

«Отличное введение в Elixir для людей с практическим складом ума. Авторы сразу переходят к сути и дают прекрасный обзор возможностей языка Elixir, достаточно глубокий, чтобы познакомить читателя с языком и вызвать желание попробовать его.»

— Андреа Леопарди, программист, член команды разработчиков Elixir

Красивый, мощный и компактный, язык программирования Elixir отлично подходит для изучения функционального программирования, и это практическое руководство покажет вам, насколько широкими возможностями он обладает. Авторы расскажут, как Elixir сочетает в себе надежность языка функционального программирования Erlang с подходом, свойственным языку Ruby, а также мощную поддержку макросов для метапрограммирования.

Познакомившись с сопоставлением с образцом, программированием процессов и другими идеями, вы поймете, почему на Elixir так просто писать параллельные, надежные и отказоустойчивые программы, которые легко масштабируются как вверх, так и вниз!

С этой книгой вы:

- освоите IEx — интерфейс командной строки Elixir;
- исследуете основные структуры данных в Elixir;
- познакомитесь с атомами, с механизмом сопоставления с образцом и ограничениями: основными конструкциями структурирования программ;
- изучите приемы обработки данных в Elixir с применением рекурсии, строк, списков и функций высшего порядка;
- узнаете, как создавать процессы и пересылать сообщения между ними;
- освоите сохранение и управление структурированными данными, хранящимися в Erlang Term Storage (ETS) и базе данных Mnesia;
- научитесь создавать отказоустойчивые приложения с Open Telecom Platform.

Симон Сенлорен (Simon St. Laurent), администратор контента на сайте *LinkedIn Learning* и прежде старший редактор в издательстве O'Reilly Media. Автор и соавтор нескольких книг, включая: «*Introducing Elixir*», «*Introducing Erlang*», «*Learning Rails 3*» и «*XML Pocket Reference. 3rd Edition*».

Дэвид Эйзенберг (J. David Eisenberg), программист и преподаватель. Разработал курсы изучения HTML, CSS, JavaScript и Linux. Читает курс «Компьютерные информационные технологии» в колледже *Evergreen Valley*. Дэвид также написал несколько книг, включая «*Etudes for Erlang*».

Интернет-магазин:

www.dmkpress.com

Книга — почтой:

orders@aliants-kniga.ru

Оптовая продажа:

«Альянс-книга»

тел. (499) 782-38-89

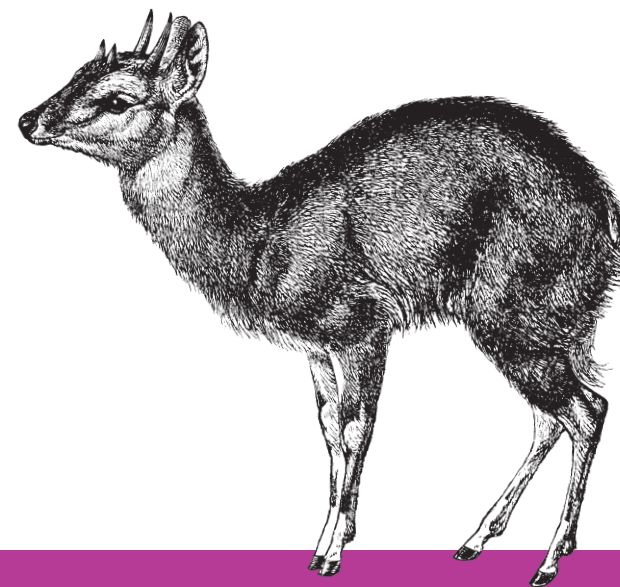
books@aliants-kniga.ru



ISBN 978-5-97060-518-9



9 785970 605189 >



Введение в Elixir

Симон Сенлорен
Дэвид Эйзенберг



Симон Сенлорен и Дэвид Эйзенберг

Введение в Elixir

Simon St. Laurent and J. David Eisenberg

INTRODUCING ELIXIR

Getting Started in Functional Programming

Second Edition

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Симон Сенлорен и Дэвид Эйзенберг

ВВЕДЕНИЕ В ELIXIR

**Введение в функциональное
программирование**



Москва, 2017

УДК 004.438Elixir
ББК 32.973.3
С31

Сенлорен С., Эйзенберг Д.
С31 Введение в Elixir: введение в функциональное программирование / пер. с англ. А. Н. Киселева – М.: ДМК Пресс, 2017. – 262 с.: ил.

ISBN 978-5-97060-518-9

Красивый, мощный и компактный, язык программирования Elixir отлично подходит для изучения функционального программирования, и это практическое руководство покажет, насколько широкими возможностями он обладает. В книге рассказано, как Elixir сочетает в себе надежность языка функционального программирования Erlang с подходом, свойственным языку Ruby, а также мощную поддержку макросов для метапрограммирования.

В итоге вы поймете, почему на Elixir так просто писать параллельные, надежные и отказоустойчивые программы, которые легко масштабируются как вверх, так и вниз!

УДК 004.438Elixir
ББК 32.973.3

Authorized Russian translation of the English edition of *Introducing Elixir*, 2nd Edition, ISBN 9781491956779 © 2017 Simon St. Laurent, J. David Eisenberg. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-491-95677-9 (анг.)
ISBN 978-5-97060-518-9 (рус.)

© 2017 Simon St. Laurent and J. David Eisenberg
© Оформление, издание, ДМК Пресс, 2017

Содержание

Предисловие	10
Глава 1. Устраиваемся поудобнее	19
Установка	19
Установка Erlang	19
Установка Elixir	20
Запуск	21
Первые шаги	22
Навигация по тексту и истории команд	23
Навигация по файлам	23
Сделаем что-нибудь	24
Вызов функций.....	25
Числа в Elixir	26
Работа с переменными в оболочке.....	28
Глава 2. Функции и модули	31
Игры с fn.....	31
И &	33
Определение модулей	34
От модулей к свободным функциям	38
Деление кода на модули	38
Комбинирование функций с помощью оператора конвейера.....	40
Импортирование функций	41
Значения по умолчанию для аргументов	42
Документирование кода	43
Документирование функций.....	44
Документирование модулей	46
Глава 3. Атомы, кортежи и сопоставление с образцом	48
Атомы	48
Сопоставление с образцом и атомы.....	49
Логические атомы.....	50

Ограничители.....	52
Обозначайте подчеркиванием все, что не важно	55
Структуры данных: кортежи.....	57
Сопоставление с образцом и кортежи.....	58
Обработка кортежей.....	59

Глава 4. Логика и рекурсия 62

Логика внутри функций.....	62
Конструкция case.....	62
Конструкция cond.....	65
if или else	66
Присваивание значений переменным в конструкциях case и if.....	68
Самый желательный побочный эффект: IO.puts.....	69
Простая рекурсия.....	71
Обратный отсчет	71
Прямой отсчет.....	73
Рекурсия с возвратом значения.....	74

