаёт значение функции и умножает полученный результат на три. В оболочке это может выглядеть следующим образом:

```
1> c(hof).
{ok,hof}
2> MyFunction=fun(Value)->20*Value end.
#Fun<erl_eval.6.111823515>
3> hof:tripler(6,MyFunction).
360
```

Мы определяем простую анонимную функцию, которая принимает один аргумент (и возвращает это число, умноженное на 20) и сохраняет его в переменной `MyFunction`. Затем она вызывает функцию `hof:tripler/2` со значением шесть и функцию `MyFunction`. В функции `hof:tripler/2` число передается в функцию, которая возвращает значение 120. Затем это число утраивается, возвращая значение 360.

Вы можете пропустить присвоение функции переменной, если хотите, просто введя описание анонимной функции на место аргумента функции `hof:tripler/2`:

```
4> hof:tripler(6,fun(Value)->20*Value end).
360
```

Возможно, но не всегда, такая форма записи может быть более читабельной. Этот случай тривиально прост, но демонстрирует общую схему использование функции в качестве аргумента.

Хотя это и мощная техника, вы можете легко перехитрить ее. (Да!) Как и в обычном коде, вам нужно убедиться, что число, а иногда и тип ваших аргументов совпадают. Дополнительная гибкость и мощь могут создать новые проблемы, если вы не будете осторожны.

У анонимной функции есть несколько других хитростей в рукаве, которые вы должны знать. Вы можете использовать её, чтобы сохранить контекст, даже тот контекст, который уже не существует.

```
5> X=20.
20
6> MyFunction2=fun(Value)->X * Value end.
#Fun<erl_eval.6.82930912>
7> f(X).
ok
8> X.
* 1: variable 'X' is unbound
9> hof:tripler(6,MyFunction2).
360
```

Строка 5 присваивает переменной с именем `X` значение, а строка `6` использует эту переменную в анонимной функции. Хотя строка 7 стирает переменную X, как показывает строка 8, строка 9 показывает, что анонимная функция `MyFunction2` по-прежнему помнит, что `X` было `20`. Даже если значение `X` было сброшено в оболочке, функция сохраняет значение и может воздействовать без неё. (Это называется *замыканием*.)

Вы также можете передать функцию из модуля, даже встроенную, в вашу (или любую) функцию более высокого порядка. Это тоже просто:

```
7> hof:tripler(math:pi(), fun math:cos/1).
-3.0
```

В этом случае функция `hof:tripler` получает значение `pi` и `fun`, который является функцией `math:cos/1` встроенного математического модуля. Поскольку косинус Пи равен `-1`, `hof:tripler` возвращает `-3.0`.

Создание новых списков с помощью функций высшего порядка

Списки являются одним из лучших и самых простых мест для применения функций высшего порядка. Применение функции ко всем компонентам списка для создания нового списка, сортировки списка или разбиения списка на более мелкие части является обычным делом. И это всё очень просто в реализации: встроенный модуль `lists` предлагает множество функций высшего порядка, которые перечислены в Приложении А. Они принимают функцию и список и что-то с ними делают. Вы также можете использовать *генераторы списков*, чтобы выполнять большую часть той же работы. Модуль `lists` на первый взгляд может показаться более простым, но, как вы увидите, генераторы списков являются мощным и лаконичным механизмом.

Получение данных на основе списка

Простейшая из этих функций – `foreach/2`, которая всегда возвращает атомы в порядке возрастания. Это может показаться странным, но `foreach/2` – это функция, которую вы будете вызывать тогда и только тогда, когда вы хотите что-то сделать со списком (реализовав побочные эффекты) – например, представить содержимое списка на консоли. Для этого определите простую функцию, которая применяет `io:format/2`, которая здесь хранится в переменной `Print`, и `List`, а затем передайте их обе в `lists:foreach/2`.

```
1> Print = fun(Value) -> io:format(" ~p~n",[Value]) end.
#Fun<erl_eval.6.111823515>
2> List = [1,2,4,8,16,32].
[1,2,4,8,16,32]
3> lists:foreach(Print,List).
 1
 2
 4
 8
 16
 32
ok
```

Функция `lists:foreach/2` прошла по списку по порядку и вызвала функцию `Print` с каждым элементом списка в качестве значения. Функция `io:format/2` внутри `Print` представила элемент списка с небольшим отступом. Когда был достигнут конец списка, `lists:foreach/2` вернул значение `ok`, которое также было передано на консоль.

Большинство демонстраций в этой главе будут работать с той же самой переменной `List`, содержащей список `[1,2,4,8,16,32]`.

Изменение значений списка с помощью функций

Вы также можете создать новый список на основе того, что функция делает с каждым элементом в исходном списке. Вы можете возвести в квадрат все значения в списке, создав функцию, которая возвращает квадрат ее аргумента, и передав ее в `lists:map/2`. Вместо того, чтобы возвращать ok, он возвращает новый список, отражающий работу функции, которую ему передали:

```
4> Square = fun(Value)->Value*Value end.
#Fun<erl_eval.6.111823515>
5> lists:map(Square,List).
[1,4,16,64,256,1024]
```

Есть еще один способ сделать то же самое, используя то, что Erlang называет *генератором списков*.

Это приводит к одному и тому же результирующему списку с другим (и более гибким) синтаксисом. Пока вы видели [A | B] синтаксис в конструкторах списков, для понимания списка используется [A || B] синтаксис. Эта дополнительная вертикальная черта меняет всю интерпретацию. Вместо того чтобы быть головой и хвостом, это выражение - здесь функция - это правило для извлечения аргументов функции из списка, называемое генератором.

```
6> [Square(Value) || Value <- List].
[1,4,16,64,256,1024]
```

В этом случае функция – это анонимная функция, которую вы помещаете в переменную Square в строке 4. Ее аргумент Value принимается для обхода списка. Эта стрелка – <- означает «элемент», или, если вы хотите быть более активным, «исходит из». Вы можете прочитать это как «Создать список, состоящий из квадратов значения. Value берутся из List.»

Строго говоря, выражение слева не обязательно должно быть объявлено, как функция. Вы можете получить те же результаты с чем-то менее формальным:

```
7> [Value * Value || Value <- List].
[1,4,16,64,256,1024]
```

Оператор умножения (*) технически является вызовом функции */2. Любое допустимое выражение Erlang может быть слева от ||.

Фильтрация списка значений

Модуль `lists` предлагает несколько различных функций для фильтрации содержимого списка на основе функции, которую вы предоставляете в качестве параметра. Наиболее очевидный список `lists:filter/2` возвращает список, состоящий из членов исходного списка, для которого функция вернула значение true. Например, если вы хотите отфильтровать список целых чисел до значений, которые могут быть представлены в виде четырех двоичных цифр, то есть чисел от 0 или больше, но меньше 16, вы можете определить функцию и сохранить ее в переменной `Four_bits`:

```
8> Four_bits = fun(Value)-> (Value>=0) and (Value<16) end.
#Fun<erl_eval.6.111823515>
```

Затем, если вы примените его к ранее определенному списку `[1,2,4,8,16,32]`, вы получите только первые четыре значения:

```
9> lists:filter(Four_bits,List).
[1,2,4,8]
```

Такой же результат вы можите достигнуть и с помощью генератора списков. На этот раз вам на самом деле не нужно создавать функцию, а вместо этого используйте охранное выражение (написанного без оператора when) с правой стороны от генератора:

```
10> [Value || Value <- List, Value>=0, Value<16].
[1,2,4,8]
```

Если вам также нужен список значений, которые не совпадают с условиями охранного выражения, то вам подойдет функция `lists:partition/2`, которую мы рассмотрим в разделе «Разделение списков», вернет кортеж, который содержит два списка – совпадающие элементы в своем первом элементе и несовпадающие элементы во втором.

Больше чем генераторы списков

Генераторы списков являются краткими и мощными, но им не хватает нескольких ключевых функций, доступных в другой рекурсивной обработке. Единственный тип результата, который они могут вернуть – это список. В работе со списками могут быть ситуации, когда вы захотите обработать список и вернуть что-то еще, например, логическое значение, кортеж или число. Понимания списков также не поддерживают аккумуляторы и не позволяют полностью приостановить обработку при соблюдении определенных условий.

Вы можете написать свои собственные рекурсивные функции для обработки списков, но большую часть времени вы обнаружите, что модуль `lists` уже предлагает функцию, которая принимает определенную вами функцию и список и возвращает то, что вам нужно.

Проверка списков

Иногда вы просто хотите узнать, соответствуют ли все значения (или любое из значений) в списке определенным критериям. Они все определенного типа, или они имеют значение, которое соответствует определенным критериям?

Функции `lists:all/2` и `lists:any/2` позволяют проверять список на соответствие правилам, указанным в функции. Если ваша функция возвращает true для всех значений списка, обе эти функции вернут true. lists:any/2 возвращает true, если одно или несколько значений в списке приведут к тому, что ваша функция вернет true. Обе вернут false, если ваша функция всегда возвращает false.

`lists:all/2` и `lists:any/2` не обязательно анализирует весь список; как только они достигают значения, дающего окончательный ответ, они остановятся и вернут этот ответ.

```
11> IsInt = fun(Value) -> is_integer(Value) end.
#Fun<erl_eval.6.111823515>
12> lists:all(IsInt, List).
true
13> lists:any(IsInt, List).
true
14> Compare = fun(Value) -> Value > 10 end.
#Fun<erl_eval.6.111823515>
15> lists:any(Compare, List).
true
16> lists:all(Compare, List).
false
```

Вы можете думать о lists:all/2 как функции and применяемой к спискам или более точно, как andalso, потому что она останавливает обработку, как только она встречает ложный результат. Аналогично, lists:any/2 похожа на or или orrelse, если функция останавливается, как только находит истинный результат. Пока вам нужно только проверить отдельные значения в списках. Эти две функции высокого порядка могут сэкономить вам написание большого количества рекурсивного кода.

Разделение списков

Фильтрация списков полезна, но иногда вы хотите знать, что не прошло через фильтр, а иногда вы просто хотите разделить элементы.

Функция lists:partition/2 возвращает кортеж, содержащий два списка. Первый – это элементы списка, которые удовлетворяли условиям, указанным в предоставленной вами функции, а второй – элементы, которые не выполнялись. Если переменная Compare определена, как показано в строке 14 предыдущей демонстрации, и возвращает значение true, если значение списка больше 10, тогда вы можете легко разбить список на два:

```
17> lists:partition(Compare,List).
{[16,32],[1,2,4,8]}
```

Иногда вам нужно разделить список, начиная с начала – с заголовка и останавиться, когда значение списка больше не соответствует условию. `lists:takewhile/2` и `lists:dropwise/2` функции создают новый список, который содержит части старого списка до или после возникновения граничного условия. Эти функции не являются фильтрами, и чтобы прояснить как они работают, примеры используют другой список, в отличии от остальных в этой главе.

```
18> Test=fun(Value) -> Value < 4 end.
#Fun<erl_eval.6.111823515>
19> lists:dropwhile(Test, [1,2,4,8,4,2,1]).
[4,8,4,2,1]
20> lists:takewhile(Test, [1,2,4,8,4,2,1]).
[1,2]
```

Обе функции пробегают список от начала до конца и останавливаются, когда достигают значения, для которого функция, указанная вами в качестве первого аргумента, возвращает false. Функция lists:dropwhile/2 возвращает то, что осталось от списка, включая значение, которое провалило тест. Однако она не отфильтровывает более поздние записи списка, которые он могла бы отбросить, если бы они появились в нём ранее. Функция lists:takewhile/2 возвращает то, что уже было обработано, не включая значение, которое провалило тест.

Свёртка списков

Добавление аккумулятора для обработки списков позволяет превратить списки в гораздо большее, чем другие списки, и открывает путь к гораздо более сложной обработке. Функции lists:foldl/3 и lists:foldr/3 позволяют вам указать функцию, начальное значение для аккумулятора и список. Вместо функций с одним аргументом, которые вы уже видели, вам нужно создать функцию с двумя аргументами, которая принимает текущее значение в обходе списка и аккумулятор. Результатом этой функции станет новое значение аккумулятора.

Определение функции, которая работает в функциях сворачивания, выглядит немного иначе из-за её двух аргументов:

```
21> Divide = fun(Value, Accumulator) -> Value / Accumulator end.
#Fun<erl_eval.6.111823515>
```

В теле функции первый аргумент делит (являющийся элементом списка) на аккумулятор (который передаётся функции для осуществления свёртывания).

У свёртки есть еще одна особенность. Вы можете выбрать, хотите ли вы, чтобы функция проходила по списку от с начала до конца (для этого используется функция `lists:foldl/3`) или от конца к началу (с помощью функции `lists:foldr/3`). Если порядок не меняет результат, то предпочтительнее использовать `lists:foldl/3`, поскольку его реализация является хвост-рекурсивной и более эффективной в большинстве ситуаций.

Функция Divide – это один из тех случаев, в котором результаты будут очень разными в зависимости от направления обработки списка (и начального значения аккумулятора). В этом случае сворачивание также дает результаты, отличные от ожидаемых при простом делении. Например, рассмотрим список целых чисел [1,2,4,8,16,32]. При переборе элементов списка с начала до конца это движение даст 1/2/4/8/16/32 выражение, а движение справа налево – 32/16/8/4/2/1, в том случае, если вы используете 1 в качестве начального значения аккумулятора. Однако эти функции не дают таких результатов.

```
22> 1/2/4/8/16/32.
3.0517578125e-5
23> lists:foldl(Divide,1,List).
8.0
24> 32/16/8/4/2/1.
0.03125
25> lists:foldr(Divide,1,List).
0.125
```

Этот код кажется слишком простым, чтобы иметь ошибку, так что же происходит? В таблице 7-1 показаны вычисления для lists:foldl(Divide, 1, List), а в таблице 7-2 показаны пошаговые lists:foldr(Divide, 1, List).

Таблица 7-1. Рекурсивная обработка списка с помощью foldl/3

Входные значения	Накопитель (аккумулятор)	Результат деления
1	1	1
2	1 (1/1)	2
4	2 (2/1)	2
8	2 (4/2)	4
16	4 (8/2)	4
32	4	8

Таблица 7-2. Рекурсивное деление списка в обратном направлении с помощью foldr/3

Входные значения	Накопитель (аккумулятор)	Результат деления
32	1	32
16	32	0.5
8	0.5	16
4	16	0.25
2	0.25	8
1	8	0.125

Перемещение по списку шаг за шагом приводит к совершенно другим значениям. В этом случае поведение простой функции «Разделить» резко меняется выше и ниже значения 1, а сочетание с обходом элемента списка по элементу приводит к результатам, которые могут быть не совсем такими, как вы ожидали.

Результат foldl такой же, как 32/(16/(8/(4/(2/(1/1))))), а результат foldr такой же, как 1/(2/(4/(8/(16/(32/1))))). Круглые скобки в них выполняют ту же реструктуризацию, что и fold, а заключительные 1 в каждом соответствуют исходному значению аккумулятора.

Свёртывание – невероятно мощная операция. В этом простом, хотя и немного странном примере в качестве аккумулятора использовалось одно значение – число. Если вы используете кортеж в качестве аккумулятора, вы можете хранить все виды информации о списке, когда элементы списка анализируются, и даже выполнять несколько операций. Вы, вероятно, не захотите создавать анонимные функции отдельно, которые вы используете для этого, как однострочные, но возможности для этого есть.

Вы можете узнать больше о работе с функциями высокого порядка в главе 9 «Erlang Programming» (O'Reilly); Раздел 3.4 «Programming Erlang» (Pragmatic); Раздел 2.7 «Erlang and OTP in Action» (Manning); и 6-я глава «Learn You Some Erlang For Great Good!» (No Starch Press). Объяснение работы со списками в главе 9 «Erlang Programming» (O'Reilly); Раздел 3.6 «Programming Erlang» (Pragmatic); Раздел 2.9 «Erlang and OTP in Action» (Manning); и 1-я глава «Learn You Some Erlang For Great Good!» (No Starch Press).

Глава 8. Работа с процессами

Хотя Erlang является функциональным языком, программы Erlang редко строятся вокруг простых функций. Вместо этого, ключевой организационной концепцией Erlang является процесс – независимый компонент (построенный из функций), который отправляет и получает сообщения. Программы развертываются как наборы процессов, которые взаимодействуют друг с другом. Такой подход значительно облегчает распределение работы между несколькими процессорами или компьютерами, а также позволяет выполнять такие действия, как обновление программ на месте, без остановки всей системы.

Однако использование этих функций означает изучение того, как создавать (и завершать) процессы, как отправлять сообщения между ними и как применять возможности сопоставления с образцом для входящих сообщений.

Оболочка - это процесс

До сих пор вы работали в рамках одного процесса на протяжении всей этой книги – оболочки Erlang. Конечно, ни один из предыдущих примеров не отправлял и не получал сообщения, но оболочка – это удобное место для отправки и (по крайней мере, в целях тестирования) получения сообщений.

Первое, о чём вам нужно знать – это *идентификатор процесса*, часто называемый *pid*. Самый простой pid – ваш, поэтому в оболочке вы можете просто попробовать вызвать функцию `self()`:

```
1> self().
<0.36.0>
```

<0.36.0> – это представление оболочки в виде набора из трех целых чисел, которые предоставляют уникальный идентификатор для этого процесса. Вы можете получить другой набор чисел, когда вызовете эту функцию в оболочке. Эта группа чисел гарантированно будет уникальной в рамках этого одной виртуальной машины Erlang и не будет постоянной в будущем. Erlang использует pid для внутреннего использования, но хотя вы можете читать их в оболочке, вы не можете вводить pid непосредственно в оболочку или в функции. Erlang предпочитает, чтобы вы рассматривали pid как абстракцию, хотя если вы действительно хотите адресовать процесс по его номерам pid, вы можете использовать для этого функцию оболочки `pid/3`.

Каждый процесс получает свой собственный pid и они работают, как адреса почтовых ящиков. Ваши программы будут отправлять сообщения от одного процесса другому, отправляя их на pid. Когда этот процесс решит проверить свой почтовый ящик, он получит и обработает сообщения.

Erlang, однако, никогда не сообщит, что отправка сообщения не удалась, даже если pid не указывает на реальный процесс. Также не сообщается, что процесс был проигнорирован. Вы сами должны убедиться, что ваши процессы функционируют правильно.

Pid может даже идентифицировать процессы, запущенные на нескольких компьютерах в кластере. Вам нужно будет проделать больше работы для настройки кластера, но вам не придется выбрасывать код, который вы написали, основанных на работе с pid и процессами, построенных на них, когда в этом будет необходимость.

Синтаксис отправки сообщения довольно прост: функция или переменная, содержащая pid, плюс оператор отправки (!) и сообщение.

```
2> self() ! test1.
test1
3> Pid=self().
<0.36.0>
4> Pid ! test2.
test2
```

Строка 2 отправляет сообщение в оболочку, содержащую атом `test1`. Строка 3 назначает pid для оболочки, полученной с помощью функции `self()`, переменной с именем Pid, а затем строка 4 использует эту переменную Pid для отправки сообщения, содержащего атом `test2`. (Оператор ! всегда возвращает сообщение, поэтому оно появляется сразу после отправки в строках 2 и 4.)

Куда делись эти сообщения? Что с ними случилось? Прямо сейчас они просто ждут в почтовом ящике оболочки, ничего не делая.

Существует функция оболочки – `flush()`, которую вы можете использовать для просмотра содержимого почтового ящика, хотя она также удаляет эти сообщения из почтового ящика. При первом использовании вы получите отчет о том, что находится в почтовом ящике, но во второй раз сообщения уже не будет – они уже прочитаны.

```
5> flush().
Shell got test1
Shell got test2
ok
6> flush().
ok
```

Правильный способ чтения сообщений из почтового ящика, который дает вам возможность что-то делать с сообщениями – это конструкция `receive… end`. Она помещает содержимое сообщения в переменную и позволяет вам обрабатывать его. Вы можете проверить это в оболочке. Первое из следующих выражений просто сообщает, что это было за сообщение, а второе ожидает число и удваивает его.

```
7> self() ! test1.
test1
8> receive X -> X end.
test1
9> self() ! 23.
23
10> receive Y -> 2*Y end.
46
```

Все идет нормально. Однако, если вы сделаете ошибку – вы проверите наличие сообщений, а ничего нет, то оболочка просто зависнет. На самом деле, он ждет, когда что-то придет в почтовый ящик. И в этот момент происходит бесконечное ожидание. Самый простой выход из этого – нажать Ctrl-G, а затем набрать q. Вам придется перезапустить Erlang. Обратите внимание, что переменные, которые мы использовали в выражении `receive… end` (X и Y) стали связанными, поэтому их уже нельзя использовать далее.

Процессы порождённые в модуле

Хотя отправка сообщений в оболочку – это простой способ увидеть, что происходит, это не особенно полезно. Процессы в своей основе являются функциями. Вы знаете, как создаются функции в модуле. Оператор `receive… end` структурирован, также как и оператор `case… end`, поэтому я думаю, вы без труда освоите работу с ним.

В примере 8-1, который приведен в *ch08/ex1-simple*, показан простой модуль содержащий функцию, которая сообщает о полученных сообщениях.

Пример 8-1. Простое определение процесса

```
-module(bounce).
-export([report/0]).

report() ->

        receive
                X -> io:format("Received ~p~n",[X])
        end.
```

Когда функция `report/0` получает сообщение, она сообщает, что получила его. Порядок действий: компиляция, затем использование функции `spawn/3`, которая превращает функцию в автономный процесс. Аргументами для `spawn/3` являются имя модуля, имя функции и список аргументов для функции. Даже если у вас нет аргументов, вам нужно включить пустой список в квадратные скобки, а один аргумент должен быть списком из одного элемента. Функция `spawn/3` вернет `Pid`, который вы должны записать в переменную, здесь `Pid`:

```
1> c(bounce).
{ok,bounce}
2> Pid=spawn(bounce,report,[]).
<0.38.0>
```

После запуска процесса вы можете отправить сообщение этому pid, и он сообщит, что он получило его:

```
3> Pid ! 23.
Received 23
23
ok
```

Однако есть одна маленькая проблема. Процесс отчета завершен - он прошел пункт приема только один раз, и когда функция выполнила свою работу, процесс также завершается. Если вы попытаетесь отправить ему сообщение, вы получите сообщение обратно, и ничего не сообщит об ошибке, но вы также не получите никакого уведомления о том, что сообщение было получено, потому что больше ничего не прослушивается.

```
4> Pid ! 23.
23
```

Чтобы создать процесс, который продолжает обрабатывать сообщения, вам нужно добавить рекурсивный вызов, как показано в операторе `receive` в Примере 8-2, в *ch08/ex2-recursion*.

Пример 8-2. Функция, которая создает стабильный процесс

```
-module(bounce).
-export([report/0]).

report() ->
        receive
                X ->    io:format("Received ~p~n",[X]),
                        report()
        end.
```

Этот дополнительный вызов метода `report()` означает, что после того, как функция отобразит поступившее сообщение, она снова запустится и будет готова к следующему сообщению. Если вы перекомпилируете модуль bounce и создадите новый процесс в новой переменной `Pid2`, вы можете отправить ей несколько сообщений, как показано здесь.

```
5> c(bounce).
{ok,bounce}
6> Pid2=spawn(bounce,report,[]).
<0.47.0>
7> Pid2 ! 23.
Received 23
23
ok
8> Pid2 ! message.
Received message
message
```

Вы также можете передать в функцию-процесс аккумулятор от вызова к вызову, если хотите, для простого примера, отслеживать, сколько сообщений было получено этим процессом. В примере 8-3 показано добавление аргумента, в данном случае просто целое число, которое увеличивается с каждым вызовом. Вы можете найти его в *ch08/ex3-counter*.

Пример 8-3. Функция, которая добавляет счетчик к своему сообщению сообщения

```
-module(bounce).
-export([report/1]).

report(Count) ->
        receive
                X ->    io:format("Received #~p: ~p~n",[Count,X]),
                        report(Count+1)
end.
```

Результаты довольно предсказуемы, но помните, что вам нужно включить начальное значение в список аргументов в вызове `spawn/3`.

```
1> c(bounce).
{ok,bounce}
2> Pid2=spawn(bounce,report,[1]).
<0.38.0>
3> Pid2 ! test.
Received #1: test
test
ok
4> Pid2 ! test2.
Received #2: test2
test2
ok
5> Pid2 ! another.
Received #3: another
```

Что бы вы ни делали в своем рекурсивном вызове, лучше всего сохранять его простым (и предпочтительно хвостовым), поскольку их можно вызывать много, много раз в жизни процесса.

Если вы хотите создать нетерпеливые процессы, которые останавливаются после ожидания определенного количества времени для сообщения, вы должны исследовать конструкцию after в условии receive.

Вы можете написать эту функцию немного иначе, что может сделать происходящее в ней более понятным и простым для обобщения. В Примере 8-4, в *ch08/ex4-state*, показано, как использовать возвращаемое значение предложения receive. Здесь выражение `Count + 1` используется, для передачи состояния от одной итерации к следующей.

Пример 8-4. Использование возвращаемого значения условия receive в качестве состояния для следующей итерации

```
-module(bounce).
-export([report/1]).

report(Count) ->
        NewCount = receive
                X ->    io:format("Received #~p: ~p~n",[Count,X]),
                        Count + 1
        end,
        report (NewCount).
```

В этой модели все (хотя здесь только один) предложения `receive` возвращают значение, которое передаётся на следующую итерацию функции. Если вы используете этот подход, вы можете думать о возвращаемом значении предложения `receive`, как о состоянии, которое должно сохраняться между вызовами функций. Это состояние может быть намного сложнее, чем счетчик – например, это может быть кортеж, который включает ссылки на важные ресурсы или незавершенную работу.

Облегченные процессы

Если вы работали на других языках программирования, возможно, вы, возможно, беспокоитесь о производительности и сложности рассматриваемой темы. Потоки и порождение процессов общеизвестно сложны и часто медленны в других контекстах. Почему же Erlang ожидает, что приложения будут группой легко порождаемых процессов? Они запускается рекурсивно?

Да, конечно. Erlang был написан специально для поддержки этой модели, и его процессы весят меньше, чем его конкуренты. Процессы Erlang предназначены для наложения абсолютно минимальных накладных расходов. Планировщик Erlang запускает процессы и распределяет между ними время обработки, а также распределяет их по нескольким процессорам.

Конечно, можно писать процессы, которые работают плохо, и структурировать приложения так, чтобы они будут долго ждать прежде чем что-то сделать. Благодаря модели процессов Erlang, вам не нужно беспокоиться об этих проблемах только потому, что вы используете несколько процессов.

Регистрация процесса

В большинстве случаев Pid – это все, что вам нужно, чтобы найти и связаться с процессом. Тем не менее, вы, вероятно, создадите некоторые процессы, которые должны

быть более доступными для поиска. Erlang предоставляет чрезвычайно простую систему регистрации процессов: вы указываете атом и pid, и тогда любой процесс, который хочет достичь этого зарегистрированного процесса, может просто использовать атом для его поиска. Это облегчает, например, добавление нового процесса в систему и связывание его с ранее существующими процессами.

Чтобы зарегистрировать процесс, используйте встроенную функцию `register/2`. Первый аргумент – это атом, фактически имя, которое вы назначаете процессу, а второй аргумент – это `pid` процесса. Как только вы зарегистрируете его, вы можете отправлять ему сообщения, используя атом вместо pid:

```
1> Pid1=spawn(bounce,report,[1]).
<0.33.0>
2> register(bounce,Pid1).
true
3> bounce ! hello.
Received #1: hello
hello
ok
4> bounce ! "Really?".
Received #2: "Really?"
"Really?"
ok
```

Если вы попытаетесь вызвать процесс, который не существует (или произошел сбой), вы получите ошибку (аргументы некорректны):

```
6> zingo ! test.
** exception error: bad argument
     in operator !/2
            called as zingo ! test
```

Если вы попытаетесь зарегистрировать процесс на имя, которое уже используется, вы также получите сообщение об ошибке, но если процесс завершился (или произошел сбой), имя фактически больше не используется, и вы можете перерегистрировать его.

Вы также можете использовать `whereis/1` для получения `pid` для зарегистрированного процесса (или неопределенного, если процесс не зарегистрирован с этим атомом) и отменить регистрацию `unregister/1`, чтобы удалить процесс из списка регистрации, не прекращая его.

```
5> GetBounce = whereis(bounce).
<0.33.0>
6> unregister(bounce).
true
7> TestBounce = whereis(bounce).
undefined
8> GetBounce ! "Still there?".
Received #3: "Still there?"
"Still there?"
ok
```

Если вы хотите увидеть, какие процессы зарегистрированы, вы можете использовать команду оболочки regs().