Глава 5. Взаимодействие с человеком 79

Строки.....	79
Многострочные строки	82
Юникод.....	82
Списки символов	83
Строковые метки.....	84
Запрос информации у пользователя.....	85
Ввод символов	85
Чтение строк текста	86

Глава 6. Списки 91

Основы списков.....	91
Деление списков на головы и хвосты	93
Обработка содержимого списков	94
Создание списка из головы и хвоста.....	96
Смешивание списков и кортежей	99
Создание списка списков	99

Глава 7. Пары имя/значение 103

Списки ключей	103
Списки кортежей с несколькими ключами	105

Словари	106
От списков к отображениям.....	107
Создание отображений	108
Изменение отображений.....	108
Чтение отображений.....	109
От отображений к структурам	109
Объявление структур.....	110
Создание и чтение экземпляров структур.....	110
Использование структур в сопоставлениях с образцом	111
Использование структур в функциях.....	112
Добавление поведения в структуры.....	114
Расширение существующих протоколов.....	116

Глава 8. Функции высшего порядка

и генераторы списков	118
Простые функции высшего порядка.....	118
Создание новых списков с помощью функций высшего порядка.....	121
Получение информации о списке	121
Обработка элементов списка с помощью функций.....	122
Фильтрация значений в списках.....	123
За пределами возможностей генераторов списков	124
Проверка списков	124
Разбиение списков	125
Свертка списков	126

Глава 9. Процессы

Интерактивная оболочка – это процесс.....	129
Порождение процессов из модулей.....	132
Легковесные процессы.....	135
Регистрация процесса.....	136
Когда процесс останавливается	137
Взаимодействие между процессами.....	138
Наблюдение за процессами	141
Наблюдение за движением сообщений между процессами.....	143
Разрыв и установка связей между процессами	145

Глава 10. Исключения, ошибки и отладка.....

Виды ошибок	152
Восстановление работоспособности после ошибок времени выполнения	153

Журналирование результатов выполнения и ошибок	156
Трассировка сообщений.....	157
Трассировка вызовов функций	159

Глава 11. Статический анализ, спецификации типов и тестирование

Статический анализ	161
Спецификации типов	164
Модульные тесты	167
Настройка тестов	170
Встраивание тестов в документацию	172

Глава 12. Хранение структурированных данных

Записи: структурирование данных до появления структур	173
Определение записей	174
Создание и чтение записей.....	175
Использование записей в функциях.....	177
Сохранение данных в долговременном хранилище Erlang.....	179
Создание и заполнение таблицы	181
Простые запросы.....	187
Изменение значений.....	187
Таблицы ETS и процессы.....	188
Следующие шаги.....	190
Хранение записей в Mnesia	191
Настройка базы данных Mnesia.....	191
Создание таблиц.....	192
Чтение данных	196

Глава 13. Основы OTP

Создание служб с помощью GenServer	199
Простой супервизор.....	204
Упаковка приложения с помощью Mix.....	209

Глава 14. Расширение языка Elixir с помощью макросов

Функции и макросы.....	213
Простой макрос	214
Создание новой логики.....	216
Программное создание функций.....	217
Когда (не) следует использовать макросы.....	219

Глава 15. Phoenix	221
Установка базовых компонентов фреймворка.....	221
Структура простого Phoenix-приложения.....	224
Представление страницы	224
Маршрутизация	225
Простой контроллер	227
Простое представление.....	228
Вычисления.....	230
Продвижение Elixir	237
 Приложение А. Каталог элементов языка Elixir	239
Команды интерактивной оболочки	239
Зарезервированные слова	240
Операторы	241
Ограничители.....	243
Часто используемые функции.....	244
 Приложение В. Создание документации с помощью ExDoc	247
Использование ExDoc вместе с Mix.....	247
 Предметный указатель	251
Об авторах	259
Колофон	260

Предисловие

Язык Elixir предлагает разработчикам мощь функционального и конкурентного языка программирования Erlang, но с более дружелюбным синтаксисом, обширным набором библиотек и средствами метапрограммирования. Программный код на языке Elixir компилируется в байт-код Erlang, что позволяет объединять его с программным кодом, написанным на Erlang, и использовать инструменты Erlang. Однако, несмотря на общий фундамент, Elixir создает ощущение совершенно другого языка, более похожего, пожалуй, на Ruby, чем на Prolog, предшественника Erlang.

Книга «Введение в Elixir» познакомит вас с этим мощным языком.



Это издание книги «Введение в Elixir» охватывает версию 1.4 языка. Мы будем обновлять и дополнять его по мере развития Elixir. Если вы найдете ошибки или опечатки, сообщите нам, воспользовавшись системой учета ошибок.

Кому адресована эта книга

Эта книга адресована в основном тем, кто имеет опыт программирования на других языках, но ищет чего-то нового. Может быть, вас интересует весьма практичная и распределенная модель программирования, обладающая преимуществами масштабирования и высокой надежности. Может быть, вы хотите увидеть и понять, что такое «функциональное программирование». Или просто желаете познакомиться с чем-то новым без конкретной цели.

Мы полагаем, что функциональный стиль программирования стал намного более доступным гораздо раньше, чем вы научились программировать с использованием других парадигм. Однако знакомство с Elixir – а иногда даже просто его установка – требует немалых навыков. Поэтому, если вы вообще не имеете опыта программирования, добро пожаловать на борт, но имейте в виду, что по пути вам встретится немало сложностей.

Для кого эта книга не предназначена

Эта книга не предназначена тем, кто торопится достичь поставленной цели.

Если вы уже знакомы с языком Elixir, эта книга, скорее всего, вам не нужна, только если вы не ищете неторопливого, постепенного введения.

Если вы уже знакомы с языком Erlang, эта книга поможет вам увидеть отличия, но, скорее всего, ключевые понятия, обсуждаемые здесь, вам уже знакомы.

Если вы уже знакомы с функциональными языками программирования, это введение покажется вам слишком медленным. Если вы заскучаете, читая эту книгу, смело откладывайте ее в сторону и обратитесь к какой-нибудь другой книге или к электронной документации, рассказывающей о языке в более быстром темпе.

Что даст вам эта книга

С помощью этой книги вы научитесь писать простые программы на Elixir. Узнаете, почему Elixir упрощает создание надежных и легко масштабируемых программ. Здесь вы познакомитесь с терминологией, благодаря чему сможете читать другие ресурсы с информацией о языке Elixir, которые требуют определенного опыта для их понимания.

В теоретическом отношении вы познакомитесь с функциональным программированием. Узнаете, как проектировать программы, основанные на передаче сообщений и рекурсии, и как создавать программы для обработки потоков данных.

Самое важное: эта книга откроет ворота в мир разработки конкурентных программ. Даже притом, что это введение познакомит вас лишь с самыми основами открытой телекоммуникационной платформы (Open Telecom Platform, OTP), знание этих основ поможет вам открыть для себя новый, удивительный мир. После овладения синтаксисом и приемами структурирования программ на языке Elixir вы сможете приступить к созданию надежных и масштабируемых приложений, и это потребует от вас намного меньше усилий, чем при использовании других подходов!

Как читать эту книгу

Эта книга рассказывает историю появления языка Elixir. Наибольшую пользу вы получите, если будете читать все по порядку, хотя бы первый раз, однако вы можете читать выборочно и возвращаться к неп прочитанным главам, если почувствуете, что это необходимо.

Сначала вы установите Elixir и опробуете его интерактивную оболочку IEx. В этой оболочке вы будете проводить много времени, поэтому постарайтесь овладеть ею в полной мере, чтобы чувствовать себя в ней уютно. Затем вы научитесь загружать программный код в оболочку, чтобы получить возможность писать программы за ее пределами, и узнаете, как вызывать и смешивать этот код.

Вы детально изучите числа, потому что это самые простые представители базовых структур в Elixir. Затем вы познакомитесь с атомами, конструкциями сопоставления с образцом и ограничителями – наиболее распространенными элементами программ. После этого вы перейдете к строкам, спискам и рекурсии, лежащей в основе Elixir. Преодолев несколько тысяч рекурсий вниз и назад, вы перейдете к процессам, ключевым элементам Elixir, которые используют модель обмена сообщениями для поддержки конкуренции и отказоустойчивости.

После освоения основ вы займетесь более детальным изучением приемов отладки и хранения данных, а затем познакомитесь с комплектом инструментов, которые пригодятся вам в разработке программ на языке Elixir: открытой телекоммуникационной платформой (Open Telecom Platform, ОТП) языка Erlang, которая охватывает не только телефоны.

Наконец, вы освоите макросы, средства, придающие языку Elixir огромную гибкость и позволяющие расширять его.

Некоторые предпочитают изучать языки программирования, знакомясь с его словарем, списками операторов, управляющих структур и типов данных. Эти списки также приводятся в книге, но они находятся не в основном тексте книги, а в приложении А.

Главное, что должна дать вам эта книга, – научить вас программированию на языке Elixir. Если вы этого не получили, дайте нам знать!

Другие ресурсы

Эта книга, возможно, не лучший для вас способ изучить язык Elixir. Все зависит от того, чему вы хотите научиться и как вы привыкли осваивать что-то новое. Если вы ищете скоростное введение в язык, обратите внимание на книгу «Programming Elixir» (Pragmatic Publishers), в которой вы быстро познакомитесь с отличительными чер-

тами Elixir. Похожее быстрое и более глубокое введение в Elixir вы также найдете в книгах «Elixir in Action» (Manning) и «The Little Elixir & OTP Guidebook» (Manning). Книга «Metaprogramming Elixir» (Pragmatic Publishers) исследует ключевые возможности языка, а книга «Programming Phoenix» (Pragmatic Publishers) подробно описывает мощный фреймворк на основе Elixir.

Если вам понравятся неспешность и основательность этой книги и у вас появится желание продолжить обучение в том же духе, обратите внимание на книгу «Études for Elixir» (O'Reilly). В ней вы найдете описания коротких программ, которые сможете сами написать на Elixir, более сложные, чем примеры в данной книге. Кроме того, ее структура очень похожа на структуру данной книги.

Существует также множество книг, но посвященных не Elixir, а языку Erlang. Однако мы надеемся, что в ближайшее время появится множество новых книг об Elixir. На сайте проекта Elixir вы найдете много руководств, документацию и ссылки на другие ресурсы.

Если в изучении Elixir вами движет желание вырваться из наезженной колеи, обязательно прочитайте книгу Брюса Тейта (Bruce Tate) «Seven Languages in Seven Weeks» (Pragmatic Publishers)¹, в которой дается обзор языков Ruby, Io, Prolog, Scala, Erlang, Clojure и Haskell. Описание Erlang в ней занимает всего 30 с небольшим страниц, но вполне возможно, что это именно то, что вам нужно.

Книги о языке Erlang тоже могут помочь понять, что делает Elixir таким замечательным.

Простое введение в Erlang, во многом похожее на эту книгу, вы найдете в книге «Introducing Erlang» (O'Reilly), которая познакомит вас с Erlang и функциональным программированием.

Попробуйте прочитать электронную версию книги (которую можно также приобрести в бумажном варианте, выпущенном издательством No Starch Books) с остроумными и забавными иллюстрациями «Learn You Some Erlang for Great Good»²!

Также можно порекомендовать классические книги об Erlang с похожими названиями: «Programming Erlang» (Pragmatic Publishers) и «Erlang Programming»³ (O'Reilly). Они очень похожи и во

¹ Тейт Б. Семь языков за семь недель. М.: ДМК Пресс, 2014. 384 с. – Прим. перев.

² Хеберт Ф. Изучай Erlang во имя добра! М.: ДМК Пресс, 2014. 688 с. – Прим. перев.

³ Чезарини Ф. Программирование в Erlang. М.: ДМК Пресс, 2015. 488 с. – Прим. перев.

многим переключаются друг с другом, и обе могут послужить прекрасными руководствами, если эта книга покажется вам слишком медлительной или если вам требуется руководство, которое можно использовать как справочник. «Erlang Programming» имеет более практическую направленность, тогда как «Programming Erlang» описывает множество деталей, касающихся настройки окружения для программирования на Erlang.

Более продвинутое руководство «Erlang and OTP in Action» (Manning) начинается с краткого 72-страничного введения в Erlang и затем обстоятельно описывает применение открытой телекоммуникационной платформы, фреймворка на языке Erlang для создания легко обслуживаемых и расширяемых конкурентных приложений. «Designing for Scalability with Erlang/OTP»¹ (O'Reilly) исследует применение OTP и Erlang для решения задач, которые выглядят чрезвычайно сложными в других окружениях.

Посетите также веб-сайт проекта Erlang², где вы найдете последние обновления, документацию, примеры и многое другое.

Elixir изменит вас

Прежде чем вы продолжите, мы хотим предупредить вас, что использование языка Elixir может кардинально изменить ваши взгляды на программирование. Обычное для этого языка сочетание функционального кода, обработки, ориентированной на процессы, и распределенных вычислений в первый момент может показаться чем-то чужеродным. Однако знакомство с Elixir может изменить ваши подходы к решению задач (причем они могут существенно отличаться от подходов, принятых в языке Erlang) и даже усложнить возврат к другим языкам, окружениям и культурам программирования.

Типографские соглашения

В этой книге приняты следующие типографские соглашения:

Курсив

Используется для обозначения новых терминов, адресов электронной почты, имен файлов и расширений имен файлов.

¹ Чезарини Ф., Виноски С. Проектирование масштабируемых систем с помощью Erlang/OTP. М.: ДМК Пресс, 2017. 486 с. – *Прим. перев.*

² <http://www.erlang.org/>.