Если вы работали на других языках программирования, весьма вероятно, знакомы с незыблимой истинной – «не используйте глобальные переменные – это черето ошибками». Вам может быть интересно, почему же Erlang разрешает общесистемный

список таких процессов. В конце концов, большая часть этой книги была посвящена изоляции изменений и минимизации общего контекста.

Однако если вы думаете о зарегистрированных процессах как об услугах, а не о функциях, то тогда всё проясняется. Зарегистрированный процесс, по сути, представляет собой сервис, публикуемый для всей системы, который можно использовать в разных контекстах. Используемые экономно, зарегистрированные процессы создают надежные точки входа для ваших программ, что может быть очень ценно по мере увеличения размера и сложности вашего кода.

Когда процессы прекращают работать

Процессы – весьма хрупки. Если есть ошибка, функция завершает выполение выражений и процесс завершается. В примере 8-5, в *ch08/ex5-division*, показана функция `report/0`, которая может прерваться, если получит нечисловое входное значение, который не является числом.

Пример 8-5. Хрупкая функция

```
-module(bounce).
-export([report/0]).

report() ->
        receive
                X ->    io:format("Divided to ~p~n",[X/2]),
                        report()
end.
```

Если вы скомпилируете и запустите этот (преднамеренно) хрупкий код, то вы обнаружите, что он работает хорошо до тех пор, пока вы отправляете ему только цифры. Отправьте что-нибудь еще, и вы увидите сообщение об ошибке в оболочке, и больше никаких ответов от этого pid. Процесс прекратил работу.

```
1> c(bounce).
{ok,bounce}
2> Pid3=spawn(bounce,report,[]).
<0.38.0>
3> Pid3 ! 38.
Divided to 19.0
38
ok
4> Pid3 ! 27.56.
Divided to 13.78
27.56
ok
5> Pid3 ! seven.
=ERROR REPORT==== 24-Aug-2016::20:59:43 ===
Error in process <0.38.0> with exit value: {badarith,[{bounce,report,0,[{file,
"bounce.erl"},{line,6}]}]}
seven
6> Pid3 ! 14.
14
```

Если вы углубитесь в изучение модели процесса Erlang, вы обнаружите, что принцып «let it crash» не является необычным дизайнерским решением в Erlang, хотя способность терпеть такие вещи и продолжать работу требует некоторых дополнительных усилий. В Главе 9 вы узнаете, как найти и устранить различного рода ошибки.

Взаимодействие процессов

Отправка сообщений процессам Erlang, как вы поняли весьма простое действие. Сложность в том, как на него ответить не имея адрес отправителя. Отправка сообщения без добавления pid отправителя – это все равно, что оставить телефонное сообщение без указания вашего собственного номера: это может доставлено, но получатель не сможет ответить вам на него.

Чтобы знать какому процессу можно отправить ответ, без регистрации большого количества процессов в системе, необходимо включить в сообщения pid. Передача pid требует добавления аргумента к сообщению. Начнём с простого теста, который вызывает в оболочке. Пример 8-6, в *ch08/ex6-talk*, основывается на модуле `drop` из примера 3-2. Добавим функцию `drop/0`, которая получает сообщения, и удалим функцию `fall_velocity/2` из экспорта.

Пример 8-6. Процесс, который отправляет сообщение обратно процессу, который вызвал его

```
-module(drop).
-export([drop/0]).

drop() ->
        receive
                {From, Planemo, Distance} ->
                        From ! {Planemo, Distance, fall_velocity(Planemo, Distance)},
                        drop()
        end.

fall_velocity(earth, Distance) when Distance >= 0 -> math:sqrt(2 * 9.8 * Distance);
fall_velocity(moon, Distance) when Distance >= 0 -> math:sqrt(2 * 1.6 * Distance);
fall_velocity(mars, Distance) when Distance >= 0 -> math:sqrt(2 * 3.71 * Distance).
```

Работу функционала модуля легко проверить из оболочки:

```
1> c(drop).
{ok,drop}
2> Pid1=spawn(drop,drop,[]).
<0.38.0>
3> Pid1 ! {self(), moon, 20}.
{<0.31.0>,moon,20}
4> flush().
Shell got {moon,20,8.0}
ok
```

В примере 8-7, который вы найдете в *ch08/ex7-talkingProcs*, показан процесс, который вызывает этот процесс для демонстрации того, что он может работать не только с оболочкой.

Пример 8-7. Вызов процесса из процесса и сообщение о результатах

```
-module(mph_drop).
-export([mph_drop/0]).

mph_drop() ->
        Drop=spawn(drop,drop,[]),
        convert(Drop).

convert(Drop) ->
        receive
                {Planemo, Distance} ->
                        Drop ! {self(), Planemo, Distance},
                        convert(Drop);
                {Planemo, Distance, Velocity} ->
                        MphVelocity= 2.23693629 * Velocity,
                        io:format("On ~p, a fall of ~p meters yields a velocity of ~p mph.~n",
```

```
                              [Planemo, Distance, MphVelocity]),
            convert(Drop)
    end.
```

Функция `mph_drop/1` порождает процесс `drop:drop/0` при его первой настройке, используя тот же модуль, который вы видели в примере 8-6, и сохраняет pid в `Drop`. Затем она вызывает `convert/1`, который также будет рекурсивно прослушивать сообщения.

Если вы не отделите инициализацию от рекурсивного слушателя, ваш код будет работать, но он будет создавать новый процесс `drop:drop/0` каждый раз, когда обрабатывает сообщение, а не используя один и то же повторно.

Условие `receive` полагается на вызов из оболочки (или другого процесса), включающий только два аргумента, в то время как процесс `drop:drop/0` возвращает результат с тремя. Когда условие `receive` получает сообщение с двумя аргументами, оно отправляет сообщение в Drop, идентифицируя себя как отправителя и передавая аргументы. Когда Drop возвращает сообщение с результатом (receive получает сообщение с тремя аргументами), условие `receive` сообщает о результате, преобразуя скорость в мили в час. (Да, он оставляет метрику расстояния, но делает скорость более понятной для американцев.)

Поскольку ваш код становится все более сложным, вы, вероятно, захотите использовать более явные флаги о типе информации, содержащейся в сообщении, например об атомах.

Работа с этим примером из оболочки выглядит следующим образом:

```
1> c(drop).
{ok,drop}
2> c(mph_drop).
{ok,mph_drop}
3> Pid1=spawn(mph_drop,mph_drop,[]).
<0.59.0>
4> Pid1 ! {earth,20}.
On earth, a fall of 20 meters yields a velocity of 44.289078952755766 mph.
{earth,20}
5> Pid1 ! {mars,20}.
On mars, a fall of 20 meters yields a velocity of 27.250254686571544 mph.
{mars,20}
```

Этот простой пример может выглядеть, как просто более сложная версия вызова функции, но есть критическое различие. Работая в оболочке, где больше ничего не работает, может создатся впечатление, что результат будет возвращаться очень быстро – настолько быстро, что в оболочке будет напечатан ответ прежде, чем будет напечатан запрос. Технически, все действия, которые происходят в системе – это серия асинхронных вызовов. Сообщения извлекаются из почтового ящика, как только они там появляются.

Оболочка отправила сообщение в Pid1, идентификатор процесса для `mph_drop:mph_drop/0`. Этот процесс отправил сообщение Drop, идентификатор процесса для drop:drop/0, который `mph_drop:mph_drop/0` установил, когда он был создан. Этот процесс возвратил другое сообщение в `mph_drop:convert/1`, которое принимает сообщения, где ответы отображаются в стандартном выводе (в данном случае в оболочке). Эти сообщения прошли и были быстро обработаны. Всё может быть не так в системах с

тысячами или миллионами сообщений. Это сообщения могли бы быть разделены многими сообщениями и появиться позже.

Наблюдение за вашими процессами

Erlang предоставляет простой, но мощный инструмент для отслеживания ваших процессов и наблюдения за тем, что происходит. `observer` предлагает минимальный графический интерфейс, который позволяет вам посмотреть текущее состояние ваших процессов и увидеть, что происходит. В зависимости от того, как вы установили Erlang, вы можете запустить его с панели инструментов, но вы всегда можете запустить его из оболочки:

```
6> observer:start().
ok
```

Вы увидите что-то вроде рисунка 8-1, представляющее обзор вашей системы Erlang. Чтобы перейти к процессам, перейдите на вкладку «Процессы», а затем щёлкните на имени столбца заголовка таблицы «Name or Initial Func», чтобы отсортировать список по имени процесса. Возможно, вам придется немного прокрутить вниз, чтобы найти процес drop:drop/0, но вы увидите нечто похожее на рисунок 8-2.

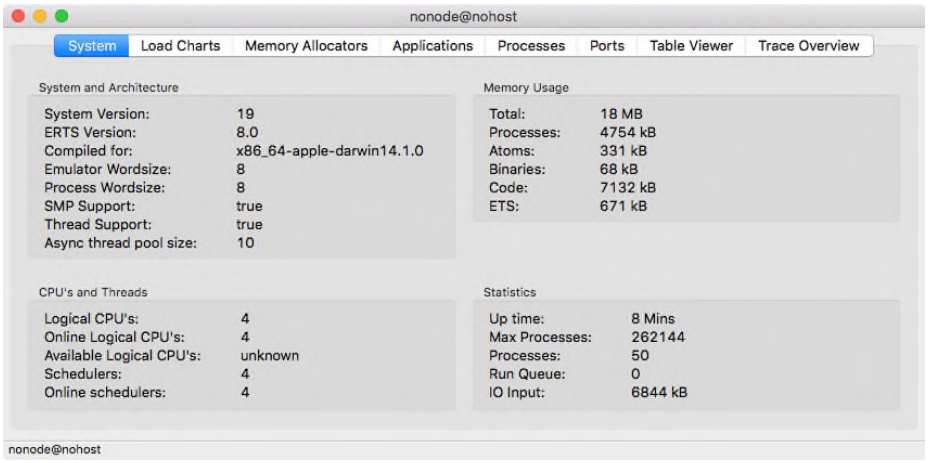


Рисунок 8-1. Наблюдатель при запуске

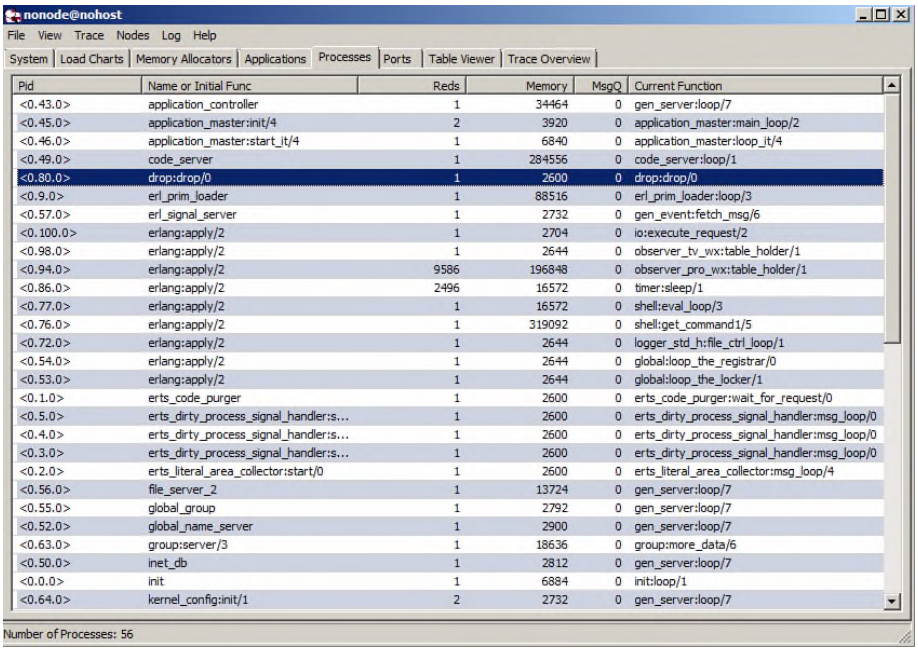


Рисунок 8-2. Окно процесса наблюдателя, отсортированное по имени процесса

Список процессов полезен, но `observer` также позволяет вам заглянуть внутрь активности процессов. Если вы дважды щелкните по процессу, скажем, `mph_drop:mph_drop/0`, вы получите некоторую основную информацию о процессе, как показано на рисунке 8-3.

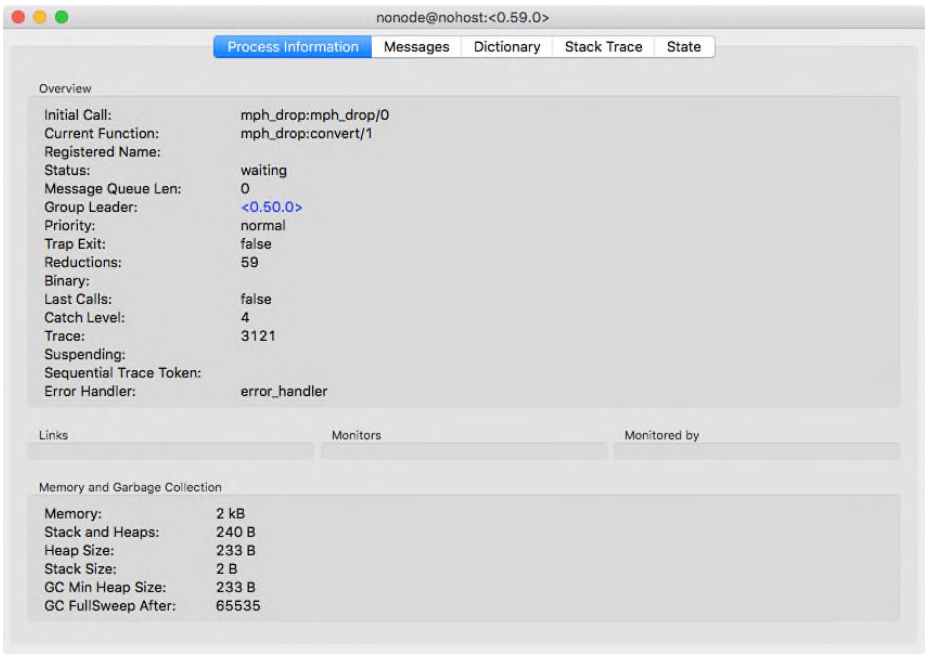


Рисунок 8-3. Присмотритесь к mph_drop

Однако для выяснения того, что делает ваш процесс, необходимо включить трассировку. Сначала найдите процесс `mph_drop:mph_drop/0` в списке процессов. Щелкните по нему правой кнопкой мыши. Затем выберите «Trace selected processes by name (all nodes)» и выберите параметры, показанные на рисунке 8-4. Затем нажмите OK. Вы вернетесь в главное окно, где вам нужно щелкнуть вкладку «Trace Overview». Затем нажмите «Start Trace». Вы получите предупреждающее сообщение, но вы можете проигнорировать его. Откроется окно «Trace Log», в котором, вероятно, будет что-то вроде «Dropped Messages».

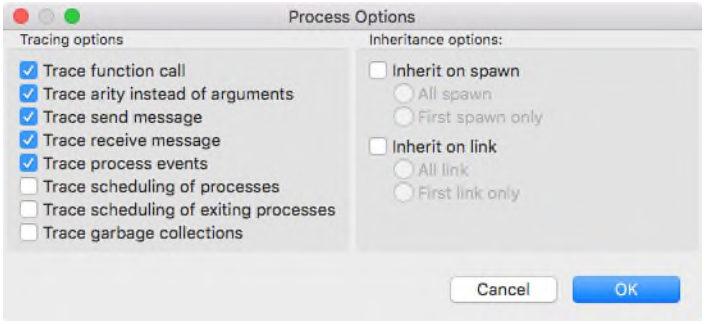


Рисунок 8-4. Основные параметры трассировки

Теперь вы можете посмотреть, как проходят сообщения.

```
7> Pid1 ! {earth,20}.
On earth, a fall of 20 meters yields a velocity of 44.289078952755766 mph.
{earth,20}
8> Pid1 ! {mars,20}.
On mars, a fall of 20 meters yields a velocity of 27.250254686571544 mph.
{mars,20}
```

Окно Observer «Trace Log» для этого процесса обновится, чтобы показать сообщения и вызовы, как показано на рисунке 8-5. Так же, как в синтаксисе языка Erlang, «!» означает, что сообщение отправлено. «<<» означает, что сообщение получено.

Цепочка сообщений начинается с вызова из оболочки на <0.59.0>, с кортежем, содержащим атом earth и 20. <0.59.0> отправляет этот же кортеж на <0.60.0>, который отправляет обратно метрическую версию расчет скорости. Поскольку трассировка соответствует только <0.59.0>, об этом сообщается с помощью «<<», получения, с кортежем, содержащим три значения. Три значения означают, что пришло время сообщить вычисляемое значение. io:format выводит на консоль результаты вычислений. Информативная структура (кортеж), начинающаяся с атома io_request, возвращает результат обратно в оболочку, а затем весь процесс повторяется для вызова для входного значения mars.

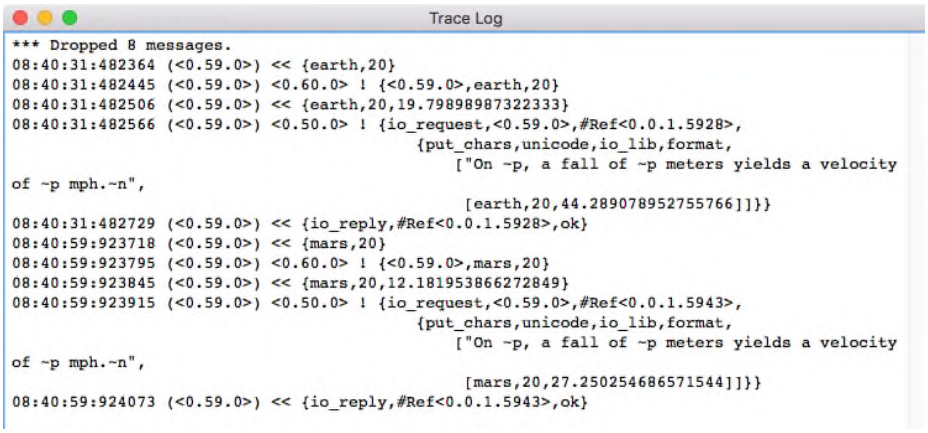


Рисунок 8-5. Отслеживание вызовов при отправке сообщения mph_drop

`observer` – это, как правило, самое простое и удобное средство, которое можно использовать, когда вам трудно понять, что происходит в нутри ваших процессов.

Ошибки и связанные процессы

Когда вы отправляете сообщение, то в оболочке вы всегда будете видеть его, как возвращаемое значение. Однако это не означает, что все нормально, и сообщение было получено и обработано правильно. Если вы отправите сообщение, которое не соответствует шаблону, в процессе получения, ничего не произойдет (на данный момент, по крайней мере), когда сообщение попадает в почтовый ящик. Просто оно не вызывает активность. Отправка сообщения, которое проходит через сопоставление с образцом, но создает ошибку, остановит процесс, в котором произошла ошибка.

Сообщения, которые не соответствуют шаблону в `receive` не исчезают; они просто задерживаются в почтовом ящике без обработки. Обновив процесс новой версией кода, сообщения будут извлечены ещё раз.

Поскольку процессы хрупки, вам может понадобиться, чтобы ваш код знал, когда произошел сбой другого процесса. В этом случае, если некорректные входные данные останавливают `drop:drop/0`, не имеет смысла оставлять зависший процесс `mph_drop:convert/1`. Вы можете увидеть, как это работает в оболочке и `observer`. Сначала запустите `observer` и создайте `mph_drop:mph_drop/0`.

```
1> observer:start().
ok
2> Pid1=spawn(mph_drop,mph_drop,[]).
<0.83.0>
```

Если вы переключитесь на вкладку «Processes» и нажмете на заголовок колонки «Pid», чтобы отсортировать данные, вы увидите что-то похожее на рисунок 8-6 в Observer. Затем введите в процесс неверные данные, атом (`zoids`) вместо числа для Distance, и обновлённый вид таблице в Observer будет больше похож на рисунок 8-7.

```
2> Pid1 ! {zoids,20}.
> Pid1 ! {moon,zoids}.
{moon,zoids}
4>

=ERROR REPORT==== 19-Dec-2016::21:03:36 ===
Error in process <0.85.0> with exit
value:{badarith,[{drop,fall_velocity,2,[{file,"drop.erl"},{line,12}]},
{drop,drop,0,[{file,"drop.erl"},{line,7}]}]}
```

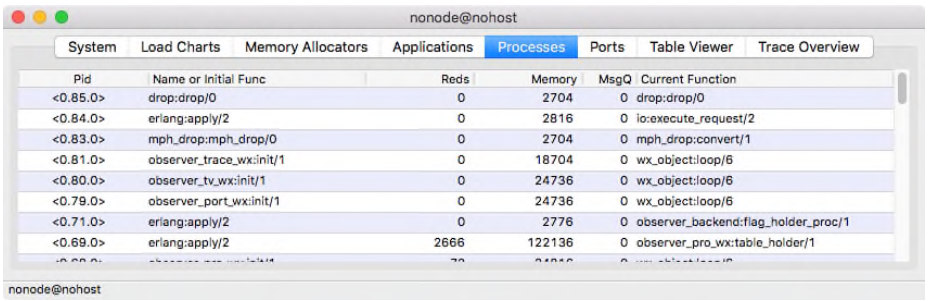


Рисунок 8-6. Список рабочих процессов

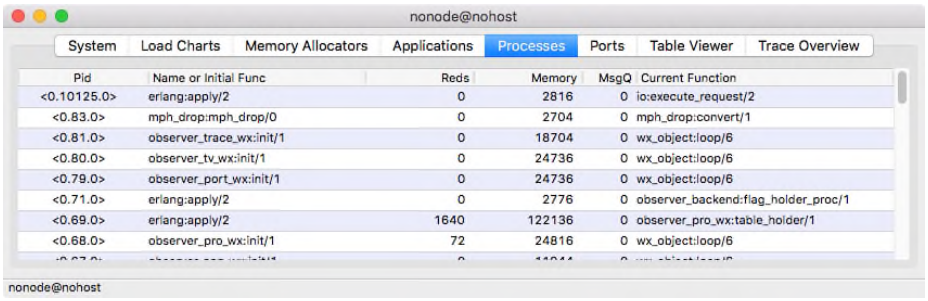


Рисунок 8-7. Из таблицы пропал только процесс drop:drop/0

Поскольку оставшийся процесс `mph_drop:mph_drop/0` теперь бесполезен. Было бы лучше остановить его при сбое `drop:drop/0`. Erlang позволяет указать эту зависимость с помощью ссылки. Самый простой способ сделать это, избегая возможных условий гонки, это использовать функцию `spawn_link/3` вместо уже изветной нам `spawn/3`. Это показано в примере 8-8, который вы можете найти в *ch08/ex8-linking*.

Пример 8-8. Вызов связанного процесса из процесса и прекращение связанных процессов при сбое

```
-module(mph_drop).
-export([mph_drop/0]).

mph_drop() ->
        Drop=spawn_link(drop,drop,[]),
        convert(Drop).

convert(Drop) ->
        receive
                {Planemo, Distance} ->
                        Drop ! {self(), Planemo, Distance},
                        convert(Drop);
                {Planemo, Distance, Velocity} ->
                        MphVelocity= 2.23693629 * Velocity,
                        io:format("On ~p, a fall of ~p meters yields a velocity of ~p
mph.~n",[Planemo, Distance, MphVelocity]),
                convert(Drop)
        end.
```

Теперь, если вы скомпилируете и протестируете код, то с помощью `observer`, вы увидите, что оба процесса исчезают при сбое `drop:drop/0`, как показано на рисунке 8-8.

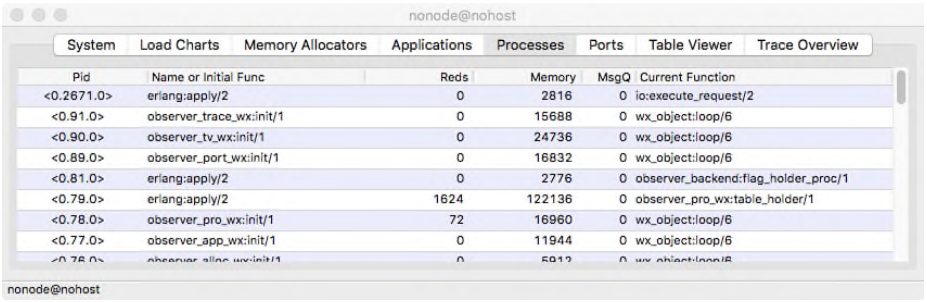


Рисунок 8-8. Оба процесса теперь прекращаются при возникновении ошибки

Ссылки являются двунаправленными. Если вы завершите процесс `mph_drop:mph_drop/0`, например, с помощью функции `exit(Pid1, kill)`. Процесс `drop:drop/0` также исчезнет. (`kill` - самая жесткая причина для выхода, и ее нельзя отследить, потому что иногда вам действительно нужно остановить процесс.)

Такой тип отказа может быть не тем, что вы имеете в виду, когда думаете о связывании процессов. Это поведение по умолчанию для связанных процессов Erlang и имеет смысл

во многих контекстах, но вы также можете имеете возможность обрабатывать выходы из процесса. При сбое процесса Erlang он отправляет сообщение собъяснением другим связанным с ним процессам в виде кортежа. Кортеж содержит атом EXIT, Pid сбойного процесса и ошибку как сложный кортеж. Если ваш процесс настроен на прерывание выходов через вызов `process_flag(trap_exit, true)`, эти сообщения об ошибках поступают в виде сообщений, а не просто убивают ваш процесс.

В примере 8-9 в *ch08/ex9-trapping* показано, как изменяется начальный метод `mph_drop/0`, чтобы включить этот вызов для установки флага процесса, добавляется ещё одно условие в `receive`, которая будет отслеживать сообщения о завершении связанного процесса и более аккуратно сообщать о них.

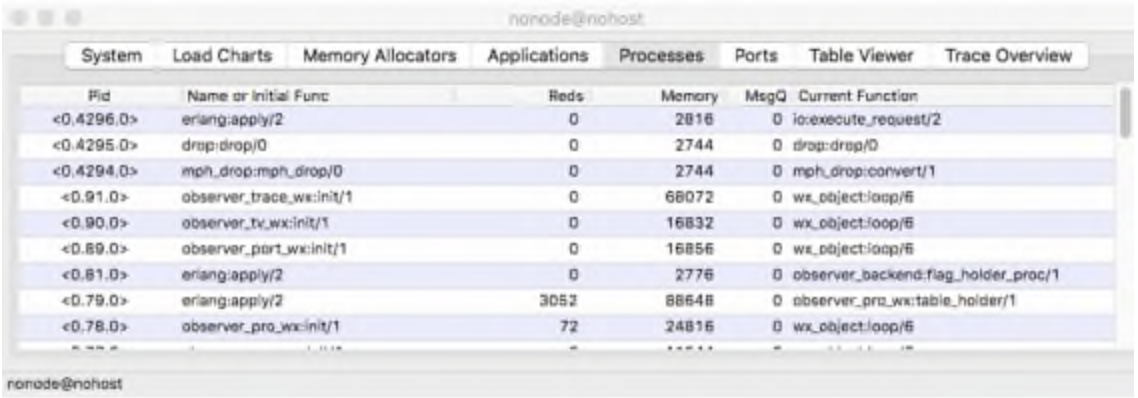
Пример 8-9. Перехват ошибок, сообщение об ошибке и выход

```
-module(mph_drop).
-export([mph_drop/0]).

mph_drop() ->
        process_flag(trap_exit, true),
        Drop=spawn_link(drop,drop,[]),
        convert(Drop).

convert(Drop) ->
        receive
                {Planemo, Distance} ->
                        Drop ! {self(), Planemo, Distance},
                        convert(Drop);
                {'EXIT', Pid, Reason} ->
                        io:format("FAILURE: ~p died because of ~p.~n",[Pid, Reason]);
                {Planemo, Distance, Velocity} ->
                        MphVelocity= 2.23693629 * Velocity,
                        io:format("On ~p, a fall of ~p meters yields a velocity of ~p
mph.~n",[Planemo, Distance, MphVelo
convert(Drop)
        end.
```

Рисунок 8-9. Процессы до ошибки. Обратите внимание на Pid для `drop:drop/0`.