Моноширинный шрифт

Применяется для оформления листингов программ и программных элементов внутри обычного текста, таких как имена переменных и функций, инструкций и ключевых слов.

Моноширинный жирный

Обозначает команды или другой текст, который должен вводиться пользователем.

Моноширинный курсив

Обозначает текст, который должен замещаться фактическими значениями, вводимыми пользователем или определяемыми из контекста.



Так обозначаются советы, предложения и примечания общего характера.



Так обозначаются предупреждения и предостережения.



Так выделяются советы и предложения.

Использование программного кода примеров

В этой книге приводится множество примеров, цель которых – познакомить вас с основными идеями. Примеры сделаны максимально короткими, чтобы вам проще было видеть отличия между ними. Вы можете свободно использовать примеры программного кода из этой книги, как посчитаете нужным, но вы не сможете взять код из этой книги и составить из него законченное приложение (если только не испытываете особую нежность к вычислениям скоростей свободно падающих объектов). Тем не менее примеры помогут вам выяснить шаги, которые необходимо проделать для создания приложения. Код примеров можно загрузить из репозитория GitHub¹. (Он также доступен по ссылке **Examples** в каталоге издательства².)

Данная книга призвана оказать вам помощь в решении ваших задач. Вам не нужно обращаться в издательство за разрешением, если

¹ <https://github.com/simonstl/introducing-elixir>.

² <http://shop.oreilly.com/product/0636920050612.do>.

вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабатываете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам необходимо получить разрешение от издательства O'Reilly. Если вы отвечаете на вопросы, цитируя данную книгу или примеры из нее, получение разрешения не требуется. Но при включении существенных объемов программного кода примеров из этой книги в вашу документацию вам необходимо будет получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: «Introducing Elixir, Second Edition, by Simon St. Laurent and J. David Eisenberg (O'Reilly). Copyright 2017 Simon St. Laurent and J. David Eisenberg, 978-1-491-95677-9».

За получением разрешения на использование значительных объемов программного кода примеров из этой книги обращайтесь по адресу permissions@oreilly.com.

Дополнительная помощь

Мы очень надеемся, что вам понравится эта книга и вы сможете самостоятельно освоить ее, мы также надеемся, что вы сможете оказать помощь другим читателям, изучающим Elixir. Оказать такую помощь можно множеством способов:

- если вы обнаружите техническую неточность, невнятное описание или что-то, что можно улучшить, сообщите об этом через систему регистрации ошибок (<http://bit.ly/elixir-2e-errata>);
- если вам понравилась (или не понравилась) книга, пожалуйста, оставьте свой отзыв. Наиболее предпочтительным местом для этого являются сайт Amazon.com и страница книги на сайте издательства O'Reilly. Подробное описание, что понравилось и что не понравилось, пригодится другим читателям и нам;
- если вы считаете, что можете рассказать о языке Elixir намного больше, поделитесь своими знаниями, например в веб или в своей книге, на учебных курсах или каким-то другим способом, наиболее удобным для вас.

Мы будем исправлять опечатки и ошибки в книге и постараемся своевременно откликаться на проблемы, поднимаемые в отзывах.

Даже после завершения книги мы, скорее всего, будем продолжать добавлять в нее абзацы и разделы. Если вы приобрели электронную версию книги, вы будете получать эти дополнения бесплатно, по крайней мере до того момента, пока не выйдет новое издание. Мы не обещаем скорого выхода нового издания, если только в мире Elixir не произойдут существенные перемены.

Надеемся, что эта книга увлечет вас настолько, что вам захочется поделиться своими знаниями.

Используйте ее во благо

У каждого из нас есть свое понимание, что такое «во благо». Тем не менее, пожалуйста, постарайтесь использовать мощь Elixir для создания проектов, улучшающих наш мир или, по крайней мере, не ухудшающих его.

Как связаться с нами

С вопросами и предложениями, касающимися этой книги, обращайтесь в издательство:

O'Reilly Media, Inc.

1005 Gravenstein Highway North Sebastopol, CA 95472

800-998-9938 (США или Канада)

707-829-0515 (международный и местный)

707-829-0104 (факс)

Список опечаток, файлы с примерами и другую дополнительную информацию вы найдете на странице книги http://bit.ly/introducing_elixir_2e.

Свои пожелания и вопросы технического характера отправляйте по адресу bookquestions@oreilly.com.

Дополнительную информацию о книгах, обсуждения, конференции и новости вы найдете на веб-сайте издательства: <http://www.oreilly.com>.

Ищите нас на Facebook: <http://facebook.com/oreilly>.

Следуйте за нами в Твиттере: <http://twitter.com/oreillymedia>.

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

Благодарности

Вокруг Elixir сплотилось удивительно дружелюбное сообщество, открытое для вопросов и предложений по широкому кругу проблем. Мы были счастливы получать ответы на свои вопросы и наслаждались общением с членами сообщества, искренне старавшимися объяснить сложные проблемы на примерах кода.

Хосе Валима (José Valim) оказывал нам неоценимую помощь по всем аспектам в книге. Наш товарищ и конкурент Дейв Томас (Dave Thomas) подтвердил, что да, Elixir жив и мир ждет его. Со стороны Erlang Франческо Чезарини (Francesco Cesarini) приветствовал наше желание рассказать об этом новом, братском языке. Обозреватели: Андреа Леопарди (Andrea Leopardi), Мэтт Миллз (Matt Mills), Байбек Пандей (Bibek Pandey), Алексей Шолик (Alexei Sholik), Дэвид Лоренцетти (David Lorenzetti), Бенгт Клеберг (Bengt Kleberg), Мистраль Контрастин (Mistral Contrastin), Оджи де Блайк мл. (Augie De Blieck Jr.), Эри ван Вингерден (Arie van Wingerden), Элиас Каррильо (Elias Carrillo) и Николас (Nicholas) – помогли нам в поиске ошибок.

Меган Бланшетт (Meghan Blanchette) помогала в работе над первым изданием. Морин Спенсер (Maureen Spencer) и Хезер Шерер (Heather Scherer) оказали ценную помощь с подготовкой второго издания.

Кроме того, участие Дж. Дэвида Эйзенберга (J. David Eisenberg) в проекте не раз спасало Симона Сенлорена (Simon St. Laurent)!

Спасибо также Симону, который дал Дэвиду первый и приятный опыт работы над книгой.

Глава 1

Устраиваемся поудобнее

Изучение Elixir удобнее всего начинать в интерактивной оболочке Interactive Elixir (IEx). Этот интерфейс командной строки – лучшее средство, чтобы выяснить, что возможно и что невозможно в Elixir. Он поможет избавиться от головной боли в будущем, поэтому устраивайтесь поудобнее!

Установка

Elixir действует поверх Erlang, поэтому для начала необходимо установить Erlang в системе, а затем Elixir.