Если вы запустите этот код и передадите неверные данные, метод `mph_drop:convert/1` сообщит об ошибке (в основном, дублируя оболочку), прежде чем аккуратно завершить работу.

```
1> c(mph_drop).
{ok,mph_drop}
2> Pid1=spawn(mph_drop,mph_drop,[]).
<0.45.0>
3> Pid1 ! {moon,20}.
On moon, a fall of 20 meters yields a velocity of 17.89549032 mph.
{moon,20}
4> Pid1 ! {moon,zoids}.
FAILURE: <0.46.0> died because of {badarith,
[{drop,fall_velocity,2,
[{file,"drop.erl"},{line,12}]},
```

```
{drop,drop,0,
[{file,"drop.erl"},{line,7}]}]}.
=ERROR REPORT==== 19-Dec-2016::21:13:46 ===
Error in process <0.46.0> with exit value:
{badarith,[{drop,fall_velocity,2,[{file,"drop.erl"},{li
{moon,zoids}
```

Более надежная альтернатива создаст новую переменную `Drop`, порождая новый процесс. Эта версия, показанная в примере 8-10, которую вы можете найти в *`ch08/ex10-resilient`*, более жизнестойкое. При получении сообщения об ошибки, создаётся новый процесс и ссылка на него (`NewDrop`) используется в дальнейшей работе.

Пример 8-10. Перехват ошибок, сообщение об ошибке и запуск нового процесса

```
-module(mph_drop).
-export([mph_drop/0]).

mph_drop() ->
        process_flag(trap_exit, true),
        Drop=spawn_link(drop,drop,[]),
        convert(Drop).

convert(Drop) ->
        receive
                {Planemo, Distance} ->
                        Drop ! {self(), Planemo, Distance},
                        convert(Drop);
                {'EXIT', _Pid, _Reason} ->
                        NewDrop=spawn_link(drop,drop,[]),
                        convert(NewDrop);
                {Planemo, Distance, Velocity} ->
                        MphVelocity= 2.23693629 * Velocity,
                        io:format("On ~p, a fall of ~p meters yields a velocity of ~p
mph.~n",[Planemo, Distance, MphVelo
                        convert(Drop)
        end.
```

Если вы скомпилируете и запустите пример 8-10, вы увидите что-то похожее на рисунок 8-9, когда вы запустите Observer, перейдете в раздел Процессы и отсортируете по `Pid`. Если вы передадите неверные данные, как показано в строке 6 в следующем примере кода, вы все равно получите сообщение об ошибке из оболочки, но процесс будет работать нормально. Как показывает `observer` на рисунке 8-10, он запустил новый процесс для обработки вычислений `drop:drop/0`, и, как показано в строке 8, он работает как его предшественник.

```
1> c(drop).
{ok,drop}
2> c(mph_drop).
{ok,mph_drop}
3> observer:start().
ok
4> Pid1=spawn(mph_drop,mph_drop,[]).
<0.4294.0>
5> Pid1 ! {moon,20}.
On moon, a fall of 20 meters yields a velocity of 17.89549032 mph.
{moon,20}
6> Pid1 ! {mars,20}.
On mars, a fall of 20 meters yields a velocity of 27.250254686571544 mph.
{mars,20}
7> Pid1 ! {mars,zoids}.
{mars,zoids}
8>
=ERROR REPORT==== 19-Dec-2016::21:18:38 ===
Error in process <0.4295.0> with exit value:
{badarith,[{drop,fall_velocity,2,[{file,"drop.erl"},{line,13}]},
{drop,drop,0,[{file,"drop.erl"},{line,7}]}]}
Pid1 ! {moon,20}.
On moon, a fall of 20 meters yields a velocity of 17.89549032 mph.
{moon,20}
```

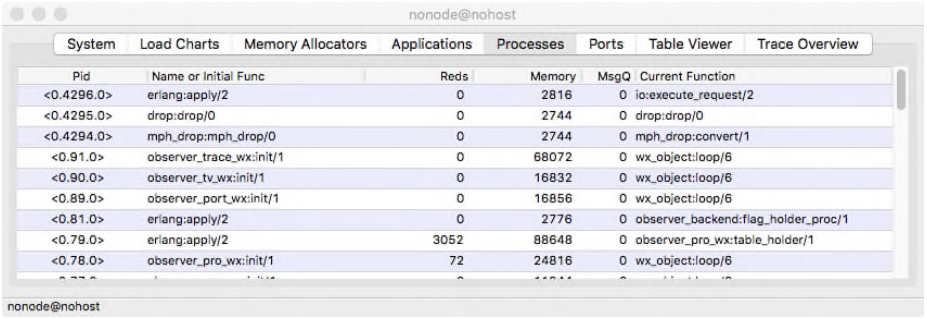


Рисунок 8-9. Процессы до ошибки. Обратите внимание, что Pid для drop:drop/0

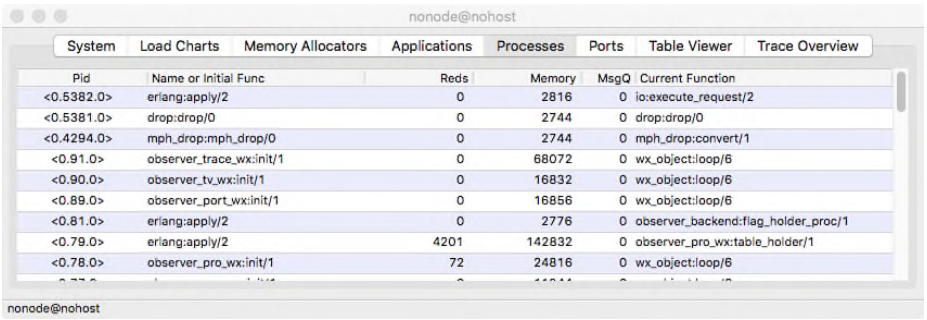


Рисунок 8-10. Процессы после ошибки. Обратите внимание на изменение в Pid для drop:drop/0

Erlang предлагает гораздо больше вариантов управления процессами. Вы можете удалить ссылку с помощью `unlink/1` или установить соединение для простого просмотра процесса с помощью `erlang:monitor/2`. Если вы хотите завершить процесс, вы можете использовать `exit/1` внутри этого процесса или `exit/2`, чтобы указать процесс и причину из другого процесса.

Создание приложений, способных переносить сбои и восстанавливать их функциональные возможности, лежит в основе надежного программирования Erlang. Разработка в этом стиле, вероятно, является большим скачком для большинства программистов, чем просто переход к функциональному программированию на Erlang, но именно здесь становится очевидной истинная сила Erlang.

Вы можете узнать больше о работе с простыми процессами в главе 4 «Erlang Programming» (O'Reilly); Глава 8 «Programming Erlang» (Pragmatic); Раздел 2.13 «Programming Erlang» (Pragmatic); и главы 10 и 11 «Learn You Some Erlang For Great Good!» (No Starch Press).

Глава 9. Исключения, ошибки и отладка

Хотя «Let it crash» - это великолепная концепция, вы бы всё-таки хотели управлять работой вашего приложения. Несмотря на то, что можно писать код, который постоянно ломается и восстанавливается, может быть проще, писать и поддерживать код, который явно обрабатывает сбои там, где это происходит. Erlang создан для решения таких проблем, как сетевые ошибки, но вы, наверное, не захотите специально добавлять свои ошибки в задачи. Если вы решите работать с ошибками, вы наверняка захотите отследить их в своем приложении.

Признаки ошибок

Как вы уже видели, некоторые виды ошибок не позволят Erlang компилировать ваш код. Компилятор также выдаст вам предупреждения о потенциальных проблемах, таких как переменные, которые объявлены, но никогда не используются. Основные виды ошибок: ошибки времени выполнения, которые возникают во время работы кода и могут фактически остановить функцию или процесс, и логические ошибки, которые могут не прервать вашу программу, но могут вызвать более сложные для выявления ошибки.

Логические ошибки часто сложнее всего диагностировать, требуют тщательного обдумывания и, возможно, некоторого времени работы с отладчиком, файлами журналов или набором тестов. Кроме этого простые математические ошибки могут потребовать много усилий, чтобы их обнаружить. Иногда проблемы связаны со временем работы последовательности операций, при которых они не соответствуют ожидаемым. В тяжелых случаях условия гонки могут привести к блокировкам и остановкам. Более простые проблемы могут также привести к плохим результатам и путанице.

Ошибки во время выполнения также могут раздражать, но они более управляемы. В некотором смысле вы можете рассматривать обработку ошибок во время выполнения, как часть логики вашей программы, хотя, пожалуйста, не увлекатесь этим. В Erlang, в отличие от многих других сред, классическая обработка ошибок может дать лишь незначительные преимущества по сравнению с тем, чтобы позволить ошибке прекратить работающий процесс и далее решать проблему на уровне процесса, как показано в примере 8-10.

Отследить ошибки во время выполнения во время их возникновения

Если вы хотите отлавливать ошибки времени выполнения близко к месту их возникновения, конструкция try… catch позволяет обернуть подозрительный код и обработать его (если таковой имеются). Это дает понять как компилятору, так и программисту, что происходит что-то необычное и позволяет вам справиться с любыми неприятными последствиями этой работы. Для простого примера, вернитесь к Примеру 3-1, в котором рассчитана скорость падения без учета возможности того, что она может быть отрицательной. Функция `math:sqrt/1` выдаст ошибку `badarith`, если у нее отрицательный аргумент. В Примере 4-2 эта проблема не возникала при применении средств защиты, но если вы хотите сделать больше, чем блокировать, вы можете применить более прямой подход с помощью try и catch, как показано в Примере 9-1. (Вы можете найти его в *ch09/ex1-tryCatch*.)

Пример 9-1. Использование try и catch для обработки возможной ошибки

```
-module(drop).
-export([fall_velocity/2]).

fall_velocity(Planemo, Distance) ->
        Gravity = case Planemo of
        earth -> 9.8;
        moon -> 1.6;
        mars -> 3.71
        end,
        try math:sqrt(2 * Gravity * Distance) of
                Result -> Result
        catch
                error:Error -> {error, Error}
        end.
```

Сам расчет теперь находится внутри блока `try`. Если вычисление выполнено успешно, будет использовано сопоставление с образцом, следующее за значением. В этом случае при вычислении создается только одно значение, поэтому при сопоставлении с переменной `Result` это значение будет помещено в `Result`, которое затем станет возвращаемым значением.

Вы можете упростить выражение try, убрав избыточное условие `Result -> Result`. Эта конструкция try… catch дает те же результаты, что и в примере 9-1.

```
try math:sqrt(2 * Gravity * Distance)
catch
        error:Error -> {error, Error}
end.
```

Если вычисление не удается, в этом случае из-за отрицательного аргумента в игру вступает совпадение с шаблоном в условии `catch`. В этом случае атом `error` будет классом или типом исключения (он может быть `error`, `throw` или `exit`), а переменная `Error` будет собирать сведения об ошибке. Затем оно возвращает кортеж, начинающегося с атома `error` и содержит переменную `Error`, которая объяснит тип ошибки.

Вы можете попробовать следующее в командной строке:

```
1> c(drop).
{ok,drop}
2> drop:fall_velocity(earth,20).
19.79898987322333
3> drop:fall_velocity(earth,-20).
{error,badarith}
```

Когда расчет будет успешным, вы просто получите результат. Когда происходит сбой, кортеж сообщает тип ошибки, вызвавшей проблему. Это не полное решение, но это фундамент, на котором вы можете строить.

Вы можете иметь несколько операторов в `try` (так же, как и в операторе case), разделенных запятыми. При написании, кода, использование конструкций `try` позволяет локализовать места возможных ошибок. В случае если, вам необходимо ослеживать ошибочный атом, который описывает небесное тело, в этом случае ваш код может выглядеть следующим образом:

```
fall_velocity(Planemo, Distance) ->
        try
                Gravity = case Planemo of
                        earth -> 9.8;
                        moon -> 1.6;
                        mars -> 3.71
                end,
                math:sqrt(2 * Gravity * Distance)
        of
                Result -> Result
        catch
```

```
            error:Error -> {error, Error}
    end.
```

Если вы передадите функции неподдерживаемое Planemo, вы увидите, что код обработал эту проблему, по крайней мере, как только вы перекомпилируете код для использования новой версии:

```
4> drop:fall_velocity(jupiter,20).
** exception error: no case clause matching jupiter
in function drop:fall_velocity/2 (drop.erl, line 5)
5> c(drop).
{ok,drop}
6> drop:fall_velocity(jupiter,20).
{error,{case_clause,jupiter}}
```

Ошибка `case_clause` указывает, что не было подходящего шаблона в условиях оператора case, а после перекомпиляции мы получили кортеж с описанием ошибки в виде кортежа.

Вы также можете иметь несколько шаблонов сопсоставления с образцом в блоке `catch`. Если ваши шаблоны не совпадают с ошибкой в условиях `catch`, то ошибка становится ошибкой времени выполнения.

Если код, потенциально, может вызвать необрабатываемое исключение, целесообразно после блока `catch` разместить блок `after`. Код в этом блоке будет гарантированно выполняться независимо от того, будет ли попытка выполнения кода успешной или неудачной. Это может быть хорошим местом для устранения любых побочных эффектов кода. Этот блок не влияет на возвращаемое значение оператора try.

Erlang также включает в себя более старую конструкцию `catch`, которая не использует `try`. Вы можете встретить её в устаревшем коде. Эта констркция не рекомендована к использованию, так как считается устаревшей. Она менее сложена, но и менее читабелена, хотя в Главе 11 мы покажем, как её использовать в оболочке.

Оператор throw – выбрасывание исключений

Возможно, вы захотите создать свои собственные ошибки или, по крайней мере, сообщить о результатах так, чтобы механизм `try… catch` мог работать с ними. Функция `throw/1` позволяет вам создавать исключения, которые затем могут быть перехвачены (или оставлены для прекращения текущего процесса и, возможно, для отправки в оболочку). Часто в качестве аргумента используется кортеж, позволяющий вам предоставить более подробную информацию об исключении, но вы можете использовать все, что считаете нужным. Обратите внимание, что если вы обрабатываете исключения, вы определенно хотите обрабатывать исключения близко к тому месту, где вы хотите их вызвать. Пример использования `throw/1` в оболочке:

```
1> throw(my_exception).
** exception throw: my_exception
```

Вы можете создать шаблон для сгенерированных исключений в предложении catch, используя throw вместо error:

```
try some:function(argument)
catch
       error:Error -> {found, Error};
       throw:Exception -> {caught, Exception}
end;
```

Вероятно, вам следует использовать `throw` в случаях, когда вы не можете придумать лучший подход для сигнализации в вашем коде. Используйте его только там, где вы знаете, что у вас есть код, который его перехватит. Злоупотребление им – не очень хорошая практика.

В предыдущем примере использовались атотмы found и caught для обработки различных видов исключений. В большинстве случаев, использование атома error вполне достаточно для обоих ошибок (является общепринятой практикой).

Логирование работы процессов и отчётов об ошибках

Функция `io:format/2` полезна для простого взаимодействия с оболочкой, но по мере роста ваших программ (особенно когда они становятся распределенными процессами), переход к стандартному выводу с меньшей вероятностью даст вам необходимую информацию. Erlang предлагает набор функций для более формального ведения журнала. Они могут подключаться к более сложным системам журналирования, но с ними легко начать как способ структурирования сообщений из вашего приложения.

Три функции в модуле `error_logger`[1] предоставляют три уровня отчетности:

`info_msg`
: Для регистрации обычных новостей, которые не требуют вмешательства.

`warning_msg`
: Для новостей это подходит меньше. Это подходит для важных сообщений.

`error_msg`
: Что-то просто перестало работать.

Как и в `io:format`, есть две версии каждой функции. Более простая версия принимает только строку, а более сложная версия принимает строку и список аргументов, которые добавляются к этой строке. Оба используют ту же структуру форматирования, что и io:format, так что вы можете в значительной степени заменить любые вызовы `io:format`, которые вы использовали для прямой отладки. Все они возвращают `ok`.

Как вы можете видеть, эти вызовы создают отчеты, которые визуально отличаются друг от друга, хотя предупреждения и ошибки обрабатываются одинаково:

```
1> error_logger:info_msg("The value is ~p. ~n",[360]).
=INFO REPORT==== 12-Dec-2016::08:00:41 ===
The value is 360.
```

[1] В Erlang / OTP 21,0, был добавлен новый API для регистрации. Старый модуль error_logger по-прежнему можно использовать в устаревшем коде, но события журнала перенаправляются в новый API-интерфейс Logger. Новый код должен напрямую использовать Logger API.

error_logger больше не запускается по умолчанию, но запускается автоматически при добавлении обработчика событий с error_logger:add_report_handler/1,2. Затем модуль error_logger также добавляется в качестве обработчика к новому регистратору.

См. logger(3) и главу «Регистрация ошибок» в Руководстве пользователя для получения дополнительной информации.

```
ok
2> error_logger:warning_msg("Connection lost; will retry.").
=WARNING REPORT==== 12-Dec-2016::08:01:33 ===
Connection lost; will retry.
ok
Connection lost; will retry.ok
3> error_logger:error_msg("Unable to read database.~n").
=ERROR REPORT==== 12-Dec-2016::08:03:45 ===
Unable to read database.

ok
```

Более многословная форма дает лишь незначительное улучшение по сравнению с `io:format`, так зачем вам ее использовать? Потому что у Erlang многое скрывает. По умолчанию, когда Erlang запускается, он устанавливает модуль `error_logger` для вывода отчетов об ошибках в оболочку. Однако, если вы включите SASL – библиотеки поддержки архитектуры системы (System Architecture Support Libraries) Erlang – вы сможете подключить эти уведомления к гораздо более сложной системе для регистрации распределенных процессов. (Если вы просто хотите записать свои ошибки на диск, вам следует изучить функцию `error_logger:logfile/1`.)

Вывести работу регистратора из рабочего состояния весьма легко – с помощью ошибок в формате вывода, поэтому, если вы хотите более надежную систему регистрацию, вы можете использовать более простые вресии этих функций.

Хотя регистрируемая информация полезна, нет ничего необычного в том, чтобы писать код с небольшими ошибками. Вы можете засорить свой код избыточными отчетами. Решение – переключаться между набором инструментов отладки Erlang.

Отладка Erlang программ с помощью графического интерфейса

Графический отладчик Erlang – самое удобное место для начала, требующее лишь незначительных изменений в том, как вы компилируете код для начала работы. В этой демонстрации будет использоваться тот же код, который показан в примере 9-1, но вам необходимо скомпилировать его с флагом `debug_info` и запустить отладчик с помощью `debugger:start()`. Вы увидите окно, подобное показанному на рисунке 9-1.

```
1> c(drop, [debug_info]).
{ok,drop}
2> debugger:start().
{ok,<0.71.0>}
```

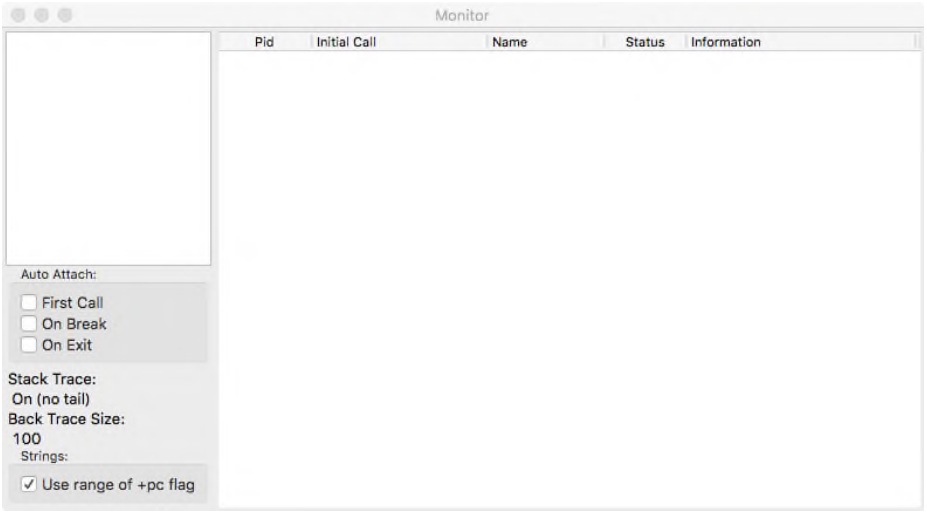


Рисунок 9-1. Окно отладчика при первом открытии

Когда он открывается впервые, окно отладчика выглядит довольно пусто. Вам нужно указать, что вы хотите отслеживать, выбрав «Module» ➜ «Interpret…» (в зависимости от вашей операционной системы это может быть обычное меню или кнопка в верхнем ряду). Как показано на рисунке 9-2, вы должны выбрать файл исходного кода модуля drop (drop.erl). Вам может потребоваться перейти к нему, если вы не запускались в том же каталоге. Далее в окне «Interpret Modules», выбрав файл исходного кода модуля, нажмете «OK». Имя модуля появится в левой части окна «Monitor», как показано на рисунке 9-3.

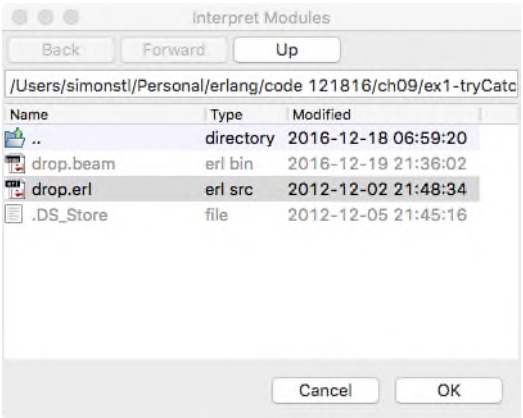


Рисунок 9-2. Выбор модуля

Единственный способ узнать, что вы на самом деле выбрали модуль - это его появление на панели окна «Monitor». Если ваши окна свёрнуты и вы не видите содержание окна, легко подумать, что ничего не произошло. Но это не так!

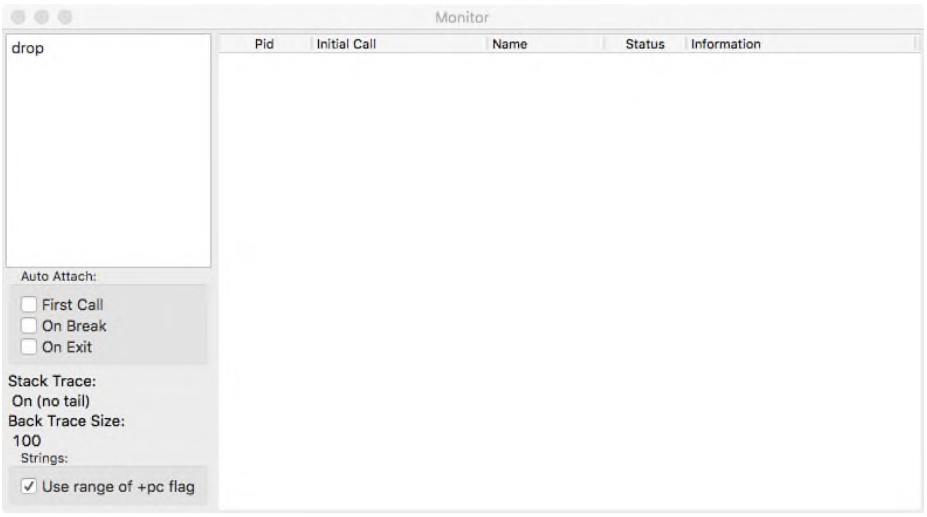


Рисунок 9-3. Имя модуля отображается слева

Как только имя модуля появится в левой части окна «Monitor», отладчик уже готов к его отладке. Однако вы должны сообщить отладчику, что именно вы хотите увидеть. Если вы дважды щелкнете по названию модуля (drop), вы увидите окно «View -> Module drop», показанное на рисунке 9-4, где показан его код.

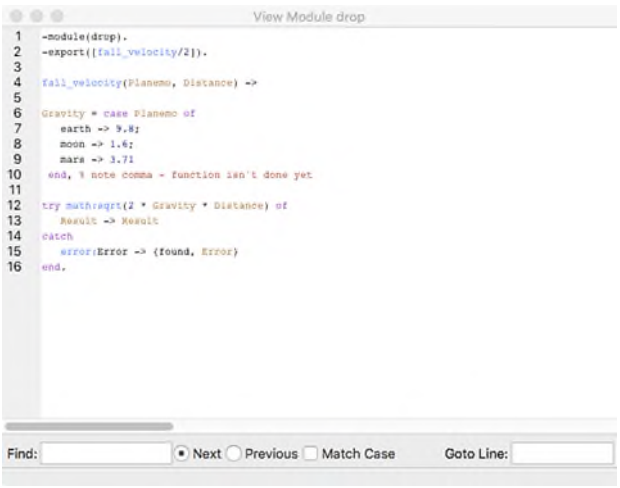


Рисунок 9-4. Просмотр кода модуля drop

Вы можете добавить точку останова, щелкнув по строке кода и выбрав «Line Break…» в меню «Break». Вы увидите диалоговое окно «Line Break», показанное на рисунке 9-5, с настройками по умолчанию. Вы можете просто нажать «OK», и в раскрывающемся окне «View Module drop» появится точка останова, как показано на рисунке 9-6. Вы можете закрыть это окно и просто оставить окно монитора открытым.

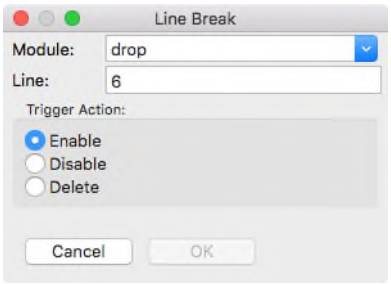


Рисунок 9-5. Диалоговое окно «Line Break» для установки точек останова

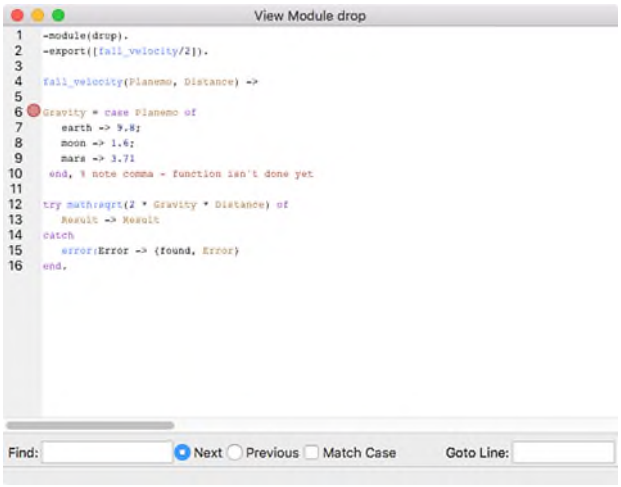


Рисунок 9-6. Окно «View Module drop» с установленной точкой останова на строке 6

Теперь, если вы вернетесь в оболочку и введёте следующую команду:

```
3> drop:fall_velocity(earth,20).
```

Ваш код просто остановит выполнение команд в строке 6. Ничего не происходит, так как точка останова остановила выполнение. Однако в окне «Monitor» вы увидите новую запись в таблице справа, как показано на рисунке 9-7. Если дважды щелкнуть на эту новую запись, вы попадете в окно «Attach Process» на рис. 9-8, которое позволяет вам выполнять ваш код строка за строкой.