Установка Erlang

Установка Erlang в Windows выполняется очень просто (<http://www.erlang.org/downloads>). Загрузите двоичный файл для Windows, запустите установку – и все! Если вы – храбрый новичок, только начинающий осваивать программирование, для вас такой вариант станет лучшим выбором.

Пользователи Linux или macOS могут загрузить исходный код и скомпилировать его. В macOS придется распаковать архив с исходным кодом, затем перейти в каталог, созданный в процессе распаковки, и выполнить последовательность команд: `./configure`, `make` и `sudo make install`. Однако эта простая последовательность действий увенчается успехом, только если в системе установлены все необходимые инструменты и библиотеки, в противном случае есть риск получить множество малопонятных ошибок. В частности,

в новейших версиях XCode компания Apple использует компилятор LLVM вместо GCC, соответственно, велика вероятность отсутствия компилятора GCC в новейших версиях системы macOS, необходимого для сборки Erlang.

Вы можете спокойно игнорировать ошибки, вызванные отсутствием пакета FOP, который Erlang использует для создания документации в формате PDF – вы сможете загрузить ее отдельно. Кроме того, в новейших версиях macOS в конце может появиться ошибка, сообщающая, что wxWidgets не работает в 64-битных версиях системы. Ее тоже можно безопасно игнорировать, пока.

Если по каким-то причинам компиляция из исходного кода невозможна или нежелательна, на сайте Erlang Solutions (<https://www.erlang-solutions.com/resources/download.html>) можно найти большое количество установочных пакетов для наиболее распространенных версий дистрибутивов. Кроме того, многие диспетчеры пакетов (Debian, Ubuntu, MacPorts, Homebrew и др.) включают Erlang. Это может быть не самая последняя версия, но наличие любой действующей версии Erlang намного лучше ее отсутствия. Обычно распространители дистрибутивов стремятся включать в свои репозитории самые свежие версии пакетов, поэтому если у вас не заладилось с компиляцией, обратите внимание на диспетчеры пакетов.



Во многих системах Erlang все чаще включается в установку по умолчанию, в том числе в Ubuntu, в основном благодаря широкому распространению CouchDB.

Установка Elixir

Завершив установку Erlang, загрузите скомпилированную версию Elixir (<https://github.com/elixir-lang/elixir/releases>) или исходный код (<https://github.com/elixir-lang/elixir/tags>) из репозитория GitHub. Некоторые диспетчеры пакетов уже включают Elixir, в том числе Homebrew. Примеры, представленные в этой книге, должны без проблем выполняться в Elixir 1.3.0.

Затем настройте переменную окружения PATH, добавив в нее путь к каталогу *elixir/bin*.

Инструкции по настройке Elixir можно найти в руководстве, по адресу: <http://elixir-lang.org/getting-started/introduction.html>¹.

¹ На русском языке: <http://elixir-lang.ru/install>. – Прим. перев.

Запуск

Перейдите в командную строку (например, откройте окно терминала) и введите команду **mix new first_app**. Она запустит инструмент Mix (<https://hexdocs.pm/mix/Mix.html>), который «реализует задачи создания, компиляции и тестирования проектов на языке Elixir, управления их зависимостями и многие другие». В данном случае введенная команда создаст новый, пустой проект в каталоге с именем *first_app*:

```
$ mix new first_app
* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/first_app.ex
* creating test
* creating test/test_helper.exs
* creating test/first_app_test.exs
```

Your Mix project was created successfully.

You can use "mix" to compile it, test it, and more:

```
cd first_app
mix test
```

Run "mix help" for more commands.

```
$
```



Не забудьте добавить путь к каталогу с выполняемым файлом Elixir в переменную окружения PATH, чтобы инструмент Mix смог найти его.

Не будем тратить время на компиляцию и тестирование пустого проекта, а сразу перейдем в каталог *first_app* и запустим оболочку IEEx, выполнив следующие команды:

```
$ cd first_app
$ iex -S mix
```

Вы увидите на экране примерно следующие строки, возможно, с курсором в строке приглашения *iex(1)>*. Обратите внимание, что везде, где это необходимо, длинные строки переформатированы, чтобы уместить их по ширине страницы:

```
$ cd first_app
[david@localhost first_app]$ iex -S mix
Erlang/OTP 19 [erts-8.0] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe]
```

```
[kernel-poll:false]
Compiling 1 file (.ex)
Generated first_app app
Interactive Elixir (1.3.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
```

Вот мы и в Elixir! (Первая строка с упоминанием Erlang подчеркивает, что Elixir действует внутри Erlang. Но не беспокойтесь об этом!)

Первые шаги

Прежде чем окунуться в захватывающий процесс программирования на языке Elixir, всегда имеет смысл научиться выходить из оболочки. Нажатие комбинации **Ctrl+C** приведет к появлению меню. Если в этом меню ввести «a», IEx завершит работу, и вы увидите приглашение к вводу командной оболочки системы, в которой вы запускали IEx:

```
iex(1)>
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
a
$
```

В оболочке iex можно также попросить вывести справку (после повторного запуска), введя h() или просто h¹:

```
iex(1)> h()
# IEx.Helpers
```

IEx.Helpers

Добро пожаловать в интерактивную оболочку Interactive Elixir. Сейчас вы смотрите документацию для модуля IEx.Helpers, включающего множество вспомогательных функций, делающих работу с интерактивной оболочкой Elixir более удобной.

Это сообщение сгенерировано вызовом вспомогательной функции h(), которую обычно обозначают как h/0 (поскольку она принимает 0 аргументов).

Функцию h можно использовать для вызова документации к любому модулю Elixir или функции:

¹ Здесь и далее в книге справочный текст приводится в переводе на русский язык, но имейте в виду, что в действительности вся документация в Elixir на английском языке. — *Прим. перев.*

```
| h Enum
| h Enum.map
| h Enum.reverse/1
```

Для исследования любого значения, созданного в интерактивной оболочке, можно также использовать функцию `i`:

```
| i "hello"
```

Существует большое количество других вспомогательных функций:

```
...
:ok
```

Что, собственно, произошло? Мы выполнили команду `iex`, вызвали вспомогательную функцию `h()`, которая вывела перед вами некоторую справочную информацию. Она выводит на экран большой объем текста и завершает вывод, возвращая `:ok`.

Навигация по тексту и истории команд

Занявшись исследованием интерактивной оболочки, вы обнаружите, что своим действием она во многом напоминает другие командные оболочки или Emacs. Клавиши со стрелками влево и вправо перемещают курсор назад и вперед по редактируемой строке. Некоторые комбинации клавиш повторяют действие этих же комбинаций в текстовом редакторе Emacs. Комбинация **Ctrl+A** перенесет курсор в начало строки, а **Ctrl+E** – в конец. Если вы по ошибке ввели два символа не в той последовательности, нажмите **Ctrl+T**, чтобы поменять их местами.

Кроме того, после ввода закрывающей круглой, квадратной или фигурной скобки будет подсвечена парная ей открывающая скобка.

Клавиши со стрелками вверх и вниз выполняют навигацию по истории команд, упрощая их повторный запуск. На конкретную команду в истории можно также сослаться как `v(N)`, где `N` – номер строки.

Навигация по файлам

Оболочка IEEx поддерживает некоторые простые операции с файловой системой, потому что вам может потребоваться переключаться между файлами, составляющими вашу программу. Команды имеют те же имена, что и команды Unix, но записываются как вызовы функций. Первоначально текущим рабочим каталогом оболочки IEEx является каталог, в котором вы ее запустили. Узнать путь к этому каталогу можно с помощью `pwd()`:


```
iex(1)> pwd()  
/Users/elixir/code/first_app  
:ok
```

Для перехода в другой каталог используйте функцию `cd()`, но вы обязательно должны заключить ее аргумент в двойные кавычки:

```
iex(2)> cd ".."  
/Users/elixir/code  
:ok  
iex(3)> cd "first_app"  
/Users/elixir/code/first_app  
:ok
```

Получить список с содержимым каталога можно с помощью команды `ls()`, которая в случае вызова без аргумента вернет список содержимого текущего каталога, а с аргументом – список содержимого указанного каталога.

Сделаем что-нибудь

Проще всего начать игру с Elixir – использовать интерактивную оболочку как калькулятор. В отличие от обычной командной строки, она позволяет вводить арифметические выражения и получать результаты:

```
iex(1)> 2 + 2  
4  
iex(2)> 27 - 14  
13  
iex(3)> 35 * 42023943  
1470838005  
iex(4)> 4 * (3 + 5)  
32  
iex(5)> 200 / 15  
13.333333333333334
```

Первые три оператора – сложение (+), вычитание (-) и умножение (*) – действуют как обычно при работе с целыми или вещественными числами. Круглые скобки позволяют изменять порядок выполнения операторов, как показано в строке 4. (Порядок выполнения операций приводится в приложении А.) Четвертый оператор, /, возвращает вещественный результат (число с дробной частью), как показано в строке 5. Операторы необязательно окружать пробелами, но они делают код более читаемым.

Вызов функций

Функции – это блоки логики, которые принимают аргументы и возвращают значение. Проще всего начать исследование с математических функций. Например, если требуется получить целочисленный результат деления (целочисленных аргументов), используйте функцию `div()` вместо оператора `/` и функцию `rem()`, чтобы получить остаток, как показано в строках 6 и 7:

```
iex(6)> div(200,15)
13
iex(7)> rem(200, 15)
5
iex(8)> rem 200, 15
5
```



Строка 8 демонстрирует особенность синтаксиса Elixir: необязательность заключения аргументов в круглые скобки. Если вы считаете, что круглые скобки улучшают читаемость кода, используйте их. Если ввод круглых скобок кажется вам лишней работой, просто не вводите их. Elixir интерпретирует пробел, следующий за именем функции, как открывающую круглую скобку, автоматически закрывающуюся в конце строки. Когда подобный стиль приводит к неожиданным результатам, Elixir может попросить вас «не вставлять пробелы между именем функции и открывающей круглой скобкой».

Elixir допускает передачу целых чисел вместо вещественных, но вещественные числа не всегда можно использовать вместо целых. Если вам потребуется преобразовать вещественное число в целое, используйте встроенную функцию `round()`:

```
iex(9)> round 200/15
13
```

Функция `round()` отбрасывает дробную часть числа. Если дробная часть больше или равна `.5`, целая часть увеличивается на 1, округляя число вверх. Если вам требуется просто отбросить дробную часть, оставив целую, используйте функцию `trunc()`.

Сослаться на предыдущую команду можно по номеру строки, с помощью функции `v()`. Например:

```
iex(10)> 4*v(9)
52
```

Команда в строке 9 вернула результат 13, соответственно, выражение `4*13` дает в результате 52.



Если вам интересно, для ссылки на предыдущие команды можно использовать отрицательные числа. `v(-1)` вернет предыдущий результат, `v(-2)` – результат, полученный перед предыдущим, и так далее.

Для выполнения более сложных вычислений Elixir позволяет использовать модуль `math` языка Erlang, содержащий классический набор функций, поддерживаемых научным калькулятором. Эти функции возвращают вещественные значения. Константа «пи» доступна как функция `:math.pi()`. Доступны также тригонометрические, логарифмические, экспоненциальные функции, функция квадратного корня и (исключая Windows) даже функция гауссовой ошибки. (Тригонометрические функции принимают аргументы в радианах, а не в градусах, поэтому при необходимости вам придется выполнять соответствующие преобразования.) При использовании этих функций вам придется также вводить префикс `:math.`, но это не слишком высокая плата за удобство.

Например, ниже показано, как получить синус от нуля радиан:

```
iex(11)> :math.sin(0)
0.0
```

Обратите внимание, что в результате получилось число `0.0`, а не просто `0`. Это указывает, что результат является вещественным числом. (И да, вы можете то же самое вычисление выполнить командой `:math.sin 0` без круглых скобок.)

Ниже показано, как получить косинус от π и 2π радиан:

```
iex(12)> :math.cos(:math.pi())
-1.0
iex(13)> :math.cos(2 * :math.pi())
1.0
```

Чтобы вычислить 2 в степени 16 , используйте функцию:

```
iex(14)> :math.pow(2,16)
65536.0
```

Полный перечень функций, имеющих в модуле `math` и доступных в Elixir, вы найдете в приложении А.

Числа в Elixir

Elixir распознает два вида чисел: целые и вещественные (их также часто называют числами с плавающей точкой). В отличие от целых чисел, не имеющих дробной части, вещественные числа включают

десятичную точку и некоторое значение правее ее – дробную часть. 1 – это целое число, а 1.0 – вещественное.

Однако в действительности все немного сложнее. Elixir хранит целые и вещественные числа в совершенно разных форматах. Elixir позволяет хранить огромные целые числа, но какими большими или маленькими они бы не были, они всегда точны. Вам не придется волноваться об их точности.

Вещественные числа, напротив, хотя и охватывают широкий диапазон значений, но имеют ограниченную точность. В Elixir используется 64-битное представление IEEE 754-1985 «с двойной точностью». То есть этот формат хранит примерно 15 цифр плюс экспоненту. Этот формат может представлять огромные значения – число 10 в степени от -308 до 308 – но из-за ограниченного количества цифр результат иногда может отличаться от точного значения, что особенно ярко проявляется при выполнении сравнения:

[illegible]

Как видите, некоторые цифры отсекаются, и в полном представлении число включает экспоненту.