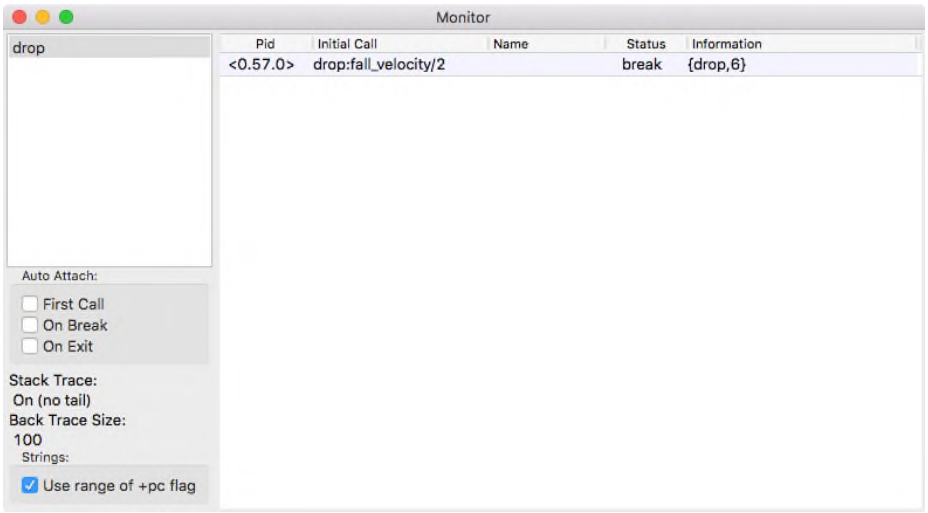


Рисунок 9-7. Окно «Monitor» демонтрирует выполнение кода

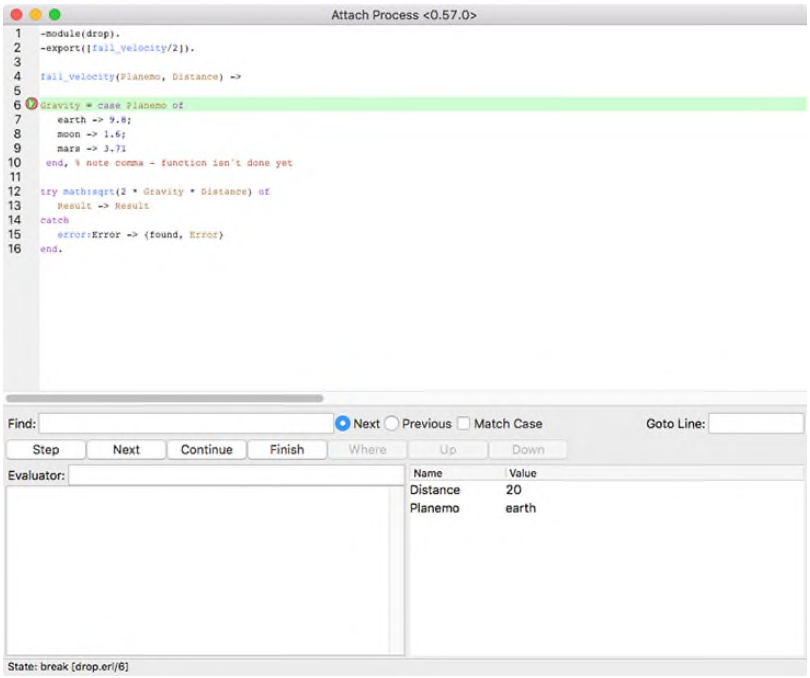


Рисунок 9-8. Код и связанные значения в окне «Attach Process»

Как только у вас откроется окно «Attach Process», вы можете построчно перебирать код (или указывать продолжить), используя кнопки в средней строке:

Step (шаг)
: Выполнить текущую строку кода и перейти к следующей строке. Если следующая строка кода, которая должна быть выполнена, находится в другой функции (а эта функция находится в модуле, скомпилированном для отладки), вы пройдетесь к коду этой функции.

Next (следующий)
: Выполните текущую строку кода и перейдите к следующей строке кода в этом модуле.

Continue (продолжить)
: Завершите поэтапное пошаговое выполнение и просто выполните код как обычно.

Finish (конец)
: Аналогично *Continue*, но продолжается только для текущей функции. Отладчик может продолжать работать с кодом, когда он возвращается из этой функции. (Это полезно, когда вы вошли в функцию, детали которой вас не интересуют, и у вас не хватает терпения ждать.)

Where (где)
: Перемещает окно кода в текущую исполняемую строку.

Up и *Down* (вверх и вниз)
: Перемещает окно кода вверх или вниз на уровне функции в стеке.

На рисунках с 9-9 по 9-12 показаны результаты пошагового выполнения кода, выполняемого вызовом drop:fall_velocity(earth, 20). Обратите внимание на изменение связанных переменных и окончательный возврат: не интерпретируется на рис. 9-12 после завершения вызова.

Каждый раз, когда код приостанавливается, вы можете использовать панель «Evaluator» для выполнения собственных вычислений с использованием значений и функций, доступных в текущей области. В отличие от некоторых отладчиков, вы не можете изменить значение связанных переменных здесь, потому что вы не можете изменять значения переменных в Erlang в целом.

В итоге результат также печатается в окне оболочки:

```
3> drop:fall_velocity(earth,20).
19.79898987322333
```

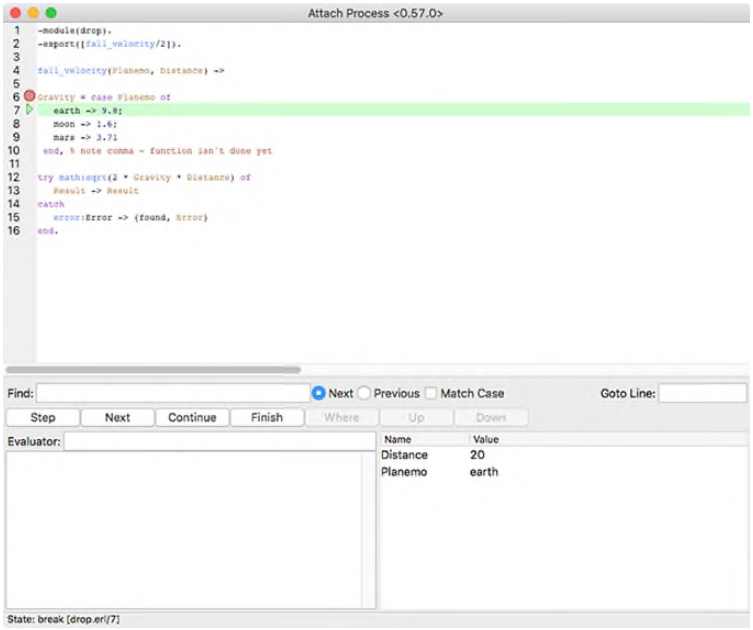


Рисунок 9-9. Переход к анализу сопоставления с шаблоном в строке 7

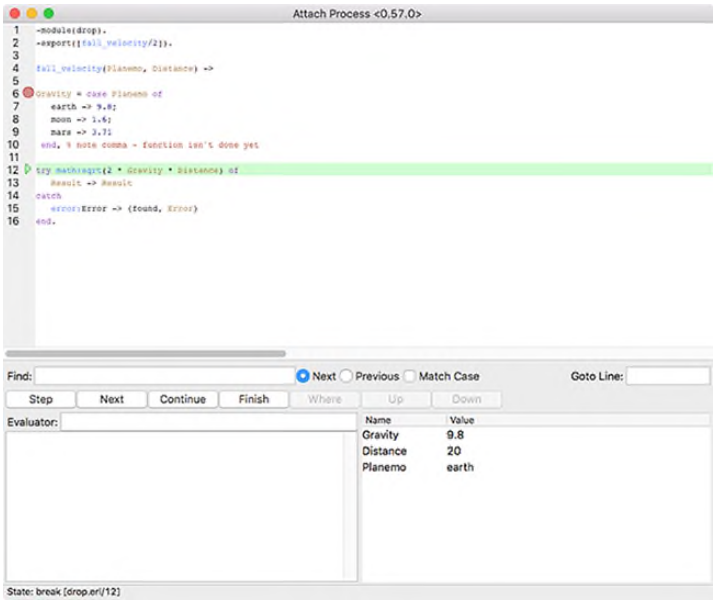


Рисунок 9-10. Следующий шаг: оператор try и расчет в строке 12 с дополнительными связанными значениями

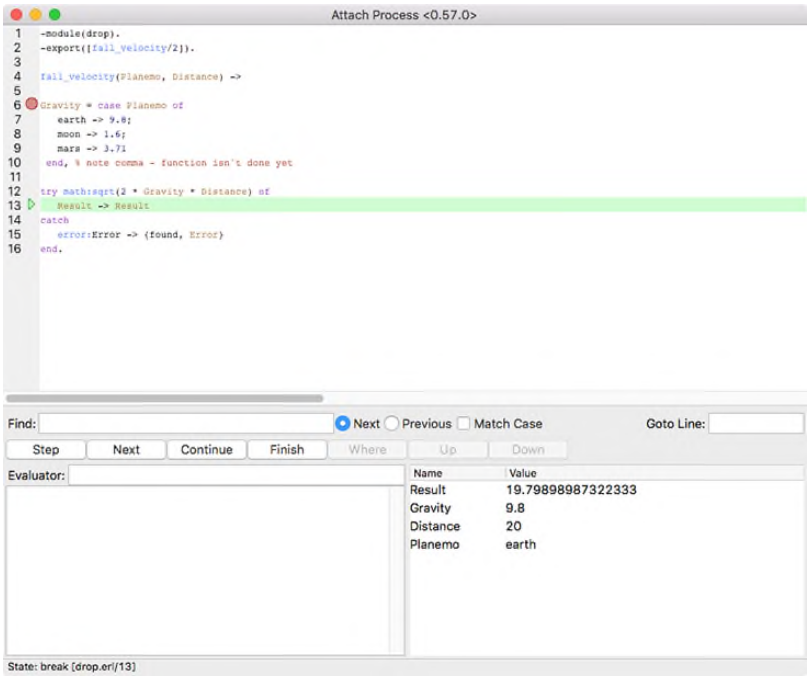


Рисунок 9-11. Успешный расчет приводит к строке 13, которая предоставляет возвращаемое значение

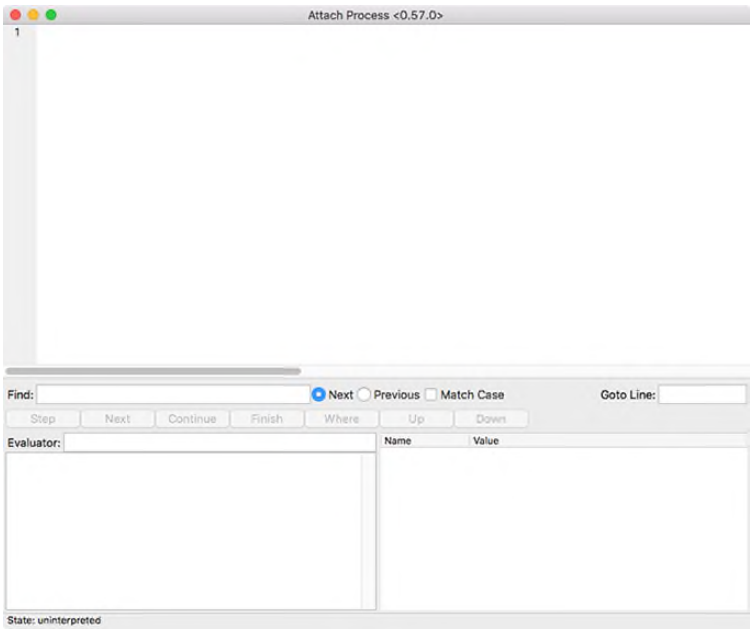


Рисунок 9-12. Когда вызов функции завершается, окно становится пустым

Отладчик предлагает гораздо больше возможностей, чем было рассмотрено в этом простом примере. Этот базовый набор действий поможет вам начать с ним работу.

Регистрация (трассировка) сообщений

Erlang также предлагает широкий спектр инструментов для трассировки кода, как с другим с помощью встроенных функций `trace` и `trace_pattern`, так и с текстовым отладчиком/монитором. Модуль dbg – это самый простой инструмент для начала работы. Он позволяющее вам указать, что вы хотите отследить, и показывающее вам результаты в оболочке.

Простое место для начала – это отслеживание сообщений, отправляемых между процессами. Вы можете использовать `dbg:p` для отслеживания сообщений, отправленных между процессом `mph_drop`, определенным в примере 8-8, и процессом `drop` из примера 8-6. После компиляции модулей – флаг `debug_info` здесь не нужен – вы вызываете `dbg:tracer()`, чтобы начать сообщать информацию трассировки в оболочку. Затем вы запускаете процесс `mph_drop` как обычно и передаете этот pid процессу `dbg:p/2`. Вторым аргументом здесь будет атом `m`, что означает, что трассировка должна сообщать о сообщениях.

```
1> c(drop).
{ok,drop}
2> c(mph_drop).
{ok,mph_drop}
3> dbg:tracer().
{ok,<0.43.0>}
4> Pid1=spawn(mph_drop,mph_drop,[]).
<0.46.0>
5> dbg:p(Pid1,m).
{ok,[{matched,nonode@nohost,1}]}
```

nonode@nohost просто ссылается на текущий узел Erlang, когда вы не распределяете обработку по нескольким системам. Если вы используете распределенную систему Erlang, у вас будет несколько узлов, каждый из которых имеет свою собственную независимую среду исполнения Erlang со своим собственным именем.

Теперь, когда вы отправляете сообщение процессу `mph_drop`, вы получаете набор отчетов о результирующем потоке сообщений. (<0.46.0> – это процесс `mph_drop`, а <0.47.0> – это процесс `drop`.)

```
6> Pid1 ! {moon,20}.
(<0.46.0>) << {moon,20}
(<0.46.0>) <0.47.0> ! {<0.46.0>,moon,20}
On moon, a fall of 20 meters yields a velocity of 17.89549032 mph.
(<0.46.0>) << {moon,20,8.0}
(<0.46.0>) <0.24.0> ! {io_request,<0.46.0>,<0.24.0>,
                              #Ref<0.3930322788.4247519235.200494>,
                              {put_chars,unicode,io_lib,format,
                                     ["On ~p, a fall of ~p meters yields a velocity of ~p ph.~n",
                                     [moon,20,17.89549032]]}}
{moon,20}
(<0.46.0>) << {io_reply,<0.24.0>,ok}
(<0.46.0>) << timeout
```

«<<» указывает на pid процесса, который получил сообщение. Отправка указывается, как обычно, с pid, за которым следует «!» с последующим сообщением. В этом случае:

- mph_drop (<0.46.0>) получает сообщение в виде кортежа {moon, 20}.
- он отправляет сообщение, кортеж {<0.46.0>, moon, 20}, процессу- адресату drop по pid <0.47.0>.
- на этом этапе на консоль выводится сообщение от `mph_drop` о том, что “On moon, a fall of 20 meters yields a velocity of 17.89549032 mph.” («На Луне при падении предмета с 20 метров получает скорость 17,89549032 миль в час»). Обратите внимание, что сообщение выводится раньше, чем информация трассировки. Остальная часть сообщения показывает, как этот отчет попал туда.
- mph_drop получает ответ - кортеж {moon, 20, 8.0} (от drop).
- затем вызывается `io:format/2`, который запускает другой набор сообщений процесса для создания отчета, заканчивая атомом timeout, который ничего не делает.

Отчеты о трассировке поступают через некоторое время после фактического выполнения кода, но они делают поток сообщений понятным. Вы захотите научиться использовать функционал модуля `dbg` во многих его вариантах для отслеживания вашего кода и, возможно, в конечном итоге захотите использовать шаблоны сопоставления и сами функции трассировки для создания более элегантных систем для просмотра конкретного кода.

Просмотр вызовов функций

Если вы просто хотите отслеживать перемещение аргументов между вызовами функций, вы можете использовать трассировщик, чтобы сообщить о последовательности вызовов. Глава 4 продемонстрировала рекурсию и сообщила о результатах с помощью `io:format`. Есть еще один способ увидеть эту работу, снова используя модуль `dbg`.

Пример 4-11, восходящий факториальный калькулятор, начинался с вызова `fact:factorial/1`, который затем вызывал `fact:factorial/3` рекурсивно. `dbg` позволит вам увидеть реальные вызовы функций и их аргументы, смешанные с отчетами вызовов `io:format`. (Исходный код вы сможете найти это в *ch09/ex4-dbg*.)

Функции трассировки немного сложнее, чем трассировки сообщений, потому что вы не можете просто передать функции `dbg:p/2` pid. Как показано в строке 3 в следующем примере кода, вы должны сообщить, что хотите, чтобы он сообщал обо всех процессах (all) и их вызовах (c). Как только вы это сделаете, вы должны указать, о каких вызовах вы хотите, чтобы он сообщал, используя `dbg:tpl`, как показано в строке 4. Он принимает имя модуля (`fact`), имя функции (`factorial`) и, необязательно, спецификацию соответствия, которая позволяет вам указать аргументы более точно. Вариации этой функции также позволяют указать арность.

Поэтому включите трассировщик, скажите, что вы хотите следить за вызовами функций, и укажите функцию (или функции через несколько вызовов `dbg:tpl`) для просмотра. Затем вызовите функцию, и вы увидите список вызовов.

```
1> c(fact).
fact.erl:13: Warning: variable 'Current' is unused
fact.erl:13: Warning: variable 'N' is unused
{ok,fact}
2> dbg:tracer().
{ok,<0.38.0>}
3> dbg:p(all, c).
{ok,[{matched,nonode@nohost,26}]}
4> dbg:tpl(fact, factorial, []).
{ok,[{matched,nonode@nohost,2}]}
5> fact:factorial(4).
1 yields 1!
(<0.31.0>) call fact:factorial(4)
(<0.31.0>) call fact:factorial(1,4,1)
2 yields 2!
(<0.31.0>) call fact:factorial(2,4,1)
3 yields 6!
(<0.31.0>) call fact:factorial(3,4,2)
4 yields 24!
(<0.31.0>) call fact:factorial(4,4,6)
Finished.
(<0.31.0>) call fact:factorial(5,4,24)
24
```

Вы можете видеть, что последовательность здесь немного запутанная: отчеты о трассировке появляются немного позже, чем результат `io:format` из отслеживаемой функции. Поскольку трассировка выполняется в отдельном процессе (в pid <0.38.0>) от

функции (в pid <0.31.0>), ее отчеты могут не выстраиваться один за одним (или совсем не попорядку, хотя обычно порядок всё же соблюдается). Когда вы закончите трассировку, вызовите `dbg:stop/0` (если вы захотите перезапустить трассировку с той же настройкой) или `dbg:stop_clear/0` (если вы знаете, что при повторном запуске вы захотите снова все настроить).

Модуль `dbg` и функции `erlang:trace`, на которых он построен, являются невероятно мощными инструментами.

Вы можете узнать больше об обработке ошибок в главах 3 и 17 «Erlang Programming» (O’Reilly); Глава 4 и Раздел 18.2 «Programming Erlang» (Pragmatic); Раздел 2.8 и главы 5 и 7 «Erlang and OTP in Action» (Manning); и главы 7 и 12 «Learn You Some Erlang For Great Good!» (No Starch Press).

Глава 10. Хранение структурированных данных

Кортежи и списки являются мощными инструментами для создания сложных структур данных, но до сих пор в этой истории отсутствуют две ключевые части. Во-первых, кортежи являются относительно анонимными структурами. Опора на определенный порядок и количество компонентов в кортежах может создать серьезные проблемы с обслуживанием. Это также означает, что кортежи не позволяют вам ссылаться на содержимое по имени: вам всегда нужно знать местоположение. Во-вторых, несмотря на общее предпочтение Erlang избегать побочных эффектов, хранение и обмен данными является фундаментальным побочным эффектом, необходимым для широкого спектра проектов.

Три инструмента обеспечивают большую поддержку структурированных данных. Карты работают хорошо, когда вы хотите обратиться к возможно различной информации через единый список имен. Записи помогут вам создавать упорядоченные наборы информации. Erlang Term Storage (ETS) поможет вам хранить и управлять этими наборами, а база данных Mnesia предоставляет дополнительные функции для надежного распределенного хранения.

Отображение ваших данных

Ссылка на данные по их месту в списке или кортеже может быстро обременять память и код программиста, особенно если данные приходят и уходят. Erlang 17 и позже решают эту общую проблему с помощью новой структуры данных – `map`(карты). Обработка карт выполняется медленнее, чем обработка списков или кортежей, но более плавное совпадение с ключевыми проблемами обработки данных по-прежнему делает ее мощным дополнением.

Создание карты требует немного другого синтаксиса представления ключей и значений:

```
1> Planemos = #{ earth => 9.8, moon => 1.6, mars => 3.71 }.
#{earth => 9.8,mars => 3.71,moon => 1.6}
```

Карта `Planemos` теперь содержит три элемента с атомами в качестве ключей. Ключ `earth` имеет значение `9.8`, `mars` `3.71` и `moon` `1.6`. Значения являются гравитационными константами и вы, как программист, наверняка это знаете (из курса физики). В отличие от записей, которые появятся дальше, разные части карт не получают имен.

Самый простой способ извлечь значения из карты – с помощью функции get модуля maps.

```
2> maps:get(moon, Planemos).
1.6
```

Если вам нужно добавить значение в карту, этого нельзя сделать, но, конечно, вы можете создать новую карту, которая содержит значения старой карты плюс новая пара ключ-значение, или карту, которая содержит старую карту без какой-либо пары.

```
3> MorePlanemos = maps:put(venus, 8.9, Planemos).
#{earth => 9.8,mars => 3.71,moon => 1.6,venus => 8.9}
4> maps:get(venus, MorePlanemos).
8.9
5> FewerPlanemos = maps:remove(moon, MorePlanemos).
#{earth => 9.8,mars => 3.71,venus => 8.9}
```

Если вы попробуете найти значение по ключу, которого нет в карте, то вы получите ошибку:

```
6> maps:get(moon, FewerPlanemos).
** exception error: {badkey,moon}
      in function maps:get/2
            called as maps:get(moon,#{earth => 9.8,mars => 3.71,venus => 8.9})
```

Хотя большая часть возможностей карт может быть использована тольско с помощью функций модуля `maps` и ещё не имеет собственного особого синтаксиса в языке Erlang, вы можете использовать сопоставление с образом:

```
17> #{earth := Gravity} = Planemos.
#{earth => 9.8,mars => 3.71,moon => 1.6}
18> Gravity.
9.8
```

Если вам нужен гибкий способ связать значения с ключами, карты могут быть тем, что вы ищете. Модуль `maps` также предоставляет множество инструментов, поддерживающих обработку карт с функциями более высокого порядка. Если вы хотите больше структуры, вы, вероятно, хотите рассмотреть записи.

Карты появились в Erlang 17, но все еще медленно развиваются и интегрируются в язык. Функции в модуле `maps` работают. К сожалению, фунционал этого модуля небольшой (только начиная с версии 19 в этом модуле начали появляться дополнительные функции). Если вы найдете примеры в Интернете или даже в книгах (Джо Армстронга), которые не работают, они, весьма вероятно, просто предвосхитили возможности развития Erlang.

От кортежей к записям

Кортежи позволяют создавать сложные структуры данных, но заставляют вас полагаться на постоянство порядка и количества элементов. Если вы изменяете последовательность элементов в кортеже или хотите добавить элемент, вы должны проверить весь код, чтобы убедиться, что изменение не повредять уже имеющееся решение. По мере роста ваших проектов, особенно если вам необходимо обмениваться структурами данных с кодом, который вы не контролируете, вам потребуется более безопасный способ хранения и обработки информации.

Записи позволяют создавать структуры данных, которые используют имена для связи с данными, а не по порядку. Вы можете читать, записывать и сопоставлять данные шаблона в записи, не беспокоясь о подробностях того, где в кортеже скрывается поле или кто-то добавил новое поле.

Записи – это всё ещё кортежи, и иногда Erlang выставляет их вам. Не пытайтесь использовать представление кортежа напрямую, иначе вы добавите все потенциальные проблемы использования кортежей в небольшой дополнительный синтаксис использования записей.

Настройка записей

Использование записей требует, чтобы Erlang сообщил о них с помощью специальной декларации. Это выглядит как объявление -module или -export, но это объявление -record:

```
-record(planemo, {name, gravity, diameter, distance_from_sun}).
```

Это выражение (декларация) определяет запись с именем `planemo`, содержащий поля с именами `name`, `gravity` и `distance_from_sun`. При создании новой записи, все поля будут иметь значение `undefined`. Вы также можете указать значения по умолчанию. Следующая выражение создает записи для разных башен из падающих объектов:

```
-record(tower, {location, height=20, planemo=earth, name}).
```

В отличие от объявлений `-module` или `-export`, вы часто будете обмениваться объявлениями записей между несколькими модулями и (по крайней мере для примеров в этой главе) даже использовать их в оболочке. Чтобы передача объявлений записей была надёжной - используйте собственный файл записей, заканчивающейся расширением .hrl. Вы можете поместить каждое объявление записи в отдельный файл или все в один файл, в зависимости от ваших потребностей. Для начала, чтобы увидеть, как они себя ведут, вы можете поместить оба объявления в один файл records.hrl, показанный в примере 10-1. (Вы можете найти его в *ch10/ex1-records*.)

Пример 10-1. Файл records.hrl, содержащий два не связанных объявлений записей

```
-record(planemo, {name, gravity, diameter, distance_from_sun}).
-record(tower, {location, height=20, planemo=earth, name}).
```

Возможно, вы захотите поместить отдельные записи записей в свои собственные файлы и импортировать их отдельно, вводя их только тогда, когда вам действительно нужно получить данные в или из определенного типа записи. Это может быть особенно важно, если вы смешиваете код в тех случаях, когда разные разработчики использовали одно и то же имя для типа записи, но разные базовые структуры.

Команда rr (для чтения записей) позволяет вам перенести их в оболочку:

```
1> rr("records.hrl").
[planemo,tower]
```

Оболочка теперь понимает записи с именами `planemo` и `tower`.

Вы также можете объявлять записи непосредственно в оболочке с помощью функции `rd/2`, но если вы делаете что-то большее, чем просто небольшие структурки, то проще поместить их в более надежное формальное импортированное объявление. Вы можете вызвать `rl/0`, если хотите посмотреть, какие записи определены, или `rl/1`, если хотите посмотреть, как определена конкретная запись.

Создание и чтение записей

Теперь вы можете создавать переменные, которые содержат новые записи. Синтаксис для ссылки на записи предшествует имени типа записи знаком # и заключает пары имя-значение в фигурные скобки. Например, вы можете создать башни с синтаксисом, как показано ниже:

```
2> Tower1=#tower{}.
#tower{location = undefined,height = 20,planemo = earth, name = undefined}
3> Tower2=#tower{location="Grand Canyon"}.
#tower{location = "Grand Canyon",height = 20, planemo = earth,name = undefined}
4> Tower3=#tower{location="NYC", height=241, name="Woolworth Building"}.
#tower{location = "NYC",height = 241,planemo = earth,name = "Woolworth Building"}
5> Tower4=#tower{location="Rupes Altai 241", height=500, planemo=moon, name="Piccolomini View"}.
#tower{location = "Rupes Altai 241",height = 500,planemo = moon,name = "Piccolomini View"}
6> Tower5=#tower{planemo=mars, height=500, name="Daga Vallis", location="Valles Marineris"}.
#tower{location = "Valles Marineris",height = 500, planemo = mars,name = "Daga Vallis"}
```

Эти башни (или, по крайней мере, пропущенные элементы) демонстрируют различные способы использования синтаксиса записей для создания переменных, а также взаимодействия со значениями по умолчанию:

- Строка 2 просто создает Tower1 со значениями по умолчанию. Вы можете добавить реальные значения позже.
- Строка 3 создает Tower2 с местоположением, но в остальном использует значения по умолчанию.
- Строка 4 переопределяет значения по умолчанию для местоположения, высоты и имени, но оставляет `planemo` значением по умолчанию.
- Строка 5 заменяет все значения по умолчанию новыми значениями.
- Строка 6 заменяет все значения по умолчанию, а также демонстрирует, что не имеет значения, в каком порядке вы перечисляете пары имя/значение. Erlang разберется с этим.

Вы можете читать записи записей двумя разными способами. Чтобы извлечь одно значение, вы можете использовать синтаксис с точкой (.), Который может показаться знакомым по другим языкам. Например, чтобы узнать, на какой планете находится `Tower5`, вы можете написать:

```
7> Tower5#tower.planemo.
mars
```

Вы также можете использовать сопоставление с образцом, чтобы извлечь несколько частей одновременно:

```
8> #tower{location=L5, height=H5} = Tower5.
#tower{location = "Valles Marineris",height = 500,
planemo = mars,name = "Daga Vallis"}
9> L5.
"Valles Marineris"
10> H5.
500
```

Синтаксис выглядит странно: переменная связана с правой стороны знака равенства, а не слева, как обычно.

Как всегда, вы не можете записать новое значение в существующую переменную, но вы можете создать новую запись на основе значений старой. Синтаксис, используемый в строке 13, очень похож на синтаксис, используемый для назначения содержимого поля переменной, но со значением вместо имени переменной:

```
12> Tower5.
#tower{location = "Valles Marineris",height = 500,
planemo = mars,name = "Daga Vallis"}
```

```
13> Tower5a=Tower5#tower{height=512}.
#tower{location = "Valles Marineris",height = 512,
planemo = mars,name = "Daga Vallis"}
```

Да, вам всегда нужно указывать тип записи. Это немного лишний набор текста.

Если вы когда-нибудь захотите заставить оболочку забыть о ваших записях, вы можете выполнить команду оболочки `rf()`. Ваши переменные, основанные на записях, все еще будут существовать, в виде необработанного кортежа, который вы должны избегать.

Использование записей в функциях и модулях

Записи хорошо работают и в модулях, используя те же файлы объявлений. Конечно, вы можете просто включить объявление записи в каждый модуль, который его использует, но для этого вам потребуется выследить каждое объявление и обновить его, если вы когда-нибудь захотите его изменить. Более разумный подход заключается в использовании файлов, подобных тем, которые показаны ранее. Вы можете сделать это легко с помощью одного дополнительного объявления в верхней части вашего модуля:

```
-include("records.hrl").
```

Как только вы включите объявление записи, вы можете сопоставить шаблон с записями, представленными в качестве аргументов. Самый простой способ сделать это - просто сравнить с типом записи, как показано в примере 10-2, который также находится в *ch10/ex1-records*.

Пример 10-2. Метод, котором сопоставление с шаблоном происходив в блоке входных данных

```
-module(record_drop).
-export([fall_velocity/1]).
-include("records.hrl").

fall_velocity(#tower{} = T) ->
fall_velocity(T#tower.planemo, T#tower.height).

fall_velocity(earth, Distance) when Distance >= 0 -> math:sqrt(2 * 9.8 * Distance);
fall_velocity(moon, Distance) when Distance >= 0 -> math:sqrt(2 * 1.6 * Distance);
fall_velocity(mars, Distance) when Distance >= 0 -> math:sqrt(2 * 3.71 * Distance).
```

При этом используется сопоставление с образцом, которое будет сопоставлять только записи башни, и помещает запись в переменную T. Опять же, синтаксис может показаться обратным, причем T находится справа от равенства, а не слева, но это работает. Затем, как и его предшественник в Примере 3-8, он передает отдельные аргументы в fall_velocity/2 для расчетов, на этот раз с использованием синтаксиса записи.

Короткие имена могут показаться более привлекательными, когда нужно добавлять имя типа записи при каждом использовании. В простых функциях это может работать, но в более сложных функциях короткие имена могут привести к путанице, особенно если у вас есть две переменные, содержащие записи одного типа.

Поскольку вы использовали одно и то же объявление -record и в оболочке, и в модуле, вы можете использовать созданные вами записи для проверки функции.

```
14> c(record_drop).
{ok,record_drop}
15> record_drop:fall_velocity(Tower5).
60.909769331364245
16> record_drop:fall_velocity(Tower1).
19.79898987322333
```

Функция `record_drop:fall_velocity/1`, показанная в примере 10-3, извлекает planemo и привязывает его к Planemo и height и привязывает его к Distance. Затем она возвращает скорость объекта, сброшенного с этого расстояния, как в предыдущих примерах в этой книге.

Вы также можете извлечь определенные поля из записи в сопоставлении с образцом, как показано в примере 10-3, который находится в *ch10/ex2-records*.

Пример 10-3. Метод, который в шаблон сопоставления с образцом, компоненты записит присваиваются переменным

```
-module(record_drop).
-export([fall_velocity/1]).
-include("records.hrl").

fall_velocity(#tower{planemo=Planemo, height=Distance}) ->
fall_velocity(Planemo, Distance).

fall_velocity(earth, Distance) when Distance >= 0 -> math:sqrt(2 * 9.8 * Distance);
fall_velocity(moon, Distance) when Distance >= 0 -> math:sqrt(2 * 1.6 * Distance);
fall_velocity(mars, Distance) when Distance >= 0 -> math:sqrt(2 * 3.71 * Distance).
```

Опять же, синтаксис может показаться задом наперед, но он позволяет вам извлекать отдельные поля. Вы можете взять созданные записи и передать их в эту функцию, и она сообщит вам скорость, возникающую в результате падения с вершины башни на дно.

Наконец, вы можете сопоставлять шаблоны как с полями, так и с записями в целом. Пример 10-4 в *ch10/ex3-records* демонстрирует использование этого смешанного подхода для создания более детального отклика, чем просто скорость падения.

Пример 10-4. Метод, в котором шаблон выбора по образцу соответствует всей записи, а также компонентам записи

```
-module(record_drop).
-export([fall_velocity/1]).
-include("records.hrl").

fall_velocity(#tower{planemo=Planemo, height=Distance} = T) ->
io:format("From ~s's elevation of ~p meters on ~p, the object will reach ~p m/s before crashing
in ~s.~n",[T#tower.name, Distance, Planemo, fall_velocity(Planemo, Distance), T#tower.location
]).

fall_velocity(earth, Distance) when Distance >= 0 -> math:sqrt(2 * 9.8 * Distance);
fall_velocity(moon, Distance) when Distance >= 0 -> math:sqrt(2 * 1.6 * Distance);
fall_velocity(mars, Distance) when Distance >= 0 -> math:sqrt(2 * 3.71 * Distance).
```

Если вы передадите запись башни в record_drop: fall_velocity/1, она сопоставит отдельные поля, необходимые для вычисления, и сопоставит всю запись с T, чтобы получить более интересный, но, возможно, не корректный грамматически, отчет.

```
17> record_drop:fall_velocity(Tower5).
From Daga Vallis's elevation of 500 meters on mars, the object will reach
60.909769331364245 m/s before crashing in Valles Marineris.
ok
```

```
18> record_drop:fall_velocity(Tower3).
From Woolworth Building's elevation of 241 meters on earth, the object will reach
68.72845116834803 m/s before crashing in NYC.
ok
```

`record_drop:fall_velocity/1` использует последовательность управления `~s` для вызова `io:format/2`. Он просто включает в себя содержимое строки без кавычек.

Вы можете узнать больше о работе с записями в главе 7 «Erlang Programming», в разделе 3.9 «Programming Erlang», в разделе 2.11 «Erlang and OTP in Action» и в главе 9 «Learn You Some Erlang For Great Good!».

Хранение записей в Erlang Term Storage

Erlang Term Storage (ETS) – это простое, но мощное хранилище в памяти. Оно содержит кортежи, и поскольку записи – это кортежи, они естественным образом подходят. ETS и ее двоюродный брат DETS, работающий на диске, обеспечивают (возможно, слишком) простое решение многих проблем управления данными. ETS не совсем база данных, но выполняет аналогичную работу и полезна сама по себе. Она также является основой базы данных Mnesia, которую вы увидите в следующем разделе.

Каждая запись в таблицах ETS является кортежем (или соответствующей записью), и один фрагмент кортежа обозначается как ключ. ETS предлагает несколько различных структурных решений в зависимости от того, как вы хотите обработать этот ключ. ETS может содержать четыре вида коллекций:

Наборы (set)
: Может содержать только одну запись с данным ключом. Это по умолчанию.

Упорядоченные наборы (ordered_set)
: То же, что и набор, но также поддерживает порядок обхода на основе ключей. Отлично подходит для всего, что вы хотите сохранить в алфавитном или числовом порядке.

Сумки (bag)
: Позволяет хранить более одной записи с данным ключом. Однако, если у вас есть несколько записей, которые имеют полностью идентичные значения, они объединяются в одну запись.

Дублирующиеся сумки (duplicate_bag)
: Не только позволяет хранить более одной записи с данным ключом, но также позволяет хранить несколько записей с полностью идентичными значениями.

По умолчанию таблицы ETS являются наборами, но вы можете указать одну из других опций при создании таблицы. Приведенные здесь примеры будут множествами, потому что их проще понять, но те же методы применимы ко всем четырем разновидностям таблицы.

ETS не требует, чтобы все ваши записи выглядели одинаково. Однако, когда вы начинаете, гораздо проще использовать записи одного типа или, по крайней мере, кортежи с одинаковой структурой. Вы также можете использовать любой тип значения для ключа, включая сложные структуры кортежей и списки, но, опять же, лучше не начинать с самого начала.

Все примеры в следующем разделе будут использовать тип записи `planemo`, определенный в предыдущем разделе, и данные в Таблице 10-1.

Таблица 10-1. Небесные тела для гравитационного исследования

Небесное тело	Гравитация ($м/с^2$)	Диаметр (км)	Расстояние от Солнца (10^6км)
mercury	3.7	4878	57.9
venus	8.9	12104	108.2
earth	9.8	12756	149.6
moon	1.6	3475	149.6
mars	3.7	6787	227.9
ceres	0.27	950	413.7
jupiter	23.1	142796	778.3
saturn	9.0	120660	1427.0
uranus	8.7	51118	2871.0
neptune	11.0	30200	4497.1
pluto	0.6	2300	5913.0
haumea	0.44	1150	6484.0
makemake	0.5	1500	6850.0
eris	0.8	2400	10210.0
mercury	3.7	4878	57.9
venus	8.9	12104	108.2
earth	9.8	12756	149.6
moon	1.6	3475	149.6

Создание и заполнение таблицы

Функция `ets:new/2` позволяет вам создать таблицу. Первый аргумент – это имя таблицы, а второй аргумент – список параметров. Существует множество опций, включая идентификаторы для типов таблиц, описанных выше, но две наиболее важные для начала работы – `named_table` и кортеж, начинающийся с `keypos`.

У каждой таблицы есть имя, но только с некоторыми можно связаться по имени. Если вы не указали `named_table`, имя будет установено по умолчани и доступ к данной таблице будет только внутри базы данных. Вам нужно будет использовать значение, возвращаемое `ets:new/2`, для ссылки на таблицу. Если вы укажете `named_table`, процессы смогут получить доступ к таблице без необходимости получать доступ к ней по возвращаемому значени функции.

Даже с именованной таблицей у вас все еще есть некоторый контроль над тем, какие процессы могут читать и записывать таблицу с помощью закрытых (`private`), защищенных (`protected`) и открытых (`public`) параметров.

Другим важным параметром, особенно для таблиц ETS, содержащих записи, является позиция ключа. По умолчанию ETS рассматривает первое значение в кортеже, как ключ. Описание данных реализовано в виде кортежа (которого вам не нужно изменять.). Общей практикой является использовние первого значения в кортеже для определения типа записи, но этот подход очень плохо работает в качестве ключа для записей. Использование кортежа `keypos` позволяет указать, какое значение записи должно быть ключевым.

Напомним, что формат записи для `planemo` выглядит следующим образом:

```
-record(planemo, {name, gravity, diameter, distance_from_sun}).
```

Поскольку эта таблица в основном используется для расчетов, основанных на заданной `planemo`, имеет смысл использовать имя в качестве ключа. Соответствующая декларация для настройки таблицы ETS может выглядеть следующим образом:

```
PlanemoTable=ets:new(planemos,[ named_table, {keypos, #planemo.name} ])
```

Здесь таблице присваивается имя `planemos` и использует параметр `named_table`, чтобы сделать эту таблицу видимой другим процессам, которые знают её имя. Так как по умолчанию уровень доступак к таблице `protected` - процесс может вносить измения в эту таблицу, а другие процессы могут только читать ее. Это также говорит ETS использовать поле имени в качестве ключа. Поскольку не указано иное, таблица будет обрабатываться как набор – каждый ключ сопоставляется только с одним экземпляром записи. ETS не сохраняет список, отсортированный по ключу.

После настройки таблицы, как показано в примере 10-5, вы используете функцию ets:info/1, чтобы проверить содержимое записи. (Вы можете найти этот пример в *ch10/ex4-ets*.)

Пример 10-5. Настройка простой таблицы ETS и отчетности

```
-module(planemo_storage).
-export([setup/0]).
-include("records.hrl").

setup() ->
       PlanemoTable=ets:new(planemos, [named_table, {keypos, #planemo.name}]),
       ets:info(PlanemoTable).
```

Если вы скомпилируете и запустите код, вы получите отчет о пустой таблице ETS с большим количеством свойств, о которых вы, вероятно, пока не интересно знать в данный момент.

```
1> c(planemo_storage).
{ok,planemo_storage}
2> planemo_storage:setup().
[{compressed,false},
{memory,317},
{owner,<0.316.0>},
{heir,none},
{name,planemos},
{size,0},
{node,nonode@nohost},
{named_table,true},
{type,set},
{keypos,2},
{protection,protected}]
```

В большинстве случаев вы получите множество служебной инфомации. Наиболее важные данные это `name` (planemos), `size` (0 - пусто!) и `keypos` (не 1 (по умолчанию), а 2, расположение имени в кортеже в записи). Как указано по умолчанию, доступ к таблице настроен, как `protected`. (`nonode@nohost` просто относится к текущей среде Erlang, когда вы не распределяете обработку по нескольким системам. Если вы используете распределенную систему Erlang, у вас будет несколько узлов, каждый из которых имеет свою собственную независимую среду выполнения Erlang со своим собственным именем.

Вы можете создать только одну таблицу ETS с тем же именем. Если вы дважды вызовете функцию `planemo_storage:setup/0`, вы получите сообщение об ошибке:

```
3> planemo_storage:setup().
** exception error: bad argument
     in function ets:new/2
            called as ets:new(planemos,[named_table,{keypos,2}])
            in call from planemo_storage:setup/0 (planemo_storage.erl, line 6)
```

Чтобы избежать этого, по крайней мере в этих ранних тестах, весьма удобно использовать команду оболочки f() для очистки уже созданные таблицы. Также можно проверить существование таблицы с помощью метода `ets:info/1`. Если функция вернёт undefined – такой таблицы ещё нет. Также можно обернуть создание таблицы в конструкцию `try…catch` вокруг вызова функции `ets:new/2`.

Разумеется, более полезная таблица ETS будет включать контент. Следующим шагом будет использование функции `ets:insert/2` для добавления содержимого в таблицу. Первый аргумент – это таблица, на которую можно сослаться по имени (если вы установили опцию `named_table`), либо переменну, которая содержит возвращаемое значение функции `ets:new/2`. В примере 10-6, который находится в *ch10/ex5-ets*, первый вызов использует имя, чтобы показать её можно получить, а остальные используют переменную. Второй аргумент – это запись, представляющая одну из строк таблицы 10-1.

Пример 10-6. Заполнение простой таблицы ETS и составление отчетов о том, что там

```
-module(planemo_storage).
-export([setup/0]).
-include("records.hrl").

setup() ->
PlanemoTable=ets:new(planemos, [named_table, {keypos, #planemo.name}]),
ets:insert(planemos,
#planemo{ name=mercury, gravity=3.7, diameter=4878, distance_from_sun=57.9 }),
ets:insert(PlanemoTable,
#planemo{ name=venus, gravity=8.9, diameter=12104, distance_from_sun=108.2 }),
ets:insert(PlanemoTable,
#planemo{ name=earth, gravity=9.8, diameter=12756, distance_from_sun=149.6 }),
ets:insert(PlanemoTable,
#planemo{ name=moon, gravity=1.6, diameter=3475, distance_from_sun=149.6 }),
ets:insert(PlanemoTable,
#planemo{ name=mars, gravity=3.7, diameter=6787, distance_from_sun=227.9 }),
ets:insert(PlanemoTable,
#planemo{ name=ceres, gravity=0.27, diameter=950, distance_from_sun=413.7 }),
ets:insert(PlanemoTable,
#planemo{ name=jupiter, gravity=23.1, diameter=142796, distance_from_sun=778.3 }),
ets:insert(PlanemoTable,
#planemo{ name=saturn, gravity=9.0, diameter=120660, distance_from_sun=1427.0 }),
ets:insert(PlanemoTable,
#planemo{ name=uranus, gravity=8.7, diameter=51118, distance_from_sun=2871.0 }),
ets:insert(PlanemoTable,
#planemo{ name=neptune, gravity=11.0, diameter=30200, distance_from_sun=4497.1 }),
ets:insert(PlanemoTable,
#planemo{ name=pluto, gravity=0.6, diameter=2300, distance_from_sun=5913.0 }),
ets:insert(PlanemoTable,
#planemo{ name=haumea, gravity=0.44, diameter=1150, distance_from_sun=6484.0 }),
ets:insert(PlanemoTable,
#planemo{ name=makemake, gravity=0.5, diameter=1500, distance_from_sun=6850.0 }),
ets:insert(PlanemoTable,
```

```
#planemo{ name=eris, gravity=0.8, diameter=2400, distance_from_sun=10210.0 }),
ets:info(PlanemoTable).
```

Обритите внимание на последнее выражение `ets:info/1`, которое теперь сообщит, что таблица содержит 14 элементов.

```
4> c(planemo_storage).
{ok,planemo_storage}
5> f().
ok
6> planemo_storage:setup().
[{compressed,false},
{memory,541},
{owner,<0.342.0>},
{heir,none},
{name,planemos},
{size,14},
{node,nonode@nohost},
{named_table,true},
{type,set},
{keypos,2},
{protection,protected}]
```

Если вы хотите увидеть, что находится в этой таблице, у вас есть несколько вариантов. Быстрый способ сделать это в оболочке – использовать функцию `ets:tab2list/1`, которая будет возвращать список записей (или кортежей, если вы пропустите импорт записей в строке 7):

```
7> rr("records.hrl").
[planemo,tower]
8> ets:tab2list(planemos).
[#planemo{name = pluto,gravity = 0.6,diameter = 2300,
distance_from_sun = 5913.0},
#planemo{name = saturn,gravity = 9.0,diameter = 120660,
distance_from_sun = 1427.0},
#planemo{name = moon,gravity = 1.6,diameter = 3475,
distance_from_sun = 149.6},
#planemo{name = mercury,gravity = 3.7,diameter = 4878,
distance_from_sun = 57.9},
#planemo{name = earth,gravity = 9.8,diameter = 12756,
distance_from_sun = 149.6},
#planemo{name = neptune,gravity = 11.0,diameter = 30200,
distance_from_sun = 4497.1},
#planemo{name = makemake,gravity = 0.5,diameter = 1500,
distance_from_sun = 6850.0},
#planemo{name = uranus,gravity = 8.7,diameter = 51118,
distance_from_sun = 2871.0},
#planemo{name = ceres,gravity = 0.27,diameter = 950,
distance_from_sun = 413.7},
#planemo{name = venus,gravity = 8.9,diameter = 12104,
distance_from_sun = 108.2},
#planemo{name = mars,gravity = 3.7,diameter = 6787,
distance_from_sun = 227.9},
#planemo{name = eris,gravity = 0.8,diameter = 2400,
distance_from_sun = 10210.0},
#planemo{name = jupiter,gravity = 23.1,diameter = 142796,
distance_from_sun = 778.3},
#planemo{name = haumea,gravity = 0.44,diameter = 1150,
distance_from_sun = 6484.0}]
```

Если вы предпочитаете отслеживать таблицу в отдельном окне, визуализатор таблиц Erlang отобразит ту же информацию в несколько более удобочитаемой форме. Вы можете запустить его из оболочки с помощью `observer`:start(), а затем нажать на вкладку «Table Viewer». Вы увидите что-то вроде таблицы на рисунке 10-1. Дважды щелкните таблицу planmos, и появится более подробный отчет о ее содержимом, подобный показанному на рис. 10-2.

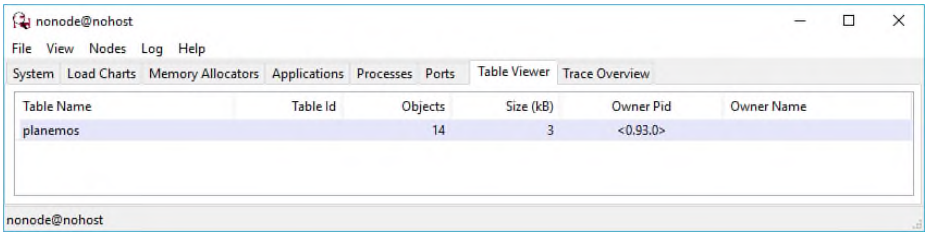


Рисунок 10-1. Открытие таблицы в визуализаторе

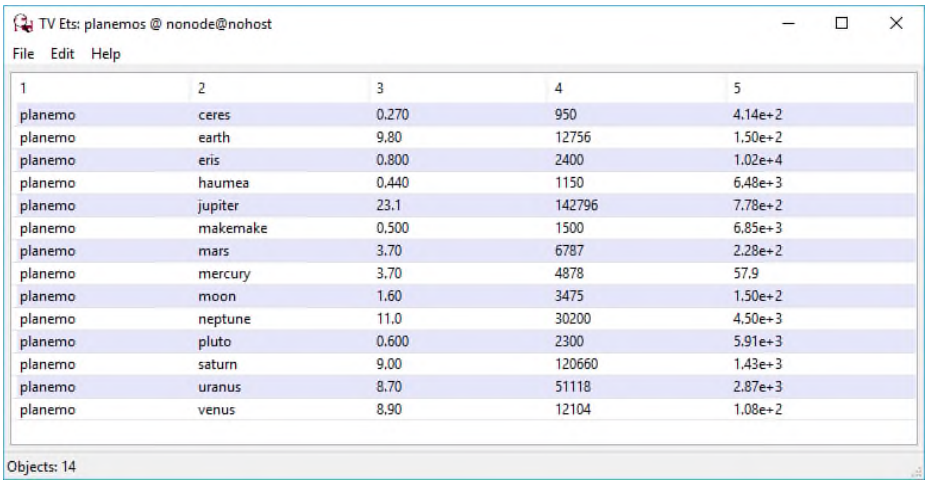


Рисунок 10-2. Просмотр таблицы planmos в визуализаторе

У Визуализатора нет данных о ваших записях. Меню «Edit» подменю «Refresh interval» позволяет вам установить интервал обновления данных таблицы, если вы хотите, чтобы она обновлялась автоматически. Если вы объявляете таблицы общедоступными, вы даже можете редактировать их содержимое в визуализаторе.

Если вы захотите ознакомится с описанием загруженных в память таблиц ETS, используйте функцию `ets:i()` в оболочке. Вы получите список таблиц и их описание в виде отчёта.

Простые Запросы

Самый простой способ поиска записей в вашей таблице ETS – это функция ets:lookup/2 и ключ. Вы можете легко проверить это в оболочке:

```
9> ets:lookup(planemos,eris).
[#planemo{name = eris,gravity = 0.8,diameter = 2400,distance_from_sun = 10210.0}]
```

Возвращаемое значение всегда является списком. Это верно, несмотря на то, что Erlang знает, что эта таблица ETS имеет тип `set`, поэтому только одно значение может соответствовать ключу. В таких ситуациях, когда вы знаете, что будет только одно возвращаемое значение, функция `hd/1`, показанная в Примере 5-5 для использования с пользовательскими данными, может быстро привести вас к началу списка. Так как есть только одино значение, первый элемент в списке будет этим самым значением.

```
10> hd(ets:lookup(planemos,eris)).
#planemo{name = eris,gravity = 0.8,diameter = 2400,distance_from_sun = 10210.0}
```

Обритите внимание, что возвращаемое значение не имеет квадратных скобок и это значит, что теперь вы можете извлечь, скажем, значение гравитаци:

```
11> Result=hd(ets:lookup(planemos,eris)).
#planemo{name = eris,gravity = 0.8,diameter = 2400,distance_from_sun = 10210.0}
12> Result#planemo.gravity.
0.8
```

Вы также можете использовать сопоставление с образцом для извлечения значения вместо функции hd/1. Например, `[Result] = ets:lookup(planemos, eris)`. Оба подхода потерпят неудачу, если возвращаемое значение будет пустым списком.

Ключевая особенность таблиц: перезапись значений

До сих пор вам приходилось работать (или обходиться с) парадигмой единственного присваивания в Erlang: вы не можете перезаписать значение переменной или изменить значение элемента в списке напрямую. Однако ETS не имеет такого ограничения. Если вы хотите изменить значение силы тяжести на меркурии, вы можете:

```
13> ets:insert(planemos, #planemo{ name=mercury, gravity=3.9, diameter=4878,
distance_from_sun=57.9 }).
true
14> ets:lookup(planemos, mercury).
[#planemo{name = mercury,gravity = 3.9,diameter = 4878,
distance_from_sun = 57.9}]
```

Однако то, что вы *можете* изменять значения в таблице ETS, не означает, что вы должны переписать свой код, чтобы заменить неизменяемые переменные гибким содержимым таблицы ETS. Не следует также делать все свои таблицы общедоступными, чтобы различные процессы могли читать и записывать все, что им нравится, в таблицу ETS, что делает ее другой формой разделяемой памяти.

Постарайтесь вспомнить всё, что вам уже пришлось выучить до этого момента. Спросите себя, когда внесение изменений будет полезно, и когда это может привести к сложным ошибкам. Вероятно, вам не придется менять гравитацию Меркурия, но, безусловно, имеет смысл изменить адрес доставки. Если у вас есть сомнения, будьте осторожны.

ETS Таблицы и процессы

Теперь, когда вы можете извлекать гравитационные константы для небесных тел, вы можете расширить модуль `drop`, чтобы вычислять скорость падения для разных небесных тел. Пример 10-7 объединяет модуль drop из примера 8-6 с таблицей ETS, построенной в примере 10-6, для создания более мощного калькулятора скоростей падения. (Вы можете найти его в *ch10/ex6-ets-calculator*.)

Пример 10-7. Вычисление скоростей падения с помощью свойств таблицы ETS planeto

```
-module(drop).
-export([drop/0]).
-include("records.hrl").

drop() ->
        setup(),
        handle_drops().
```

```
handle_drops() ->
        receive
                {From, Planemo, Distance} ->
                        From ! {Planemo, Distance, fall_velocity(Planemo, Distance)},
                handle_drops()
end.

fall_velocity(Planemo, Distance) when Distance >= 0 ->
        P=hd(ets:lookup(planemos,Planemo)),
        math:sqrt(2 * P#planemo.gravity * Distance).

setup() ->
        ets:new(planemos, [named_table, {keypos, #planemo.name}]),
        ets:insert(planemos,
        #planemo{ name=mercury, gravity=3.7, diameter=4878, distance_from_sun=57.9 }),
        ets:insert(planemos,
#planemo{ name=venus, gravity=8.9, diameter=12104, distance_from_sun=108.2 }),
        ets:insert(planemos,
#planemo{ name=earth, gravity=9.8, diameter=12756, distance_from_sun=149.6 }),
        ets:insert(planemos,
#planemo{ name=moon, gravity=1.6, diameter=3475, distance_from_sun=149.6 }),
        ets:insert(planemos,
#planemo{ name=mars, gravity=3.7, diameter=6787, distance_from_sun=227.9 }),
        ets:insert(planemos,
#planemo{ name=ceres, gravity=0.27, diameter=950, distance_from_sun=413.7 }),
        ets:insert(planemos,
#planemo{ name=jupiter, gravity=23.1, diameter=142796, distance_from_sun=778.3 }),
        ets:insert(planemos,
#planemo{ name=saturn, gravity=9.0, diameter=120660, distance_from_sun=1427.0 }),
        ets:insert(planemos,
#planemo{ name=uranus, gravity=8.7, diameter=51118, distance_from_sun=2871.0 }),
        ets:insert(planemos,
#planemo{ name=neptune, gravity=11.0, diameter=30200, distance_from_sun=4497.1 }),
        ets:insert(planemos,
#planemo{ name=pluto, gravity=0.6, diameter=2300, distance_from_sun=5913.0 }),
        ets:insert(planemos,
#planemo{ name=haumea, gravity=0.44, diameter=1150, distance_from_sun=6484.0 }),
        ets:insert(planemos,
#planemo{ name=makemake, gravity=0.5, diameter=1500, distance_from_sun=6850.0 }),
        ets:insert(planemos,
#planemo{ name=eris, gravity=0.8, diameter=2400, distance_from_sun=10210.0 }).
```

Функция `drop/0` немного меняется, чтобы вызывать инициализацию отдельно и избегать настройки таблицы при каждом вызове. Это перемещает обработку сообщений в отдельную функцию `handle_drop/0`. Функция `fall_velocity/2` также изменяется, поскольку теперь она ищет имена `planemo` в таблице ETS и получает их гравитационную постоянную из этой таблицы, а не жестко кодирует это содержимое в функцию. (Хотя, безусловно, было бы возможно передать переменную `PlanemoTable` из предыдущего примера в качестве аргумента обработчику рекурсивных сообщений, проще использовать её как именованную таблицу.)