При вводе вещественных чисел всегда указывайте хотя бы одну цифру слева от десятичной точки, даже если это цифра «ноль». Иначе Elixir сообщит о синтаксической ошибке – он не понимает, чего вы хотите:

[illegible]

Вещественные числа можно также вводить в экспоненциальной нотации:

```
iex(4)> 2.923e127
2.923e127
iex(5)> 7.6345435e-231
7.6345435e-231
```

Из-за недостаточной точности применение вещественных чисел может приводить к аномальным результатам. Например, синус нуля равен нулю, и синус числа π также равен нулю. Но, производя вычисления в Elixir, вы не получите нулевого результата при использовании вещественной аппроксимации π :

```
iex(6)> :math.sin(0)
0.0
iex(7)> :math.sin(:math.pi())
1.2246467991473532e-16
```

Если бы в Elixir использовалось более точное представление числа π и вычисления с вещественными числами выполнялись бы с большей точностью, результат в строке 7 был бы ближе к нулю.

Для финансовых вычислений лучше всего использовать целые числа. Используйте наименьшую единицу измерения – например, копейки – и помните, что одна копейка – это 1/100 рубля. (Финансовые сделки могут выполняться в более мелких единицах, но и в этом случае желательно использовать целые числа и при необходимости применять известный множитель.) Однако для более сложных вычислений вполне можно использовать вещественные числа, просто помните, что результат получается не совсем точным.

Elixir поддерживает операции с целыми числами еще в нескольких системах счисления, кроме десятичной. Например, если понадобится, двоичное значение 1010111 можно записать как:

```
iex(8)> 0b01010111
87
```

Elixir вернет значение в виде десятичного числа. Аналогично можно указать шестнадцатеричное (в системе счисления с основанием 16) число, используя префикс `x` вместо `b`:

```
iex(9)> 0xcafe
51966
```

Чтобы ввести отрицательное число, просто добавьте знак минус (-) перед ним. Этот прием можно использовать с целыми числами, а также шестнадцатеричными, двоичными и вещественными числами:

```
iex(10)> -1234
-1234
iex(11)> -0xcafe
-51966
iex(12)> -2.045234324e6
-2045234.324
```

Работа с переменными в оболочке

Функция `v()` позволяет ссылаться на результаты предыдущих вычислений, но это не самый удобный способ хранения числовых ре-

зультатов, и к тому же функция `v()` работает только в интерактивной оболочке – это не универсальный механизм. Намного разумнее хранить значения в памяти, давая им текстовые имена, то есть создавая переменные.

Имена переменных в Elixir начинаются с буквы в нижнем регистре или символа подчеркивания. Обычные переменные начинаются с буквы в нижнем регистре, а «необязательные» – с символа подчеркивания (см. раздел «Обозначайте подчеркиванием все, что не важно» в главе 3). Но пока давайте придерживаться обычных переменных. Синтаксис присваивания значений переменным должен быть вам знаком по школьному курсу алгебры или по другим языкам программирования, например:

```
iex(1)> n=1
1
```

Чтобы узнать значение переменной, просто введите ее имя:

```
iex(2)> n
1
```

Elixir, в отличие от многих других функциональных языков программирования (включая Erlang), позволяет присваивать переменным новые значения:

```
iex(3)> n=2
2
iex(4)> n=n+1
3
```

Elixir сначала выполняет выражение справа, затем оператор `=` сопоставляет результат с левой стороной. Он присвоит переменной `n` новое значение, если попросить его об этом, и даже будет использовать прежнее значение `n` в выражении справа, чтобы получить новое значение. Инstrukция `n=n+1` означает: «присвоить значение выражения `n+1`, которое равно 3, переменной `n`».

В операции присваивания все вычисления должны производиться справа от знака «равно». Даже если известно, что `m` имеет значение 6, Elixir не сможет выполнить инструкцию `2*m = 3*4`:

```
iex(5)> 2*m=3*4
** (CompileError) iex:12: illegal pattern
```

Оболочка IEx будет помнить переменные, пока вы не попросите ее забыть их.

Кроме того, в одну строку можно включить несколько инструкций, разделив их точкой с запятой (;). Синтаксически она действует подобно окончанию строки:

```
iex(6)> distance = 20; gravity = 9.8
9.8
iex(7)> distance
20
iex(8)> gravity
9.8
```

IEx сообщит только значение последней инструкции, но, как можно видеть в строках 7 и 8, все переменные получили соответствующие значения.

Если в оболочке накопилось много ненужного, вызовите `clear`. Эта функция просто очистит экран.

Прежде чем перейти к следующей главе, где рассказывается о модулях и функциях, поиграйте немного в IEx. Полученный опыт, даже такой простой, поможет вам увереннее двигаться вперед. Попробуйте использовать переменные и посмотрите, что происходит с большими целыми числами. Elixir прекрасно справляется с поддержкой даже гигантских чисел. Попробуйте смешивать в вычислениях целые и вещественные числа и посмотрите, что из этого получится. Все это должно быть для вас не очень сложно.

Глава 2

ФУНКЦИИ И МОДУЛИ

Подобно большинству языков программирования, Elixir позволяет определять функции, помогающие представлять повторяющиеся вычисления. Функции Elixir могут быть очень сложными, но в своей основе это очень простые элементы программ.

Игры с `fn`

Создать функцию в оболочке IEx можно с помощью ключевого слова `fn`. Например, ниже показано, как создать функцию, вычисляющую скорость падающего объекта на определенном расстоянии от точки сброса, выраженном в метрах:

```
iex(1)> fall_velocity = fn (distance) -> :math.sqrt(2 * 9.8 * distance) end
#Function<6.6.111823515/1 in :erl_eval.expr/5>
```

Эта инструкция свяжет переменную `fall_velocity` с функцией, принимающей аргумент `distance`. (Круглые скобки, окружающие аргумент, можно опустить.) Функция возвращает (нам больше нравится читать стрелку `->` как «производит») квадратный корень из удвоенного произведения постоянной ускорения свободного падения для Земли ($9,8 \text{ м/с}^2$) на расстояние (в метрах). Определение функции завершается ключевым словом `end`.

Возвращаемое значение в интерактивной оболочке, `Function<6.6.111823515/1 in :erl_eval.expr/5>`, само по себе не имеет особого смысла, но оно сообщает, что функция создана и никаких ошибок не возникло. (Точный формат этого возвращаемого значения зависит от версии Elixir, поэтому у вас оно может выглядеть несколько иначе.)

Связывание функции с переменной `fall_velocity` позволяет использовать эту переменную для вычисления скорости падающего объекта на Земле:


```
iex(2)> fall_velocity.(20)
19.79898987322333
iex(3)> fall_velocity.(200)
62.609903369994115
11
iex(4)> fall_velocity.(2000)
197.9898987322333
```

Если вам понадобится определить более сложную функцию, вы можете разместить ее в нескольких строках. Оболочка IEх сохранит определение открытым, пока не будет введено слово `end`, как в следующем примере:

```
iex(5)> f = fn (distance) ->
...(5)> :math.sqrt(2 * 9.8 * distance)
...(5)> end
#Function<6.54118792/1 in :erl_eval.expr/5>
iex(6)> f.(20)
19.79898987322333
```

Это может пригодиться, когда потребуется включить в функцию несколько инструкций.