Если этот процесс дает сбой и его необходимо перезапустить, его перезапуск вызовет функцию `setup/0`, которая в настоящее время не проверяет, существует ли таблица ETS. Это может вызвать ошибку, если таблица всё ещё существует. ETS предлагает опцию `heir` и функцию `ets:give_away/3`, если вы хотите избежать такого поведения, но пока и это работает хорошо.

Если вы объедините этот модуль с модулем mph_drop из Примера 8-7, вы сможете вычислить скорости падения на всех этих небесных телах:

```
1> c(drop).
{ok,drop}
2> c(mph_drop).
{ok,mph_drop}
3> Pid1=spawn(mph_drop,mph_drop,[]).
<0.33.0>
4> Pid1 ! {earth,20}.
On earth, a fall of 20 meters yields a velocity of 44.289078952755766 mph.
{earth,20}
```

```
5> Pid1 ! {eris,20}.
On eris, a fall of 20 meters yields a velocity of 12.65402255793022 mph.
{eris,20}
6> Pid1 ! {makemake,20}.
On makemake, a fall of 20 meters yields a velocity of 10.003883211552367 mph.
{makemake,20}
```

Эти примены разнообразнее предыдущих (так как у нас были только Земля, Луна и Марс)!

Следующие шаги

В то время как многим приложениям просто нужно быстрое хранилище ключей/значений, таблицы ETS гораздо более гибкие, чем примеры, которые до сих пор демонстрировали. Вы можете использовать спецификации соответствия Erlang и `ets:fun2ms` для создания более сложных запросов с помощью `ets:match` и `ets:select`. Вы можете удалить строки (и таблицы) с помощью `ets:delete`. Функции `ets:first`, `ets:next` и `ets:last` позволяют рекурсивно просматривать таблицы.

Возможно, самое главное, вы также можете изучить DETS, дисковое хранилище термов, которое предлагает аналогичные функции, но с таблицами, хранящимися на диске. Это решение медленнее, с ограничением в 2 ГБ, но данные не исчезают, когда процесс управления останавливается.

Вы можете углубиться в ETS и DETS, но если ваши потребности более сложны, особенно если вам нужно разделить данные по нескольким узлам, вам, вероятно, следует изучить базу данных Mnesia.

ETS и DETS обсуждаются в главе 10 «Erlang Programming»; Глава 19 «Programming Erlang», 2-е издание; Раздел 2.14 и Глава 6 «Erlang and OTP in Action»; и 25-я глава «Learn You Some Erlang For Great Good!»

Хранение записей в СУБД Mnesia

Mnesia - это система управления базами данных (СУБД), которая поставляется с Erlang. Он использует ETS и DETS, но предоставляет гораздо больше возможностей, чем эти компоненты.

Вы должны рассмотреть возможность перехода от таблиц ETS (и DETS) к базе данных Mnesia, если:

- Вам необходимо хранить и получать доступ к данным через набор узлов, а не только на одном узеле.
- Вы не хотите иметь прозрачным хранилище данных (не важно хранятся ли данные в памяти или на диске (или в обоих случаях)).
- Вы должны иметь возможность откатить транзакции, если что-то пойдет не так.
- Вам нужен более простой синтаксис для поиска и объединения данных. Менеджмент предпочитает звучание «базы данных» звучанию «таблицы».
- Вы можете даже использовать ETS для некоторых аспектов проекта и Mnesia для других.

Это не «амнезия» – забывчивость, а «мнезия» – греческое слово «память».

Запуск Mnesia

Если вы хотите хранить данные на диске, то вам нужно сообщить Mnesia некоторую дополнительную информацию. Прежде чем запустить Mnesia, вам нужно создать базу данных, используя функцию mnesia:create_schema/1. На данный момент, поскольку вы начнете использовать только локальный узел, это будет выглядеть следующим образом:

```
1> mnesia:create_schema([node()]).
ok
```

По умолчанию, когда вы вызываете `mnesia:create_schema/1`, Mnesia будет сохранять данные схемы в каталоге, в котором вы находитесь, когда вы его запускаете. Если вы посмотрите в каталог, где вы запустили Erlang, вы увидите новый каталог с именем, таким как *Mnesia.nonode@nohost*. Первоначально он содержит файл *LATEST.LOG* и файл *schema.DAT*. Функция `node()` просто возвращает идентификатор узла, на котором вы находитесь, что хорошо, когда вы начинаете работу. (Если вы хотите изменить место хранения данных в Mnesia, вы можете запустить Erlang с некоторыми дополнительными опциями: `erl -mnesia dir "path"`. Путь будет местом, где Mnesia хранит любое дисковое хранилище.)

Если вы запустите Mnesia без вызова `mnesia:create_schema/1`, она сохранит свою схему в памяти, и эта схема исчезнет, если и когда Mnesia остановится.

В отличие от ETS и DETS, которые всегда доступны, вам нужно включить Mnesia:

```
2> mnesia:start().
ok
```

Также есть функция `mnesia:stop/0`, если вы хотите остановить ее.

Если вы запускаете Mnesia на компьютере, который переходит в спящий режим, при пробуждении вы можете получить странные сообщения, такие как `Mnesia(nonode@nohost): ** WARNING ** Mnesia is overloaded: {dump_log, time_threshold}`. Не беспокойтесь. Это побочный эффект от пробуждения, и ваши данные должны быть в безопасности. Конечно, лучше не запускать производственные системы на устройствах, которые переходят в спящий режим.

Создание таблиц

Как и ETS, базовая концепция Mnesia – это набор записей. Он также предлагает параметры set, order_set и bag, как и в ETS, но не предлагает duplicate_bag.

Mnesia также хочет знать больше о ваших данных, чем ETS. ETS в значительной степени принимает данные в кортежах любой формы, рассчитывая только на наличие ключа, который он может использовать. Остальное зависит от вас, чтобы интерпретировать. Mnesia хочет узнать больше о том, что вы храните, и берет список имен полей. Простой способ справиться с этим – определить записи и последовательно использовать имена полей из записей в качестве имен полей Mnesia. Есть даже простой способ передать имена записей в Mnesia, используя `record_info/2`.

Таблица сожержащая информацию о планетах в Mnesia может работать так же легко, как и в ETS, и некоторые аспекты работы с ним будут проще. В примере 10-8, описанном в разделе *ch10/ex7-mnesia*, показано, как настроить таблицу `planemo` в Mnesia. Метод `setup/0` создает схему, затем запускает Mnesia, а затем создает таблицу на основе `planemo` записи. После создания таблицы в нее записываются значения из таблицы 10-1.

Пример 10-8. Настройка таблицы свойств planemo в Mnesia

```
-module(drop).
-export([setup/0]).
-include("records.hrl").

setup() ->
mnesia:create_schema([node()]),
mnesia:start(),
mnesia:create_table(planemo, [{attributes, record_info(fields, planemo)}]),

F = fun() ->
mnesia:write(
#planemo{ name=mercury, gravity=3.7, diameter=4878, distance_from_sun=57.9 }),
mnesia:write(
#planemo{ name=venus, gravity=8.9, diameter=12104, distance_from_sun=108.2 }),
mnesia:write(
#planemo{ name=earth, gravity=9.8, diameter=12756, distance_from_sun=149.6 }),
mnesia:write(
#planemo{ name=moon, gravity=1.6, diameter=3475, distance_from_sun=149.6 }),
mnesia:write(
#planemo{ name=mars, gravity=3.7, diameter=6787, distance_from_sun=227.9 }),
mnesia:write(
#planemo{ name=ceres, gravity=0.27, diameter=950, distance_from_sun=413.7 }),
mnesia:write(
#planemo{ name=jupiter, gravity=23.1, diameter=142796, distance_from_sun=778.3 }),
mnesia:write(
#planemo{ name=saturn, gravity=9.0, diameter=120660, distance_from_sun=1427.0 }),
mnesia:write(
#planemo{ name=uranus, gravity=8.7, diameter=51118, distance_from_sun=2871.0 }),
mnesia:write(
#planemo{ name=neptune, gravity=11.0, diameter=30200, distance_from_sun=4497.1 }),
mnesia:write(
#planemo{ name=pluto, gravity=0.6, diameter=2300, distance_from_sun=5913.0 }),
mnesia:write(
#planemo{ name=haumea, gravity=0.44, diameter=1150, distance_from_sun=6484.0 }),
mnesia:write(
#planemo{ name=makemake, gravity=0.5, diameter=1500, distance_from_sun=6850.0 }),
mnesia:write(
#planemo{ name=eris, gravity=0.8, diameter=2400, distance_from_sun=10210.0 })
end,
mnesia:transaction(F).
```

Помимо настройки, важно отметить, что все записи содержатся в fun, которая затем передается в функцию `mnesia:transaction(F)`, которая будет выполнена как транзакция. Mnesia перезапустит транзакцию, если есть другие действия, блокирующие ее, поэтому код может выполняться повторно до того, как транзакция произойдет. Из-за этого не включайте никакие вызовы, которые создают побочные эффекты для функции, которую вы будете передавать в `mnesia:action`, и не пытайтесь перехватывать исключения для функций Mnesia в транзакции. Если ваша функция вызывает `mnesia:abort/1` (возможно, из-за того, что какое-то условие для ее выполнения не было выполнено), транзакция будет откатываться, возвращая кортеж, начинающийся с `aborted` вместо `atomic`.

Вы также можете изучить более гибкую функцию `mnesia:activity/2`, когда вам нужно смешать больше видов задач в транзакции.

Ваше взаимодействие с Mnesia должно использовать транзакции, особенно когда база данных используется несколькими узлами. Основные функции `mnesia:write`, `mnesia:read` и `mnesia:delete` работают только внутри транзакций. Существуют, конечно, небезопасные методы, но каждый раз, когда вы их используете, особенно для записи в базу данных, вы очень рискуете потерять данные или испортить базу данных.

Как и в ETS, вы можете перезаписывать значения, записывая новое значение с тем же ключом, что и предыдущая запись.

Если вы хотите проверить, как работает эта функция, попробуйте функцию `mnesia:table_info`, которая может рассказать вам даже больше, чем вы хотите знать. Следующий список сокращен, чтобы сосредоточиться на ключевых результатах.

```
1> c(drop).
{ok,drop}
2> rr("records.hrl").
[planemo,tower]
3> drop:setup().
{atomic,ok}
4> mnesia:table_info(planemo,all).
[{access_mode,read_write},
{active_replicas,[nonode@nohost]},
{all_nodes,[nonode@nohost]},
{arity,5},
{attributes,[name,gravity,diameter,distance_from_sun]},
...
{memory,541},
{ram_copies,[nonode@nohost]},
{record_name,planemo},
{record_validation,{planemo,5,set}},
{type,set},
{size,14},
...]
```

Вы можете видеть, какие узлы участвуют в таблице (nonode@nohost является значением по умолчанию для текущего узла). В этом случае `arity` – это количество полей в записи, а атрибуты сообщают вам, каковы их имена. `ram_copies` плюс имя текущего узла говорит вам, что эта таблица хранится в памяти локально. Здесь всё также, как в примере ETS - таблица имеет типа set и содержит 14 записей.

По умолчанию Mnesia будет хранить вашу таблицу только в оперативной памяти (`ram_copies`) на текущем узле. Это быстро, но это означает, что данные исчезают в случае сбоя узла. Если вы укажете `disc_copies` (обратите внимание на написание), Mnesia сохранит копию базы данных на диске, но все равно будет использовать оперативную память для обеспечения высокой скорости. Вы также можете указать `disc_only_copies`, что замедлит работу Mnesia. В отличие от ETS, созданная вами таблица будет по-прежнему рядом, если процесс, в котором она была создана, аварийно завершится и, скорее всего, выживет даже при сбое узла, если только она не находится в ОЗУ на

одном узле. Комбинируя эти опции и (в конечном итоге) несколько узлов, вы сможете создавать быстрые и отказоустойчивые системы.

Теперь таблица настроена, и вы можете начать ее использовать. Если вы запускаете средство просмотра таблиц или запускаете его с помощью `observer:start()`, вы можете просмотреть содержимое таблиц Mnesia, а также таблиц ETS. В меню «View» выберите «Mnesia Tables». Интерфейс похож на интерфейс для таблиц ETS.

Чтение данных

Точно так же, как для записил, вы должны обернуть вызов функции `mnesia:read` в свою функцию, которую вы затем передадите функции `mnesia:transaction`. Вы можете сделать это в оболочке:

```
5> mnesia:transaction(fun() -> mnesia:read(planemo,neptune) end).
{atomic,[#planemo{name = neptune,gravity = 11.0,
diameter = 30300,distance_from_sun = 4497.1}]}
```

Результат поступает в виде кортежа, который в случае успеха содержит атом `atomic` плюс список с данными таблицы. Данные таблицы упакованы в виде записи, и вы можете легко получить доступ к её полям.

Вы можете переписать функцию `fall_velocity/2` из примера 10-8, чтобы использовать транзакцию Mnesia вместо вызова ETS. Версия ETS выглядела следующим образом:

```
fall_velocity(Planemo, Distance) when Distance >= 0 ->
       P=hd(ets:lookup(planemos,Planemo)),
       math:sqrt(2 * P#planemo.gravity * Distance).
```

Строка 2 версии Mnesia немного отличается:

```
fall_velocity(Planemo, Distance) when Distance >= 0 ->
       {atomic, [P | _]} = mnesia:transaction(fun() -> mnesia:read(planemo,Planemo) end),
       math:sqrt(2 * P#planemo.gravity * Distance).
```

Поскольку Mnesia возвращает кортеж, а не список, этот код использует сопоставление с образцом для извлечения первого элемента в списке, содержащемся во втором элементе кортежа (и отбрасывает хвост этого списка с помощью _). Эта таблица является набором, поэтому там всегда будет только один элемент. Тогда данные, содержащиеся в P, можно использовать для того же расчета, что и раньше.

Если вы скомпилируете и запустите этот код, вы увидите знакомый результат:

```
6> c(drop).
{ok,drop}
7> drop:fall_velocity(earth,20).
19.79898987322333
8> Pid1=spawn(mph_drop,mph_drop,[]).
<0.120.0>
9> Pid1 ! {earth,20}.
{earth,20}
On earth, a fall of 20 meters yields a velocity of 44.289078952755766 mph.
```

Для этих целей использование mnesia:read достаточно. Вы можете указать Mnesia создать индексы для полей, отличных от ключа, и запросить их с помощью `mnesia:index_read`.

Если вы хотите удалить записи, вы можете запустить mnesia:delete/2, также внутри транзакции.

Язык запросов

Если Mnesia – это действительно СУБД, то она должна уметь делать больше, чем просто запрос значения ключа, не так ли? Она, определенно, может. Вы можете использовать спецификации соответствия Erlang (также как в ETS), но запросы в виде генераторов списков (Query List Comprehensions – QLC) гораздо более читабельны. Они выглядят как списки, которые вы видели в главе 7, но работают с таблицами Mnesia, а не со списками.

Предположим, вы хотите найти все небесные тела с гравитацией меньше, чем у Земли. Вы можете проанализировать всю таблицу с помощью функций `mnesia:first` и `mnesia:next`, но есть более простой способ. Вместо этого вы можете использовать функцию `qlc:q` для хранения списка и функцию `qlc:e` (или эквивалентную, но более длинную `qlc:eval`) для ее обработки. Затем вы запускаете это внутри функции mnesia:transaction.

Вы можете запустить QLC в оболочке, но если вы хотите использовать их в модулях, вам нужно добавить декларациюю -include_lib ("stdlib/include/qlc.hrl"). в верхней части вашего модуля.

Простейшее QLC просто возвращает все значения в таблице. Чтобы текст был более понятным, я разбил его на отдельные строки, чтобы вы могли видеть, как они взаимодействуют:

```
mnesia:transaction(
        fun() ->
                qlc:e(
                qlc:q( [X || X <- mnesia:table(planemo)] )
                )
        end
)
```

Функция mnesia:transaction принимает в качестве аргумента анонимную функцию. В этом случае она содержит вызов функции `qlc:e`, которая содержит вызов функции `qlc:q`, в котром находится текст запроса. Запрос создаст список из содержимого таблицы `planemo`.

Если вы запишите этот вызов функции mnesia:transaction и запустите в оболочке, вы увидите, что результирующий список, заключенный в кортеж результата транзакции, содержит всю таблицу.

```
10> mnesia:transaction( fun() -> qlc:e(qlc:q([X || X <- mnesia:table(planemo)]))
end).
{atomic,[#planemo{name = pluto,gravity = 0.6,
diameter = 2300,distance_from_sun = 5913.0},
#planemo{name = saturn,gravity = 9.0,diameter = 120660,
distance_from_sun = 1427.0},
#planemo{name = moon,gravity = 1.6,diameter = 3475,
distance_from_sun = 149.6},
#planemo{name = mercury,gravity = 3.7,diameter = 4878,
distance_from_sun = 57.9},
#planemo{name = earth,gravity = 9.8,diameter = 12756,
```

```
distance_from_sun = 149.6},
#planemo{name = neptune,gravity = 11.0,diameter = 30200,
distance_from_sun = 4497.1},
#planemo{name = makemake,gravity = 0.5,diameter = 1500,
distance_from_sun = 6850.0},
#planemo{name = uranus,gravity = 8.7,diameter = 51118,
distance_from_sun = 2871.0},
#planemo{name = ceres,gravity = 0.27,diameter = 950,
distance_from_sun = 413.7},
#planemo{name = venus,gravity = 8.9,diameter = 12104,
distance_from_sun = 108.2},
#planemo{name = mars,gravity = 3.7,diameter = 6787,
distance_from_sun = 227.9},
#planemo{name = eris,gravity = 0.8,diameter = 2400,
distance_from_sun = 10210.0},
#planemo{name = jupiter,gravity = 23.1,diameter = 142796,
distance_from_sun = 778.3},
#planemo{name = haumea,gravity = 0.44,diameter = 1150,
distance_from_sun = 6484.0}]}
```

Вы можете добавить условия в QLC. Чтобы найти все небесные тела гравитацией меньше, чем у Земли 9.8, вы должны добавить условие в запрос:

```
mnesia:transaction(
        fun() ->
                qlc:e(
                qlc:q( [X || X <- mnesia:table(planemo),
                X#planemo.gravity < 9.8] )
                )
        end
)
```

Пожалуйста, введите это в оболочке в одну строку. Вы получите более короткий список планет, где гравитация меньше, чем у Земли.

```
11> mnesia:transaction( fun() -> qlc:e(qlc:q( [X || X <- mnesia:table(planemo),
X#planemo.gravity < 9.8] )) end).
{atomic,[#planemo{name = pluto,gravity = 0.6,
diameter = 2300,distance_from_sun = 5913.0},
#planemo{name = saturn,gravity = 9.0,diameter = 120660,
distance_from_sun = 1427.0},
#planemo{name = moon,gravity = 1.6,diameter = 3475,
distance_from_sun = 149.6},
#planemo{name = mercury,gravity = 3.7,diameter = 4878,
distance_from_sun = 57.9},
#planemo{name = makemake,gravity = 0.5,diameter = 1500,
distance_from_sun = 6850.0},
#planemo{name = uranus,gravity = 8.7,diameter = 51118,
distance_from_sun = 2871.0},
#planemo{name = ceres,gravity = 0.27,diameter = 950,
distance_from_sun = 413.7},
#planemo{name = venus,gravity = 8.9,diameter = 12104,
distance_from_sun = 108.2},
#planemo{name = mars,gravity = 3.7,diameter = 6787,
distance_from_sun = 227.9},
#planemo{name = eris,gravity = 0.8,diameter = 2400,
distance_from_sun = 10210.0},
#planemo{name = haumea,gravity = 0.44,diameter = 1150,
distance_from_sun = 6484.0}]}
```

Этот вывод по-прежнему содержит больше информации, чем это может быть необходимо. Вы можете изменить левую сторону запроса, чтобы сократить количество выводимой информации, создавая кортеж, который содержит данные свойств `name` и `gravity` небесных тел:

```
mnesia:transaction(
        fun() ->
                qlc:e(
                qlc:q( [{X#planemo.name, X#planemo.gravity} ||
                X <- mnesia:table(planemo),
                X#planemo.gravity < 9.8] )
                )
        end
)
```

Результат будет намного короче:

```
12> mnesia:transaction( fun() -> qlc:e(qlc:q( [ {X#planemo.name, X#planemo.gravity}||
X <- mnesia:table(planemo), X#planemo.gravity < 9.8] )) end).
{atomic,[{pluto,0.6},
{saturn,9.0},
{moon,1.6},
{mercury,3.7},
{makemake,0.5},
{uranus,8.7},
{ceres,0.27},
{venus,8.9},
{mars,3.7},
{eris,0.8},
{haumea,0.44}]}
```

Существуют способы уменьшить количество кода, которое необходимо писать для реализации подобных выражений. Например, нетрудно переместить определение mnesia:transaction, анонимную функцию и `qlc:e` в отдельную функцию, которая принимает функцию `qlc:q` в качестве аргумента. В книге «Programming Erlang» Джо Армстронг делает именно это, чтобы создать функцию `do`. Вам решать, как лучше упростить код. Это зависит от вашего стиля кодирования и структур данных.

Вы можете использовать QLC для более чем одной таблицы одновременно, что позволяет создавать эквивалент соединений между таблицами. Это также возможно при работе с таблицами ETS.

Это всего лишь краткое введение в СУБД Mnesia. Он получил некоторое освещение во всех книгах об Erlang. Я все-таки надеюсь, что в конце концов, о ней будет написана своя книга, примерно такую же, как и эта.

Mnesia описана в главе 13 «Erlang Programming» (O'Reilly); Глава 20 «Programming Erlang», 2-е издание (Pragmatic); Раздел 2.7 «Erlang and OTP in Action» (Manning); и Глава 29 «Learn You Some Erlang For Great Good!» (No Starch Press).

Глава 11. Начало работы с OTP

На данный момент может показаться, что у вас есть все, что вам нужно для создания ориентированных на процессы проектов на Erlang. Вы знаете, как создавать полезные функции, умеете работать с рекурсией, знаете структуры данных, которые предлагает Erlang, и, возможно, самое главное, знаете, как создавать процессы и управлять ими. Что еще может понадобиться?

Процессно-ориентированное программирование – это здорово, но детали реализации имеют важное значение. Базовые инструменты Erlang являются мощными, но они также могут привести вас к расстраивающим лабиринтам и гонкам при отладке, которые случаются только время от времени. Смешивание различных стилей программирования может привести к несовместимым ожиданиям, и код, который хорошо работает в одной среде, может оказаться сложнее интегрировать в другую.

Происхождение OTP

Ericsson столкнулся с этими проблемами давольно таки рано и создал библиотеку кода, которая облегчает их решение. OTP, изначально Open Telecom Platform, полезен практически для любого крупномасштабного проекта, который вы хотите сделать с Erlang, а не только для телекоммуникационной работы. Он включен в Erlang, и хотя он не является частью языка, он определенно является частью среды Erlang и помогает определить культуру программирования Erlang. Границы того, где заканчивается Erlang и начинается OTP, не всегда ясны, но отправной точкой, безусловно, является поведение (behavior). С помощью OTP ваши приложения будут объединять процессы, построенные с использованием поведений и управляемые супервизорами.

До сих пор жизненный цикл процессов, показанных в предыдущих главах, был довольно простым. При необходимости они настраивают другие ресурсы или процессы. После запуска они слушают сообщения и обрабатывают их, а также завершаются при сбое. Некоторые из них могут перезапустить сбойный процесс, если это необходимо.

OTP формализует эти действия, и еще дополнительные, в набор поведений. Наиболее распространенными видами поведения являются `gen_server` (универсальный сервер) и `supervisor`. Также доступны `gen_statem` (автомат состояния), `gen_fsm` (конечный автомат) и `gen_event`. Поведение приложения позволяет вам упаковать ваш OTP-код в единую работоспособную (и обновляемую) систему.

Поведение предопределяет механизмы, которые ваш код будет использовать для создания и взаимодействия с процессами, и компилятор предупредит вас, если вы пропустите некоторые из них. Ваш код должен обрабатывать обратные вызовы, определяя, как реагировать на определенные виды событий. OTP предлагает вам несколько вариантов структурирования вашего приложения.

Если вы хотите познакомится с бесплатным видео-введение в OTP в течение одного часа, см. Стив Виноски “Erlang’s Open Telecom Platform (OTP) Framework”. Вы, наверное, уже знаком материал первого полчаса или около того. Обзор отличный. В совершенно ином стиле, если вы хотите понять, почему стоит изучать OTP и процессно-ориентированное программирование в целом, познакомьтесь со слайдами Франческо

Чезарини.

Создание сервисов в gen_server

Большая часть работы, которую вы считаете основой программы – вычисление результатов, хранение информации и подготовка ответов – будет описываться поведением `gen_server`. Он предоставляет основной набор методов, которые позволяют вам настроить процесс, отвечать на запросы, корректно завершать процесс и даже передавать состояние новому процессу, если этот процесс необходимо обновить на месте.

В таблице 11-1 показаны методы, которые необходимо реализовать в службе, которая использует поведение `gen_server`. Для простого сервиса первые два или три являются наиболее важными, а для остальных вы можете просто использовать код заполнителя.

Таблица 11-1. Что вызывает и получает запрос в gen_server

Метод	Запускается с помощью	Резутат
`init/1`	`gen_server:start_link`	Настройка процесса
`handle_call/3`	`gen_server:call`	Обзработка синхронных вызовов
`handle_cast/2`	`gen_server:cast`	Обзработка асинхронных вызовов
`handle_info/2`	случайные сообщения	Взаимодействие с не OTP-сообщениями
`terminate/2`	ошибки или сигналы прекращения работы от supervisor	Очистка состояния процесса
`code_change/3`	Системные библиотеки для обновления кода	Обновление кода без потери состоняия процесса

В [приложении Б] показан полный шаблон gen_server из emacs-режима Erlang. В этом контексте шаблон – это просто файл, который можно использовать в качестве основы для создания собственного кода.) Однако он довольно большой. Пример 11-1, который вы можете найти в ch11/ex1-drop, показывает менее подробный пример (на основе шаблона), который вы можете использовать для начала. Он смешивает простой расчет из примера 2-1 со счетчиком, как в примере 8-4.

Пример 11-1. Простой пример gen_server на основе шаблона из режима Erlang для Emacs

```
-module(drop).
-behaviour(gen_server).
-export([start_link/0]). % convenience call for startup
-export([init/1,
        handle_call/3,
        handle_cast/2,
        handle_info/2,
        terminate/2,
        code_change/3]). % gen_server callbacks
-define(SERVER, ?MODULE). % macro that just defines this module as server
-record(state, {count}). % simple counter state

%%% convenience method for startup
start_link() ->
        gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).

%%% gen_server callbacks
init([]) ->
        {ok, #state{count=0}}.
```

```
handle_call(_Request, _From, State) ->
        Distance = _Request,
        Reply = {ok, fall_velocity(Distance)},
        NewState=#state{ count = State#state.count+1 },
        {reply, Reply, NewState}.

handle_cast(_Msg, State) ->
        io:format("So far, calculated ~w velocities.~n", [State#state.count]),
        {noreply, State}.

handle_info(_Info, State) ->
        {noreply, State}.

terminate(_Reason, _State) ->
        ok.

code_change(_OldVsn, State, _Extra) ->
        {ok, State}.

%%% Internal functions
fall_velocity(Distance) -> math:sqrt(2 * 9.8 * Distance).
```

Название модуля (`drop`) должно быть знакомо из предыдущих примеров. Вторая строка – это объявление `-behaviour`, указывающее, что оно будет использовать поведение `gen_server`. Это объявление сообщает Erlang, что он может ожидать, что этот код будет поддерживать основные функции обратного вызова этого поведения.

Вы можете записать `-behaviour` объявление `-behavior`, если предпочитаете американскую версию. Erlang поймёт обе.

Объявления `-export` довольно стандартны, хотя они метод `start_link/0` объявлен отдельно от основных методов `gen_server`. В этом нет необходимости, но это хорошее напоминание о том, что `start_link` не требуется для работы поведения `gen_server`. (Он вызывает код `gen_server`, но не является обратным вызовом.)

Декларация `-define`, вероятно, вам незнакома. Erlang позволяет вам объявлять макросы, используя `-define`. Макросы – это простые замены текста. Это объявление сообщает компилятору, что всякий раз, когда он встречает `?SERVER`, он должен заменить его на `?MODULE`. Что такое `?MODULE`? Это встроенный макрос, который всегда ссылается на имя модуля, в котором он появляется. В этом случае это означает, что он будет обработан в `drop`. (Вы можете найти случаи, когда вы хотите зарегистрировать сервер под именем, отличным от имени модуля, но это работоспособное значение по умолчанию.)

Объявление `-record` должно быть знакомым, хотя оно содержит только одно поле, чтобы вести подсчет количества выполненных вызовов. Многие сервисы будут иметь больше полей, включая такие как соединения с базой данных, ссылки на другие процессы, возможно, информацию о сети и метаданные, характерные для этой конкретной службы. Также возможно иметь сервисы без состояния, которые будут представлены здесь пустым кортежем. Как вы увидите ниже, каждая функция `gen_server` будет ссылаться на состояние.

Объявление записи `state` является хорошим примером объявления записи, которое вы должны сделать внутри модуля, а не объявлять через включенный файл. Вполне возможно, что вы захотите поделиться моделями состояний в разных процессах `gen_server`, но проще понять, какова структура переменной `State`, если информация о ней будет внутри модуля.

Первая функция в примере `start_link/0` *не* является одной из обязательных функций `gen_server`. Она важна, так как запускает функцию `gen_server start_link` для запуска процесса. Когда вы только начинаете работать с OTP, это полезно для тестирования. По мере продвижения к созданию профессионального кода, вам может оказаться проще использовать другие механизмы.

Функция `start_link/0` использует макрос `?SERVER`, определенный в объявлении `-define`, а также встроенное объявление `?MODULE`.

```
%%% convenience method for startup
start_link() ->
        gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).
```

Первый аргумент `gen_server:start_link`, кортеж, начинающийся с атома, который должен быть `local` или `global`, в зависимости от того, хотите ли вы, чтобы имя процесса было зарегистрировано только в локальном экземпляре Erlang или во всех связанных узлах. Макрос `?SERVER` будет расширен до `?MODULE`, который сам по себе будет расширен до имени текущего модуля и будет использоваться в качестве имени для этого процесса. Второй аргумент – это имя модуля, обозначенное здесь макросом `?MODULE`, а затем список аргументов и параметров. В этом случае они оба пусты. Параметры могут указывать такие данные, как параметры отладки, тайм-ауты и параметры для запуска процесса.

Вы также можете увидеть форму `gen_server:start_link` с `via` в качестве атома в первом кортеже. Это позволяет вам настраивать пользовательские реестры процессов, из которых наиболее известен `gproc`. Подробнее об этом см. https://github.com/uwiger/gproc.

Все остальные функции являются частью поведения `gen_server`. `init/1` создает новый экземпляр записи состояния и устанавливает поле счетчика в ноль (скорости еще не рассчитаны). Две функции, которые выполняют большую часть работы здесь – это `handle_call/3` и `handle_cast/2`. В этом примере `handle_call/3` ожидает получить расстояние в метрах и возвращает скорость падения с этой высоты на Земле, в то время как handle_cast/2 представляет собой триггер, сообщающий количество рассчитанных скоростей.

`handle_call/3` упрощает синхронную связь между процессами Erlang.

```
handle_call(_Request, _From, State) ->
        Distance = _Request,
        Reply = {ok, fall_velocity(Distance)},
        NewState=#state{ count = State#state.count+1 },
        {reply, Reply, NewState}.
```

Здесь извлекается расстояние из `_Request`, в котором нет необходимости, за исключением того, что я хотел оставить имена переменных для функции такими же, как они были в оригинальном шаблоне. (Написание `handle_call(Distance, _From, State)` было бы более ясным.) Переменная `_Request`, скорее всего, будет содержать кортеж или список, а не пустым значением, но это работает и для простых вызовов.

Затем создаётся ответ на основе отправки этого расстояния простой функции `fall_velocity/1` в конце модуля. Затем он инициируется переменная `NewState`, содержащая увеличенный счетчик. Затем атом `replay`, кортеж `Reply`, содержащий

расчитанную скорость, и `NewState`, содержащий количество выполненных вызовов, возвращаются обратно.

Поскольку вычисления действительно просты, обработка отбрасывания как простого синхронного вызова вполне приемлема. В более сложных ситуациях, когда вы не можете предсказать, сколько времени может занять ответ, вы можете рассмотреть возможность ответа с атомом `noreply` и использования аргумента `_From` для отправки ответа позже. (Существует также возможность отправить `stop` в качестве ответа, который вызовет метод `terminate/2` и остановит процесс.)

По умолчанию OTP отключает все синхронные вызовы, для расчета которых требуется больше 5 секунд. Вы можете переопределить это, сделав ваш вызов с помощью `gen_server:call/3`, чтобы явно указать время ожидания (в миллисекундах), или с помощью атома `infinity`.

Функция `handle_cast/2` поддерживает асинхронную связь. Она не должна возвращать что-либо напрямую, хотя она может вернуть `noreplay` (или `stop`) и обновить состояние. В данном примере это весьма ненадёжное решение, но оно хорошо подходит для демонстрации, вызывая `io:format/2` для отчета о количестве вызовов:

```
handle_cast(_Msg, State) ->
       io:format("So far, calculated ~w velocities.~n", [State#state.count]),
       {noreply, State}.
```

Состояние не изменяется, потому что запрос количества вычислений скорости падения процесса – это не то же самое, что вычисление скорости падения.

Пока у вас нет веских причин для их изменения, вы можете игнорировать функции `handle_info/2`, `terminate/2` и `code_change/3`.

Запуск процесса `gen_server` и его вызов выглядит несколько иначе, чем запуск процессов, которые вы видели в Главе 8. Будьте очень осторожны при вводе следующих выражений в оболочке, так как ошибки, как вы скоро увидите, могут иметь неожиданные последствия:

```
1> c(drop).
{ok,drop}
2> drop:start_link().
{ok,<0.33.0>}
3> gen_server:call(drop, 20).
{ok,19.79898987322333}
4> gen_server:call(drop, 40).
{ok,28.0}
5> gen_server:call(drop, 60).
{ok,34.292856398964496}
6> gen_server:cast(drop, {}).
So far, calculated 3 velocities.
ok
```

Вызов `drop:start_link()` настраивает процесс и делает его доступным. Затем вы можете использовать `gen_server:call` или `gen_server:cast` для отправки ему сообщений и получения ответов.

Хотя вы можете получить ссылку на `pid`, вам не нужно хранить её, чтобы использовать этот процесс. Поскольку `start_link` возвращает кортеж, если вы хотите получить `pid`, вы можете сделать что-то вроде `{ok, Pid} = drop:start_link()`.

В способе, которым OTP вызывает функции `gen_server`, есть дополнительный плюс – или, возможно, опасность – в том, что вы можете обновлять код на лету. Например, я настроил функцию `fall_velocity/1`, чтобы немного ослабить гравитацию Земли, используя `9.1` в качестве константы вместо `9.8`. Перекомпиляция кода и запрос скорости дают другой ответ:

```
7> c(drop).
{ok,drop}
8> gen_server:call(drop, 60).
{ok,33.04542328371661}
```

Это может быть очень удобно на этапе разработки, но будьте осторожны, делая что-то подобное на рабочей машине. OTP имеет другие механизмы для обновления кода на лету. Этот подход также имеет встроенное ограничение: `init` вызывается только тогда, когда `start_link` устанавливает сервис. Эта функция не вызывается, если вы перекомпилировали код. Если ваш новый код требует каких-либо изменений в структуре его состояния, ваш код станет неработоспособным при следующем вызове.

Простой supervisor

Когда вы запустили модуль `drop` из оболочки, вы фактически сделали оболочку супервизором (`supervisor`) для этого модуля (хотя на самом деле оболочка не осуществляет никакого надзора). Вы можете легко прекратить работу модуля:

```
9> gen_server:call(drop, -60).
=ERROR REPORT==== 2-Dec-2012::21:14:51 ===
** Generic server drop terminating
** Last message in was -60
** When Server state == {state,0}
** Reason for termination ==
** {badarith,[{math,sqrt,[-1176.0],[]},
       {drop,fall_velocity,1,[{file,"drop.erl"},{line,42}]},
       {drop,handle_call,3,[{file,"drop.erl"},{line,23}]},
       {gen_server,handle_msg,5,[{file,"gen_server.erl"},{line,588}]},
       {proc_lib,init_p_do_apply,3,
       [{file,"proc_lib.erl"},{line,227}]}]}
** exception exit: badarith
       in function math:sqrt/1
       called as math:sqrt(-1176.0)
       in call from drop:fall_velocity/1 (drop.erl, line 42)
       in call from drop:handle_call/3 (drop.erl, line 23)
       in call from gen_server:handle_msg/5 (gen_server.erl, line 588)
       in call from proc_lib:init_p_do_apply/3 (proc_lib.erl, line 227)
10> gen_server:call(drop, 60).
** exception exit: {noproc,{gen_server,call,[drop,60]}}
       in function gen_server:call/2 (gen_server.erl, line 180)
```

Сообщение об ошибке весьма подробно. Собщено, даже последнее сообщение. Но когда вы снова вызываете службу в строке 10, ее там уже нет. Вы можете перезапустить его с помощью `drop:start_link/0` еще раз, но вы не всегда сможите лично наблюдать за вашими процессами.

Вместо этого вам нужно что-то, что может следить за вашими процессами и убедиться, что они перезапускаются (или нет) в зависимости от ситуации. OTP формализует управление процессом, которое вы видели в Примере 8-10, с его поведением супервизора. Пример B-2 в Приложении B показывает полный шаблон (опять же, из режима Erlang для Emacs), но вы можете создать менее подробный супервизор.

Базовый супервизор должен поддерживать только одну функцию обратного вызова, `init/1`, и также может иметь функцию `start_link` для ее запуска. Возвращаемое значение этой функции `init/1` сообщает OTP, каким дочерним процессом управляет ваш супервизор и как вы хотите обрабатывать их сбои. Супервизор для модуля удаления может выглядеть как Пример 11-2, который находится в *ch11/ex2-drop-sup*.

Пример 11-2. Простой supervisor

```
-module(drop_sup).
-behaviour(supervisor).
-export([start_link/0]). % convenience call for startup
-export([init/1]). % supervisor calls
-define(SERVER, ?MODULE).

%%% convenience method for startup
start_link() ->
        supervisor:start_link({local, ?SERVER}, ?MODULE, []).

%%% supervisor callback
init([]) ->
        RestartStrategy = one_for_one,
        MaxRestarts = 1,
        MaxSecondsBetweenRestarts = 5,
        SupFlags = #{strategy => one_for_one,
                     intensity = 1, % one restart every
                     period => 5}, % five seconds
        Restart = permanent,
        Shutdown = 2000, % milliseconds, could be infinity or brutal_kill
        Type = worker, % could also be supervisor
        Drop = {id => 'drop',
                start => {'drop', start_link, []},
                restart => permanent, % or temporary, or transient
                shatdown => 5000,
                type => worker,
                modules => ['drop']},

        {ok, {SupFlags, [Drop]}}.

%%% Internal functions (none here)
```

Задача функции `init/1` состоит в том, чтобы собрать довольно сложную структуру данных, состоящую из двух структур (карт (map)).

Первая карта, определенная в шаблоне, SupFlags, определяет, как супервизор должен обрабатывать ошибки. Стратегия `one_for_one` сообщает OTP, что он должен создавать новый дочерний процесс каждый раз, когда происходит сбой процесса, который должен быть постоянным. Вы также можете использовать `one_for_all`, который завершает и перезапускает все процессы, которые контролирует супервизор при сбое, или `rest_for_one`, который перезапускает процесс и любые процессы, начавшиеся после запуска сбойного процесса. Существует также `simple_one_for_one`, оптимизированный для случая, когда все дочерние процессы выполняют идентичный код.

Когда вы будете готовы получить более контроль над тем, как ваши процессы реагируют на среду выполнения, вы можете изучить работу динамических функций `supervisor:start/2`, `supervisor:terminate_child/2`, `supervisor:restart_child/2`, `supervisor:delete_child/2`, а также стратегию перезапуска `simple_one_for_one`.

Следующие два значения определяют частоту сбоев рабочих процессов перед завершением самого супервизора. В этом случае по умолчанию используется 1 (интенсивность) перезапуск каждые 5 секунд (период). Настройка этих значений

позволяет вам обрабатывать различные условия, но, вероятно, изначально не сильно повлияет на вас.

Эти значения, которые здесь объединяются в карту, содержащуюся в `SupFlags`, применяются ко всем рабочим процессам, управляемым этим супервизором. Следующие несколько строк определяют свойства, которые применяются только к одному рабочему процессу, в этом случае `gen_server`, указанный в `Drop`. Он предназначен для постоянной службы, поэтому супервизор должен перезапустить его в случае сбоя. Супервизор может подождать 2 секунды, прежде чем полностью его отключить, если только это не супервизор, а рабочий процесс. Более сложные приложения OTP могут содержать деревья супервизоров, управляющих другими супервизорами, которые сами управляют другими супервизорами или работниками.

Карта `Drop` может показаться избыточной, но она создает полный набор информации для запуска процесса. Сначала она указывает идентификатор; а затем кортеж, содержащий имя модуля; содержащего код, функцию, используемую для запуска процесса и список аргументов (аргументов нас нет). Затем указываются тип перезагрузки, завершение работы и тип, а в окончательном списке модулей указываются все модули, от которых будет зависеть этот процесс. В этом случае все это помещается в один модуль, поэтому список содержит только имя этого модуля.

OTP хочет знать зависимости, чтобы помочь вам обновить программное обеспечение на месте. Это всё часть скрытого функционала для поддержания систем в рабочем состоянии, даже не останавливая их.

Теперь, когда у вас есть процесс супервизора, вы можете настроить функцию удаления, просто вызвав супервизор. Однако запуск супервизора из оболочки с использованием вызова функции `start_link/0` создает свой собственный набор проблем. Оболочка сама является супервизором и завершает процессы, сообщающие об ошибках. После длинного сообщения об ошибке вы обнаружите, что ваш работник и руководитель не существуют.

На практике это означает, что существует два способа тестирования контролируемых процессов OTP (которые еще не являются частью приложения) непосредственно из оболочки. Первый явно разрушает связь между оболочкой и процессом супервизора, перехватывая pid супервизора (строка 2), а затем используя функцию `unlink/1` для удаления ссылки (строка 3). Затем вы можете вызвать процесс, как обычно с помощью `gen_server:call/2` и получить ответы. Если вы получите ошибку (строка 6), все будет хорошо. Супервизор перезапустит рабочий процесс и вы сможете успешно делать новые вызовы (строка 7). Вызовы `whereis(drop)` в строках 5 и 8 демонстрируют, что супервизор перезапустил процесс `drop` с новым pid.

```
1> c(drop_sup).
{ok,drop_sup}
2>{ok, Pid} = drop_sup:start_link().
{ok,<0.38.0>}
3> unlink(Pid).
true
4> gen_server:call(drop, 60).
{ok,34.292856398964496}
5> whereis(drop).
<0.39.0>
6> gen_server:call(drop, -60).
=ERROR REPORT==== 2-Dec-2012::21:17:19 ===
** Generic server drop terminating
** Last message in was -60
** When Server state == {state,1}
```

```
** Reason for termination ==
** {badarith,[{math,sqrt,[-1176.0],[]},
       {drop,fall_velocity,1,[{file,"drop.erl"},{line,42}]},
       {drop,handle_call,3,[{file,"drop.erl"},{line,23}]},
       {gen_server,handle_msg,5,[{file,"gen_server.erl"},{line,588}]},
       {proc_lib,init_p_do_apply,3,
              [{file,"proc_lib.erl"},{line,227}]}]}
** exception exit: {{badarith,
              [{math,sqrt,[-1176.0],[]},
              {drop,fall_velocity,1,
              [{file,"drop.erl"},{line,42}]},
              {drop,handle_call,3,
       [{file,"drop.erl"},{line,23}]},
       {gen_server,handle_msg,5,
       [{file,"gen_server.erl"},{line,588}]},
       {proc_lib,init_p_do_apply,3,
       [{file,"proc_lib.erl"},{line,227}]}]},
       {gen_server,call,[drop,-60]}}
       in function gen_server:call/2 (gen_server.erl, line 180)
7> gen_server:call(drop, 60).
{ok,34.292856398964496}
8> whereis(drop).
<0.44.0>
```

Вы также можете указать оболочке прекратить беспокоиться о таких исключениях, введя команду оболочки `catch_exception(true)`. Однако это отключает поведение всей оболочки, что может быть не тем, что вы хотите. (Он вернет `false`, предыдущую настройку для этого свойства. Не волнуйтесь, он установил значение `true`.)

Вы также можете открыть Диспетчер процессов или Обозреватель и отключить рабочие процессы с помощью параметра «Kill» в меню «Trace» и наблюдать за обновлением их pid.

Это работает, но это лишь маленькая часть того, что могут сделать наблюдатли (supervisor). Они могут динамически создавать дочерние процессы и более детально управлять своими жизненными циклами. Эксперименты с ними – лучший способ узнать о их работе.

Упаковка приложений

OTP также позволяет упаковывать наборы компонентов в приложение. Хотя остановить и запустить OTP-работников и супервизоров может быть проще, чем непосредственно работать с процессами, возможности OTP для описания приложений приведут вас к гораздо более легкому запуску, обновлению, администрированию и (если необходимо) остановке ваших проектов.

Приложения Erlang включают в себя два дополнительных компонента, помимо рабочих процессов, супервизоров и связанных файлов, которые им нужны. Файл ресурса приложения, также известный как файл приложения, содержит метаданные вашего приложения. Вам также понадобится модуль с приложением поведения, чтобы определить запуск и остановку.

Если вы работаете на Mac, расширение файла для *.app* файла исчезнет, и операционная система сочтет, что это какое-то нерабочее Mac-приложение. Не беспокойся Он по-прежнему будет работать в Erlang, хотя Mac не будет знать, что делать, если дважды щелкнуть по нему.

Файл приложения представляет собой небольшой кортеж, но его легче прочитать, чем тот, который возвращается функциями супервизора `init/1`. В примере 11-3 в *ch11/ex3-drop-app* показан минимальный файл приложения, помещенный в подкаталог *ebin*, который настраивает это простое приложение `drop`.

Пример 11-3. drop.app, файл ресурсов приложения или файл приложения, для программы drop

```
{application, drop,
[{description, "Dropping objects from towers"},
{vsn, "0.0.1"},
{modules, [drop, drop_sup,drop_app]},
{registered,[drop, drop_sup]},
{applications, [kernel,stdlib]},
{mod, {drop_app,[]} }]}.
```

Первая строка идентифицирует это как приложение с именем **drop**, а затем список аргументов предоставляет дополнительную информацию:

- description - это (иногда) понятное человеку описание того, что здесь. vsn – это номер версии, который в данном случае крошечный.
- modules выводит список модулей, которые составляют приложение. В данном случае `drop`, `drop_sup` и `drop_app`.
- registered список модулей являются видимыми. Это снова `drop` и `drop_sup`.
- applications перечисляет обязательные приложения, от которых зависит это приложение. `kernel` и `stdlib` являются минимальным стандартным набором.
- mod имеет кортеж, который указывает на модуль с поведением приложения. Он может принимать список аргументов, которые будут переданы функции `start/2` модуля, хотя здесь их нет.

Этот модуль тривиален даже по сравнению с другим кодом OTP, который вы видели, как показано в примере 11-4, который также есть в *ch11/ex3-drop-app*. (Пример B-3 показывает более полный шаблон.)

Пример 11-4. Модуль приложения для программы drop

```
-module(drop_app).
-behaviour(application).
-export([start/2, stop/1]).

start(_Type, _StartArgs) ->
       drop_sup:start_link().

stop(_State) ->
       ok.
```

Единственное, что вам действительно нужно сделать – это запустить супервизоры для вашего приложения в функции `start/2`. В этом случае параметры `_Type` и `_StartArgs` не имеют значения.

Запуск этого приложения из оболочки потребует от вас дополнительных усилий. Вам, конечно, нужно будет скомпилировать `drop_app`, но вам также нужно сообщить Erlang о каталоге `ebin`, содержащем файл *drop.app*, как показано в строке 2. (OTP ожидает, что он будет там, но выдаст ошибку"нет такого файла или каталога ", если вы не сообщите Erlang о каталоге явным образом).

```
1> c(drop_app).
{ok,drop_app}
```

```
2> code:add_path("ebin/").
true
3> application:load(drop).
ok
4> application:loaded_applications().
[{kernel,"ERTS CXC 138 10","2.15.2"},
{drop,"Dropping objects from towers","0.0.1"},
{stdlib,"ERTS CXC 138 10","1.18.2"}]
5> application:start(drop).
ok
6> gen_server:call(drop, 60).
{ok,34.292856398964496}
```

Когда Erlang знает, где искать, вы можете использовать функции модуля приложения, чтобы загрузить приложение и убедиться, что Erlang его нашел. После того, как вы запустите приложение, вы можете выполнить вызовы с помощью `gen_server:call`. Поскольку супервизор связан с приложением, вам не нужно беспокоиться о закрытии оболочки. Вы можете пойти дальше и введёте некоректные данные в процесс расчета `drop` с отрицательным значением, и супервизор просто запустит его снова.

```
7> whereis(drop).
<0.45.0>
8> gen_server:call(drop, -60).
=ERROR REPORT==== 2-Dec-2012::21:25:38 ===
** Generic server drop terminating
** Last message in was -60
** When Server state == {state,1}
** Reason for termination ==
** {badarith,[{math,sqrt,[-1176.0],[]},
        {drop,fall_velocity,1,[{file,"drop.erl"},{line,42}]},
        {drop,handle_call,3,[{file,"drop.erl"},{line,23}]},
        {gen_server,handle_msg,5,[{file,"gen_server.erl"},{line,588}]},
        {proc_lib,init_p_do_apply,3,
        [{file,"proc_lib.erl"},{line,227}]}]}
** exception exit: {{badarith,
        [{math,sqrt,[-1176.0],[]},
        {drop,fall_velocity,1,
        [{file,"drop.erl"},{line,42}]},
        {drop,handle_call,3,
        [{file,"drop.erl"},{line,23}]},
        {gen_server,handle_msg,5,
        [{file,"gen_server.erl"},{line,588}]},
        {proc_lib,init_p_do_apply,3,
        [{file,"proc_lib.erl"},{line,227}]}]},
        {gen_server,call,[drop,-60]}}
        in function gen_server:call/2 (gen_server.erl, line 180)
9> gen_server:call(drop, 60).
{ok,34.292856398964496}
10> whereis(drop).
<0.49.0>
```

Есть много, намного больше, что было мной опущено в книге. OTP заслуживает одну или несколько книг для расскытия его возможностей. Надеемся, что эта глава предоставит вам достаточно информации, чтобы опробовать некоторые возможности и лучшеть понять более сложные книги на эту тему. Тем не менее, разрыв между тем, что может разумно представить эта глава, и тем, что вам нужно знать для написания надежных программ на базе OTP, огромем. Желаю в этой интересно работе всем успеха!

Вы можете узнать больше о работе с основами OTP в главах 11 и 12 «Erlang Programming» (O'Reilly); Главы 22 и 23 «Programming Erlang», 2-е издание (Pragmatic); Глава 4 «Erlang and OTP in Action» (Manning); и в главах с 14 по 20 книги «Learn You Some Erlang For Great Good!» (No Starch Press). Вы также можете углубиться в OTP с помощью книги «Designing for Scalability with Erlang/OTP» (O'Reilly).

Глава 12. Следующие шаги в Erlang

Надеюсь, теперь вы чувствуете себя комфортно при написании базовых программ на языке Erlang и понимаете, как модули и процессы встраиваются в программы. Вы должны быть готовы экспериментировать с написанием кода Erlang, но, что более важно, вы должны быть готовы изучить другие ресурсы для освоения Erlang и его многочисленных мощных библиотек. Там есть что исследовать!

Перемещение за оболочку Erlang

Оболочка Erlang – это отличное место для тестирования кода, а также для написания и расширения кода Erlang. Скорее всего, вы будете проводить в оболочке гораздо больше времени, если продолжите использовать Erlang, но способ его использования может измениться.
Вы можете компилировать и запускать код Erlang вне оболочки, что значительно упрощает интеграцию работы Erlang с инструментами, которые вы обычно используете для управления кодом и связанными ресурсами. Модуль make – это средство для запуска, позволяющее создавать файлы *Emakefile*, которые предоставляют инструкции для команды `erl -make`. Команда escript, описанная по адресу http://erlang.org/doc/man/escript.html, позволит вам запустить Erlang из командной строки в различных средах.

Если вы хотите еще больше автоматизировать сборку Erlang, вы можете изучить rebar3. Вы можете использовать rebar3 совместно с другими инструментами, чтобы применить сильные стороны каждого.

Если вы хотите использовать Erlang из IDE, вы можете изучить http://erlide.org/, набор инструментов для работы с Erlang в Eclipse. Пользователи Emacs, возможно, захотят изучить режим Erlang.

Распределенных вычислений

Почти все, что вы узнали в этой книге, указывает на вычислительную модель, которая позволяет легко распределять программы по сети узлов Erlang. Настроить набор узлов не так уж и сложно. Возможно, даже слишком легко (в некоторых отношениях) позволить одержимым безопасностью администраторам спать спокойно. Книга «Designing for Scalability with Erlang/OTP» (O'Reilly) исследует эту тему более подробно.

Прежде чем настраивать большие наборы узлов, вы захотите узнать больше о том, как Erlang планирует запуск кода, как сообщения передаются между узлами и как удаленно администрировать узлы Erlang. Вся эта информация существует, обеспечивая основу для таких инструментов, как OTP и Mnesia.

Обработка двоичных данных

Erlang включает двоичный тип данных, двоичные операторы и множество библиотек для их обработки. Если вам нужно создать сетевые протоколы, которые обрабатывают биты в сети или данные ASN.1, то например, Erlang предлагает мощные инструменты для

получения информации в двоичной форме и из них. Если вы видите числа или строки, заключенные в << и >>, вы, весьма вероятно, уже столкнулись с инструментами Erlang для обработки двоичных данных. Они позволяют вам задавать, сопоставлять с образцом и обрабатывать структуры двоичных данных.

Вход и выход

В главах 4 и 5 вы познакомились с функционалом `io:format` в контексте представления информации в оболочке. Однако модуль `io` предлагает гораздо больше возможностей для чтения и записи данных, а модули `file`, `filename`, `filelib` и `io_lib` предоставляют вам инструменты, необходимые для входа и выхода из файлов. Если ваша работа связна с работой в сети, то я рекомендую также ознакомится с функционалом модулей `gen_tcp`, `gen_udp` и `inet`.

Тестирование, анализ и рефакторинг

Подходы к функциональному программированию должны, как только вы к ним привыкнете, упростить создание чистого кода. Тем не менее, это не всегда удобно, особенно когда вы продвигаетесь к решению более сложных проблем, чем те, которые представлены в этой книге.

Модульное тестирование (Unit testing) – это один из способов убедиться, что ваш код продолжает работать по мере продвижения вперед. Ориентированное на небольшие компоненты ваших программ, которые должны иметь возможность надежно возвращать набор правильных выходных данных из заданного набора входных данных, модульное тестирование может помочь вам узнать, когда вы сделали свой код работающим, и предупредить вас, когда он сломается. Erlang включает платформу EUnit для модульного тестирования и платформу Common Test для системного тестирования.

Erlang также включает в себя Dialyzer, анализатор расхождений для Erlang, который может помочь вам выявить основные ошибки при вводе неверно набранных данных, код, который никогда не вызывается, и аналогичные проблемы, которые компиляторы для статически типизированных языков обычно хороши. Erlang также включает в себя профилировщики и инструменты покрытия; вам следует изучить модули `eprof`, `fprof` и `cover`, а также инструмент cprof.

Если вы переживаете по поводу возможного рефакторинга кода (какак только вы будете работать с большим проектом, то с этой задачей вы обязательно столкнётесь), то вам будет весьма полезен Wrangler. Этот инстумент позволит вам исследовать ваш код и автоматизировать широкий спектр распространенных модификаций программы.

Сеть и Интернет

Erlang – естественная среда, в которой число пользователей постоянно увеличивается (в часности в мире веб-программирования). В сети данные должны плавно перемещаться между узлами. Пользователи веб-приложений хотят, что критически важные веб-приложения могли работать так же надежно, как телефонная система. Большинство веб-приложений представляют собой большие конвейеры данных, хорошо подходящие для этого.

Несколько платформ на основе Erlang позволяют создавать веб-приложения. Cowboy, Yaws и Mochiweb – это веб-серверы, написанные на Erlang. Они предлагают среды, которые должны быть знакомы любому, кто создает веб-приложения. Yaws позволяет вам смешивать код Erlang с HTML (и другими) шаблонами, что позволяет довольно легко собрать веб-сайт или приложение на основе Erlang. Для более всесторонней структур, которым необходимо работать с любым из них подойдёт Nitrogen.

Если вы создаёте сервисы на основе REST, вы можете также изучить Webmachine, инструментарий для обработки HTTP, который приближает вас к основному протоколу сети. Даже если вы в конечном итоге не используете его, изучение его блок-схемы многому научит вас, чтобы лучше понять систему обработки веб-запросов.

Хранилище данных

У вас уже есть ETS, DETS и Mnesia. Что еще может понадобиться?

Многие люди используют Erlang, не зная, что они его используют, поскольку они взаимодействуют с популярными базами данных NoSQL CouchDB и Riak. Так как в их основе лежить Erlang – это облегчает их распространение и управление, что позволяет удовлетворять запросы большой и растущей аудитории. Для краткого ознакомления с ними (и пятью другими вариантами баз данных) вы можите изучить материал книги «Seven Databases in Seven Weeks» (Pragmatic). Книга не научит вас многому об их использовании с Erlang, но даст вам прочную основу, которая поможет вам изучить их интерфейсы Erlang, как только вам придётся сделать что-то интересное.

Многие другие базы данных имеют интерфейсы Erlang, и есть поддержка классических соединений ODBC.

Расширение возможностей Erlang

Если вам вожда производительность в решении сложных задач или вы хотите избежать переписывания библиотеки, написанной на языке, отличном от Erlang, то вы захотите изучить инструменты Erlang для создания программных комплексов совместно с другими языками программирования. «Erlang Programming» (O'Reilly) исследует соединения с Java, C и Ruby, но также отмечает подходы, которые вы можете использовать для соединения с языками .NET, Python, Perl, PHP, Haskell, Scheme и Emacs Lisp. Вы также захотите изучить встроенные функции (NIF) и драйверы.

Языки, созданные на основе Erlang

Программирование на Erlang может быть приятна для функционального программирования, но его структура может показаться хрупкой и неудобной, если вы привыкли к гибкости, которую обеспечивают многие другие языки. Elixir сочетает в себе систему времени исполнения Erlang с совершенно другим (в стиле Ruby) синтаксисом, более сфокусированным на полиморфизме, метапрограммировании и ассоциативных структурах данных. Если вы предпочитаете подходы к функциональному программированию на Лиспе, возможно, вы захотите изучить LFE.

Модель и инструменты среды исполнения Erlang являются мощными и уникальными, и у вас могут появиться другие замечательные идеи, которые позволят вам применить их к работе.

Сообщество

Когда вы узнаете больше об Erlang, вы найдете сообщество, которое с радостью поможет вам на каждом уровне. Список рассылки erlang-questions приветствует новичков. Во время написания этой книги я обнаружил, что ее архивы невероятно ценны. Вы можете найти информацию о подписке и архиве по адресу http://erlang.org/mailman/listinfo/erlang-questions, и, вероятно, вы будете регулярно просматривать их архивы, если будете выполнять поиск по темам Erlang. Если вы предпочитаете «настоящий» общение в реальном времени электронной почте, на freenode есть также IRC-канал #erlang и Erlang slack.

Если вы предпочитаете реальное общение, то вы можите посетить конфереции, которые регулярно проводятся. Erlang Factory производит множество мероприятий различного формата по всему миру, включая Erlang User Conference. Специальная группа по изучению языков программирования (SIGPLAN) Ассоциации вычислительной техники (ACM) также проводит семинары по Erlang.

Также об Erlang говорят на многих более неформальных мероприятий «Erlounges» в самых разных уголках планеты, а такжена на более крупные конференции, такие как Open Source Convention (OSCON), включают в себя сессии Erlang и учебные пособия.

Если вы хотите изучить код Erlang, его много на GitHub. Вы можете посмотреть на самые активные проекты, посетив GitHub.

Делитесь знаниями о Erlang

Изучение Erlang могло показаться лёгкой прогулкой. Широкий переход от одиночных компьютеров к сетевым и распределенным системам многопроцессорных вычислений дает среде Erlang огромное преимущество перед практически любой другой средой. Все больше и больше компьютерного мира начинают сталкиваться именно с теми проблемами, для решения которых был создан Erlang. Ветераны этих испытаний могут почувствовать облегчение, когда обнаружат существование Erlang. Они могут прекратить обдумывать наборы инструментов, которые слишком старались перенести односистемные подходы в многосистемный мир.

В то же время, однако, я бы посоветовал вам подумать о мудрой мысли Джо Армстронга: *«Новые технологии имеют наилучшие шансы для использования а) сразу после катастрофы или б) при запуске нового проекта»*.

Возможно, вы читаете эту книги, потому что в проекте, над которым вы работаете, уже произошла катастрофа (или вы подозреваете, что она скоро случится). Проще всего применить Erlang к новым проектам, предпочтительно в проектах, в которых неизбежные ошибки новых проектом могут быть легко преодолимы.

Найдите проекты, которые кажутся вам забавными и которыми вы можете поделиться в своей организации или со всем миром. Нет лучшего способа продемонстрировать всю

мощь языка программирования и среды, чем создавать из этого замечательные полезные продукты!

Приложение А. Элементы Erlang

Как и любой языке программирования, у Erlang есть своя экосистема, с которой всегда интересно ознакомиться. В положении я собрал основные справочные материалы, которые вам могут пригодиться для лучшего понимания материала книги. Если вам этого будет недостаточно и вы захотите узнать больше, пожалуйста, посетите руководство пользователя, пройдя по ссылке http://erlang.org/doc/reference_manual/users_guide.html.

Команды оболочки

Вы можете использовать большинство функций Erlang в оболочки. Обратите внимание, что есть функции (команды оболочки), которые будут работать только в оболочке.

Таблица 0-1-1. Команды оболочки Erlang

Команда	Действие
`q()`	Завершает работу оболочки *и* работу виртуальной машины Erlang
`C(file)`	Компилирует выбранный файл *.erl
`b()`	Выводит список переменных текущего сеанса работы оболочки
`F()`	Удаляет все переменные
`F(X)`	Удаляет выбранную переменную
`h()`	Выводит список введенных команды с текущем сеансе работы оболочки
`e(N)`	Повторяет команду по номеру строки N
`V(N)`	Выводит значение команды по номеру строки N
`catch_exeption(boolean)`	Определяет поведение оболочки при возникновении ошибки
`rd(Name,Definition)`	Создаёт запись с именем Name и содержанием Definition
`rr(File)`	Создаёт запись на основании содержания файла *.hrl
`rf()`	Удаляет все записи, созданные в текущем сеансе работы оболочки
`rl()`	Выводит список доступных записей
`pwd()`	Выводит путь к текущему рабочему каталогу
`ls()`	Выводит список всех файлов и каталогов в текущем каталоге
`cd(Directory)`	Изменяет текущие рабочий каталог на каталог определённый параметром Directory

Зарезервированные слова

Есть несколько с атомов, которые вы не можете использовать вне их предполагаемого контекста.

Компилятору Erlang будет пытаться интерпретировать выражения, если вы используете определенные ключевые слова в качестве атомов или имен функций. Он будет интерпретировать их определённым образом и вы можете получать очень странные ошибки компиляции. Например, вы заходите инициировать переменную значением band, верно? Компилятору это не понравится.

Таблица A-2 Список зарегистрированных слов

after	and	andalso	band	begin	bnot	bor	bsl	bsr	bxor
case	catch	cond	div	end	fun	if	let	not	of
or	orelse		receive	rem	try	when	xor		

Ваши функции не должны называться именами из списка зарегистрированных слов. Точка. Для значений переменных есть лазейка. Вы можете заключить зарезервированное слово в кавычки. Например: 'receive'.

```
1> A1 = 'after'.
'after'
2> A2 = 'and'.
'and'
3> A3 = 'andalso'.
'andalso'
4> A4 = 'band'.
'band'
5> A5 = 'begin'.
'begin'
6> A6 = 'bnot'.
'bnot'
7> A7 = 'bor'.
'bor'
8> A8 = 'bsl'.
'bsl'
9> A9 = 'bsr'.
'bsr'
10> A10 = 'bxor'.
'bxor'
11> A11 = 'case'.
'case'
12> A12 = 'catch'.
'catch'
13> A13 = 'cond'.
'cond'
14> A14 = 'div'.
'div'
15> A15 = 'end'.
'end'
16> A16 = 'fun'.
'fun'
17> A17 = 'if'.
'if'
18> A18 = 'let'.
'let'
19> A19 = 'not'.
'not'
20> A20 = 'of'.
'of'
21> A21 = 'or'.
'or'
22> A22 = 'orelse'.
'orelse'
23> A23 = 'query'.
query
24> A24 = 'receive'.
'receive'
25> A25 = 'rem'.
'rem'
26> A26 = 'try'.
'try'
27> A27 = 'when'.
'when'
28> A28 = 'xor'.
```

```
'xor'
29>
```

Хотя следующие атомы и не являются зарезервированными словами, они обычно используются в возвращаемых значениях функций. Вероятно, лучше использовать их только в тех случаях, когда их возвращение ожидается.

Таблица А-3. Часто используемые возвращаемые значения функций

Команда	Действие
`ok`	Стандартное возвращаемое значение функции. (Это не значит, что выполнена успешно.)
`error`	Что-то пошло не так. Обычно сопровождается более широким объяснением.
`undefined`	Значение еще не было назначено. Обычно используется, как значение неопределенного поля записи.
`reply`	Ответ включен с некоторым возвращаемым значением.
`noreply`	Возвращаемое значение не включено. Однако какой-то ответ может исходить от другого сообщения.
`stop`	Используется в OTP для информирования о том, что сервер должен остановиться. После должна запуститься функция terminate.
`ignore`	Возвращается процессом супервизора OTP, который не может запустить дочерний процесс.

Операторы

Таблица А-4. Логические (булевые) операторы

Оператор	Описание
`and`	логическое И
`or`	логическое ИЛИ
`xor`	логическое исключающее ИЛИ
`not`	НЕ

Оператор not выполняется первым.

andalso и orelse также являются логическими операторами для логического И и логического ИЛИ, но они являются сокращенными операторами. Если им не нужно обрабатывать все аргументы, они останавливают анализ на первом, который дает им определенный ответ.

Таблица А-5. Операторы сравнения

Оператор	Описание
==	равно
/=	не равно
=<	меньше или равно
<	меньше
>=	больше или равно
>	больше
=:=	строго равно
=/=	строго не равно

Вы можете сравнивать элементы разных типов в Erlang. Отношение типов от «наименьшего» к «величайшему» таково:

```
number < atom < reference < fun < port < pid < tuple < list < bit string
```

Внутри числа вы можете сравнивать целые числа и числа с плавающей запятой, за исключением случаев, когда используются операторы =:= и =/=, которые возвращают false при сравнении чисел разных типов.

Вы также можете сравнивать кортежи, даже если они содержат разное количество значений. Erlang будет проходить через кортежи слева направо, сравнивая элементы по порядку.

Таблица А-6. Арифметические операторы

Оператор	Описание
`+`	унарный + (положительное число)
`-`	унарный - (отрицательное число)
`+`	сложение
`-`	вычитание
`*`	умножение
`/`	деление
`div`	целочисленное деление
`rem`	Целочисленный остаток от деления (X/Y)

Таблица А-7. Бинарные операторы

Оператор	Описание
`bnot`	унарное бинарное не
`band`	бинарное И
`bor`	бинарное ИЛИ
`bxor`	арифметическое бинарное исключающее ИЛИ
`bsl`	арифметический бинарный сдвиг влево
`bsr`	арифметический бинарный сдвиг вправо

Таблица А-8. Приоритет оператора, от высшего к низшему

Оператор	Описание
:	
#	
Унарный + - bnot not	
/ * div rem band and	Лево ассоциативный
= + - bor bxor bsl bsr or	
xor	Лево ассоциативный
++ --	Право ассоциативный
== /= =< >= =:= =/=	
andalso	
orelse	
= !	Право ассоциативный
catch	

Оператор с наивысшим приоритетом в выражении вычисляется первым. Последовательность выполнения оператором Erlang с одинаковым приоритетом зависит от ассоциативных путей (левые ассоциативные операторы идут слева направо, правые ассоциативные операторы идут справа налево).

Компоненты охранных выражений

Erlang допускает ограниченное подмножество функций и операторов в охранных выражениях. Правило выбора элементов этого списка довольно простое – «без побочных эффектов». Это ограничения необходимо для сохранения простоты подмножество операторов и функций. Список разрешенных компонентов включает в себя следующие:

- true
- Другие константы (включая false)
- Операторы сравнения (Таблица A-5)
- Арифметические выражения (Таблица A-6)
- Логические выражения (в том числе сокращенные andalso orelse)
- Следующие BIF (**b**uilt **i**nto **E**rlang) функции: `is_atom/1`, `is_binary/1`, `is_bitstring/1`, `is_boolean/1`, `is_float/1`, `is_function/1`, `is_function/2`, `is_integer/1`, `is_list/1`, `is_map/1`, `is_number/1`, `is_pid/1`, `is_port/1`, `is_record/2`, `is_record/3`, `is_reference/1`, `is_tuple/1`.

Кроме этих BIF функций есть ещё устаревшие, которые оставлены в новых версиях Erlang по причинам обратной совместимости (`abs(Number)`, `bit_size(Bitstring)`, `byte_size(Bitstring)`, `element(N, Tuple)`, `float(Term)`, `hd(List)`, `length(List)`, `map_size(Map)`, `node()`, `node(Pid|Ref|Port)`, `round(Number)`, `self()`, `size(Tuple|Bitstring)`, `tl(List)`, `trunc(Number)`, `tuple_size(Tuple)`). Их не рекомендуется использовать в новом коде в охранных выражениях. Также их приминимать ограничена охранными выражениями верхнего уровня[2].

Часто используемые функции

Таблица A-9. Математические функции

Функция	Описание
`math:pi/0`	Число Пи
`math:sin/1`	Синус
`math:cos/1`	Косинус
`math:tan/1`	Тангенс
`math:asin/1`	Арксинус
`math:acos/1`	Арккосинус
`math:atan/1`	Арктангенс
`math:atan2/2`	Арктангенс, который понимает квадранты
`math:sinh/1`	Гиперболический синус
`math:cosh/1`	Гиперболический косинус
`math:tanh/1`	Гиперболический тангенс
`math:asinh/1`	Гиперболический арксинус
`math:acosh/1`	Гиперболический косинус
`math:atanh/1`	Гиперболический арктангенс
`math:exp/1`	Экспонента
`math:log/1`	Натуральный логарифм
`math:log10/1`	Логарифм по основанию 10
`math:pow/2`	Функция возведения в степень
`math:sqrt/1`	Квадратный корень
`math:erf/1`	Интеграл вероятности ошибки
`math:erfc/1`	Добавление функции ошибок

Аргументы для всех тригонометрических функций выражены в радианах. Чтобы преобразовать градусы в радианы, разделите их 180 и умножьте на число Пи.

[2] http://erlang.org/doc/reference_manual/expressions.html#guard-sequences

Функции `math:erf/1` и `math:erfc/1` могут быть не реализованы для ОС Windows. В документации так и пишут, что не все функции могут реализованы. Это касается библиотек, которые реализуются на языке Си (а модуль math как раз реализован на Си).

Таблица А-10. Доступные функции высшего порядка для обработки списков

Функция	Возвращаемое значение	Описание
`lists:foreach/2`	ok	Побочные эффекты, указанные в функции
`lists:map/2`	новый список	Применить функцию к списку значений
`lists:filter/2`	подсписок	Создание списка, где функция возвращает true
`lists:all/2`	boolean	Возвращает true, если функция true для всех значений, иначе false
`lists:any/2`	boolean	Возвращает true, если функция true для любых значений, иначе false
`lists:takewhile/2`	подсписок	Собирает заголовок списка, пока функция не станет истинной
`lists:dropwhile/2`	подсписок	Удаляет заголовок списка, пока функция не станет истинной
`lists:foldl/3`	аккумулятор	Передает значение списка функций и аккумулятор, обрабатывая список
`lists:foldr/3`	аккумулятор	Передает значение списка функций и накопитель, обрабатывая список
`lists:partition/3`	кортеж из двух списков	Разделить список на основе функции
`lists:foreach/2`	ok	Побочные эффекты, указанные в функции
`lists:map/2`	новый список	Применить функцию к списку значений
`lists:filter/2`	подсписок	Создание списка, где функция возвращает true
`lists:all/2`	boolean	Возвращает true, если функция true для всех значений, иначе false
`lists:any/2`	boolean	Возвращает true, если функция true для любых значений, иначе false
`lists:takewhile/2`	подсписок	Собирает заголовок списка, пока функция не станет истинной
`lists:dropwhile/2`	подсписок	Удаляет заголовок списка, пока функция не станет истинной
`lists:foldl/3`	аккумулятор	Передает значение списка функций и аккумулятор, обрабатывая список
`lists:foldr/3`	аккумулятор	Передает значение списка функций и накопитель, обрабатывая список
`lists:partition/3`	кортеж из двух списков	Разделить список на основе функции
`lists:foreach/2`	ok	Побочные эффекты, указанные в функции

В Главе 7 эта тема раскрывается более подробно.

Строки и форматирование

Таблица А-11. Простые последовательности управления для функций io:format и error_logger

Последовательность	Возвращаемое значение
`~p`	Значение, удобное для чтения
`~w`	Значение, без отступа
`~s`	Содержимое строки
`~c`	ASCII символ, соответствующий числу
`~tc`	Unicode символ, соответствующий числу
`~i`	Игнорирование входящего элемента
`~n`	Новая строка (не используетсписок аргументов)

Таблица А-12. Escape-последовательности для строк

Escape-последовательность	Возвращаемое значение
`\"`	двойная кавычка
`\'`	одиночная кавычка
`\\`	обратный слэш
`\b`	возврат на одну позицию
`\d`	удалить символ
`\e`	escape
`\f`	form feed
`\n`	новая строка
`\r`	возврат каретки
`\s`	пробел
`\t`	табуляция
`\v`	вертикальная табуляция
`\XYZ, \YZ, \Z`	символ с восьмеричным представлением XYZ, YZ или Z
`\xXY`	символ в шестнадцатеричном представлении
`\x{X…}`	символы в шестнадцатеричном формате, где X… – один или несколько шестнадцатеричных символов
`^a…\^z or ^A…\^Z`	control-A to control-Z

Таблица А-13. Функции для работы со строками

Escape-последовательность	Возвращаемое значение
`string:len/1`	Длина строки (проходит по всей стоке, поэтому работает медленно с длинными строками)
`length/1`	Длина строки (проходит по всей стоке, поэтому работает медленно с длинными строками)
`string:concat/2`	Объединение двух строк в одну
`lists:concat/1`	Объединение двух строк в одну
`lists:append/1-2`	Объединение двух строк в одну
`lists:nth/2`	Символ в указанной позиции.
`hd/1`	Первый символ строки.
`string:chr/2`	Позиция, в которой указанный символ появляется впервые.
`string:str/2`	Положение подстроки в строке.
`string:substr/2-3`	Сегмент из строки в заданной позиции заданной длины.
`string:sub_string/2-3`	Сегмент из строки между двумя позициями.
`string:tokens/2`	Список фрагментов строки, разделенный текстом

`string:join/2`	Строка, созданная из нескольких с добавленным разделителя
`string:words/1-2`	Количество слов в строке.
`string:chars/2-3`	Строка, которая повторяет данный символ указанное количество раз.
`string:copies/2`	Строка, которая повторяет данную строку заданное количество раз.
`string:strip/1-3`	Строка с начальными и / или конечными пробелами (или указанными символами) удалена.
`string:left/2-3`	Строка указанной длины, дополненная пробелами справа, если это необходимо.
`string:right/2-3`	Строка указанной длины, дополненная пробелами слева, если это необходимо.
`string:centre/2-3`	Строка указанной длины, дополненная пробелами слева и справа, если это необходимо.
`lists:reverse/1-2`	Строка в обратном порядке.
`string:to_float/1`	Содержимое строки с плавающей точкой плюс остатки или кортеж ошибки.
`string:to_integer/1`	Целочисленное содержимое строки плюс остатки или кортеж ошибки.
`string:to_lower/1`	Версия строки со всеми заглавными (латинскими) символами, преобразованными в строчные.
`string:to_upper/1`	Версия строки со всеми строчными (Latin-1) символами, преобразованными в верхний регистр.
`integer_to_list/1-2`	Строковая версия целого числа, необязательно в указанном основании
`float_to_list/1`	Строковое представления числа с плавающей запятой

Примечание: я работаю над созданием единого модуля-обертки, который собирает инструменты Erlang для работы со строками в одном месте. Для получения дополнительной информации посетите https://github.com/simonstl/erlang-simple-string

Типы данных для документации и анализа

Таблица А-14. Основные типы данных для -spec и EDoc

`atom()`	`binary()`	`float()`	`fun()`	`integer()`	`list()`	`tuple()`
`union()`	`node()`	`number()`	`string()`	`char()`	`byte()`	`[] (nil)`
`any()`	`pid()`	`port()`	`reference()`			

Для получения более подробной информации, пожалуйста перейдите по ссылке http://erlang.org/doc/reference_manual/typespec.html

Приложение B. Шаблоны OTP

Это полные шаблоны для gen_server, supervisor и application. Некоторые части более полезны, чем другие, но может быть полезно увидеть полный набор ожидаемых ответов. В этом контексте шаблон – это просто файл, полный кода, который вы можете использовать в качестве основы для создания собственного кода.

Помните, что атом noreply не означает «никогда не будет ответа», но скорее «этот ответ не был получен».

Пример B-1. Шаблон gen_server

```
 %%%-------------------------------------------------------------------
%%% @author $author
%%% @copyright (C) $year, $company
%%% @doc
%%%
%%% @end
%%% Created : $fulldate
%%%-------------------------------------------------------------------
-module($basename).

-behaviour(gen_server).

%% API
-export([start_link/0]).

%% gen_server callbacks
-export([init/1,
      handle_call/3,
      handle_cast/2,
      handle_info/2,
      terminate/2,
      code_change/3]).

-define(SERVER, ?MODULE).

-record(state, {}).

%%%===================================================================
%%% API
%%%===================================================================

  %%--------------------------------------------------------------------
%% @doc
%% Starts the server
%%
%% @spec start_link() -> {ok, Pid} | ignore | {error, Error}
%% @end
%%--------------------------------------------------------------------
start_link() ->
      gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).

%%%===================================================================
%%% gen_server callbacks
%%%===================================================================
```

```
%%--------------------------------------------------------------------
%% @private
%% @doc
%% Initializes the server
%%
%% @spec init(Args) -> {ok, State} |
%% {ok, State, Timeout} |
%% ignore |
%% {stop, Reason}
%% @end
%%--------------------------------------------------------------------
init([]) ->
      {ok, #state{}}.

%%--------------------------------------------------------------------
%% @private
%% @doc
%% Handling call messages
%%
%% @spec handle_call(Request, From, State) ->
%% {reply, Reply, State} |
%% {reply, Reply, State, Timeout} |
%% {noreply, State} |
%% {noreply, State, Timeout} |
%% {stop, Reason, Reply, State} |
%% {stop, Reason, State}
 %% @end
%%--------------------------------------------------------------------
handle_call(_Request, _From, State) ->
      Reply = ok,
      {reply, Reply, State}.

%%--------------------------------------------------------------------
%% @private
%% @doc
%% Handling cast messages
%%
%% @spec handle_cast(Msg, State) -> {noreply, State} |
%% {noreply, State, Timeout} |
%% {stop, Reason, State}
%% @end
%%--------------------------------------------------------------------
handle_cast(_Msg, State) ->
      {noreply, State}.

%%--------------------------------------------------------------------
%% @private
%% @doc
%% Handling all non call/cast messages
%%
%% @spec handle_info(Info, State) -> {noreply, State} |
%% {noreply, State, Timeout} |
%% {stop, Reason, State}
%% @end
%%--------------------------------------------------------------------
handle_info(_Info, State) ->
      {noreply, State}.

%%--------------------------------------------------------------------
%% @private
%% @doc
%% This function is called by a gen_server when it is about to
%% terminate. It should be the opposite of Module:init/1 and do any
%% necessary cleaning up. When it returns, the gen_server terminates
```

```
%% with Reason. The return value is ignored.
%%
%% @spec terminate(Reason, State) -> void()
%% @end
%%--------------------------------------------------------------------
terminate(_Reason, _State) ->
      ok.

%%--------------------------------------------------------------------
%% @private
%% @doc
%% Convert process state when code is changed
%%
%% @spec code_change(OldVsn, State, Extra) -> {ok, NewState}
%% @end
%%--------------------------------------------------------------------
code_change(_OldVsn, State, _Extra) ->
      {ok, State}.

%%%===================================================================
%%% Internal functions
%%%===================================================================
```

Пример B-2. Шаблон supervisor

```
%%%-------------------------------------------------------------------
%%% @author $author
%%% @copyright (C) $year, $company
%%% @doc
%%%
%%% @end
%%% Created : $fulldate
%%%-------------------------------------------------------------------
-module($basename).

-behaviour(supervisor).

%% API
-export([start_link/0]).

%% Supervisor callbacks
-export([init/1]).

-define(SERVER, ?MODULE).

%%%===================================================================
%%% API functions
%%%===================================================================

%%--------------------------------------------------------------------
%% @doc
%% Starts the supervisor
%%
%% @spec start_link() -> {ok, Pid} | ignore | {error, Error}
%% @end
%%--------------------------------------------------------------------
start_link() ->
      supervisor:start_link({local, ?SERVER}, ?MODULE, []).

%%%===================================================================
%%% Supervisor callbacks
%%%===================================================================

%%--------------------------------------------------------------------
```

```
%% @private
%% @doc
%% Whenever a supervisor is started using supervisor:start_link/[2,3],
%% this function is called by the new process to find out about
%% restart strategy, maximum restart frequency and child
%% specifications.
%%
%% @spec init(Args) -> {ok, {SupFlags, [ChildSpec]}} |
%% ignore |
%% {error, Reason}
%% @end
%%----------------------------------------------------------------------
init([]) ->
      RestartStrategy = one_for_one,
      MaxRestarts = 1000,
      MaxSecondsBetweenRestarts = 3600,
      SupFlags = {RestartStrategy, MaxRestarts, MaxSecondsBetweenRestarts},
      Restart = permanent,
      Shutdown = 2000,
      Type = worker,
      AChild = {'AName', {'AModule', start_link, []},
      Restart, Shutdown, Type, ['AModule']},
      {ok, {SupFlags, [AChild]}}.

%%%==========================================================================
%%% Internal functions
%%%==========================================================================
```

Пример B-3.Шаблон application

```
%%%----------------------------------------------------------------------
%%% @author $author
%%% @copyright (C) $year, $company
%%% @doc
%%%
%%% @end
%%% Created : $fulldate
%%%----------------------------------------------------------------------
-module($basename).

-behaviour(application).

%% Application callbacks
-export([start/2, stop/1]).

%%%==========================================================================
%%% Application callbacks
%%%==========================================================================

%%----------------------------------------------------------------------
%% @private
%% @doc
%% This function is called whenever an application is started using
%% application:start/[1,2], and should start the processes of the
%% application. If the application is structured according to the OTP
%% design principles as a supervision tree, this means starting the
%% top supervisor of the tree.
%%
%% @spec start(StartType, StartArgs) -> {ok, Pid} |
%% {ok, Pid, State} |
%% {error, Reason}
%% StartType = normal | {takeover, Node} | {failover, Node}
%% StartArgs = term()
%% @end
```

```
%%--------------------------------------------------------------------
start(_StartType, _StartArgs) ->
      case 'TopSupervisor':start_link() of
            {ok, Pid} ->
                  {ok, Pid};
            Error ->
                  Error
      end.

%%--------------------------------------------------------------------
%% @private
%% @doc
%% This function is called whenever an application has stopped. It
%% is intended to be the opposite of Module:start/2 and should do
%% any necessary cleaning up. The return value is ignored.
%%
%% @spec stop(State) -> void()
%% @end
%%--------------------------------------------------------------------
stop(_State) ->
      ok.

%%%===================================================================
%%% Internal functions
%%%===================================================================
```