Чтобы вызвать функцию, хранящуюся в переменной, между именем переменной и аргументом требуется добавлять точку. Это не нужно при использовании функций, объявленных в модулях, о которых пойдет речь ниже в этой главе.

Если вам потребуется преобразовать метры в секунду в мили в час, просто создайте еще одну функцию. Вы можете скопировать и вставить в вызовы прежние результаты (как мы сделали это здесь) или ввести свои, более короткие числа:

```
iex(7)> mps_to_mph = fn mps -> 2.23693629 * mps end
#Function<6.54118792/1 in :erl_eval.expr/5>
iex(8)> mps_to_mph.(19.79898987322333)
44.289078952755766
iex(9)> mps_to_mph.(62.609903369994115)
140.05436496173314
iex(10)> mps_to_mph.(197.9898987322333)
442.89078952755773
```

Да, пожалуй, лучше не падать с 2000-метровой высоты.

Предпочитаете выражать скорость в километрах в час?

```
iex(11)> mps_to_kph = fn(mps) -> 3.6 * mps end
#Function<6.54118792/1 in :erl_eval.expr/5>
iex(12)> mps_to_kph.(19.79898987322333)
71.27636354360399
```

```
iex(13)> mps_to_kph.(62.60990336999411)
225.39565213197878
iex(14)> mps_to_kph.(197.9898987322333)
712.76363543604
```

Можно сразу получать результаты в желаемых единицах измерения, вкладывая вызовы функций друг в друга:

```
iex(15)> mps_to_kph.(fall_velocity.(2000))
712.76363543604
```

Скорость получается впечатляющей. Впрочем, в действительности она будет существенно ниже из-за сопротивления воздуха.

Функции удобны для выполнения повторяющихся вычислений, но определять их в интерактивной оболочке не очень практично, так как они исчезнут с завершением сеанса. Функции такого вида называют *анонимными функциями*, потому что сами функции не имеют имен. (Имя переменной не является именем функции.) Анонимные функции удобно использовать для передачи функций в виде аргументов другим функциям и для возврата функций в виде результатов. Однако внутри модулей можно определять именованные функции, доступные откуда угодно.

ИГ

Elixir поддерживает более компактный способ определения анонимных функций с использованием *&*, *оператора захвата*. В этом случае вместо `fn` применяется *&*; а вместо именованных параметров – параметры вида *&1*, *&2* и т. д.

Выше мы определили `fall_velocity` как:

```
iex(1)> fall_velocity= fn (distance) -> :math.sqrt(2 * 9.8 * distance) end
#Function<erl_eval.6.111823515>
```

Если такое определение кажется вам избыточно подробным, используйте *&*:

```
iex(2)> fall_velocity= &(:math.sqrt(2 * 9.8 * &1))
#Function<6.17052888/1 in :erl_eval.expr/5>
iex(3)> fall_velocity.(20)
19.79898987322333
```

Сначала, вероятно, проще использовать именованные параметры, но по мере накопления опыта оператор захвата будет выглядеть все привлекательнее. Его ценность станет очевидной, когда вы начнете создавать более сложные функции, как те, что показаны в главе 8.

Определение модулей

Большинство программ на Elixir, кроме самых простых вычислений, как в примерах выше, определяет свои функции в компилируемых модулях, а не в интерактивной оболочке. Модули лучше подходят для хранения программ, и они дают возможность более эффективно хранить, скрывать, экспортировать ваш код и управлять им.

Каждый модуль должен находиться в отдельном файле с расширением *.ex*. (Можно, конечно, поместить в один файл несколько модулей, но старайтесь сохранять их максимально простыми.) Модуль должен сохраняться в файле с именем *name_of_module.ex*, где *name_of_module* – версия имени модуля, указанного внутри файла, состоящая только из символов нижнего регистра. Инструмент Mix поможет вам в этом. Мы создали пример 2.1, который можно найти в архиве примеров (<https://github.com/simonstl/introducing-elixir/tree/master/2nd-edition>), в папке *ch02/ex1-drop*, введя в командной строке:

```
$ mix new ch02/ex1-drop --app drop
```

ch02/ex1-drop – это путь к каталогу, который требуется создать. Mix выберет *drop* в качестве имени приложения и создаст в подкаталоге *lib* файл с именем *drop.ex*. Вот как выглядит содержимое этого файла сразу после создания:

```
defmodule Drop do
end
```

Затем мы вставили в него функции, что были определены выше, и получили результат, показанный в примере 2.1.

Пример 2.1 ❖ Модуль для вычисления и преобразования скорости падения

```
defmodule Drop do
  def fall_velocity(distance) do
    :math.sqrt(2 * 9.8 * distance)
  end

  def mps_to_mph(mps) do
    2.23693629 * mps
  end

  def mps_to_kph(mps) do
    3.6 * mps
  end
end
```

Инструкция *defmodule* включает функции, экспортируемые модулем. Она принимает имя модуля – на этот раз с буквы в верхнем ре-

гистре – и содержит определения функций. На этот раз определения начинаются с ключевого слова `def` и имеют немного иную структуру, в сравнении с использованием ключевого слова `fn`, и их не требуется присваивать переменным.



Определения функций внутри модуля должны использовать более длинный синтаксис `do...end` вместо `->`. Если функция достаточно короткая, ее определение можно поместить в одну строку, например:

```
def mps_to_mph(mps), do: 2.23693629 * mps
```

Такой «однострочный» синтаксис часто будет встречаться вам в коде, написанном другими, но для единообразия и удобочитаемости мы рекомендуем придерживаться полного синтаксиса `do...end`.

Любые функции, объявленные с помощью ключевого слова `def`, доступны за пределами модуля и могут вызываться другим кодом. Если требуется ограничить доступность каких-то функций рамками модуля, используйте ключевое слово `defp` вместо `def`.

Обычно весь код в модулях сосредоточен внутри функций.

Но как включить модуль в работу?

Пришло время заняться компиляцией кода на Elixir. Интерактивная оболочка позволяет скомпилировать модуль и тут же использовать его. Если вы создали проект с помощью Mix (как это было сделано здесь), запустите IElixir в каталоге проекта:

```
$ iex -S mix
Erlang/OTP 19 [erts-8.0] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe]
[kernel-poll:false]

Compiling 1 file (.ex)
Generated drop app
Interactive Elixir (1.3.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
```

Mix создаст каталог `_build`; если заглянуть в `_build/dev/lib/drop/ebin`, можно увидеть файл с именем `Elixir.Drop.beam`.

Если вы редактируете файл с исходным кодом в редакторе и желаете повторно скомпилировать его, не выходя из интерактивной оболочки IElixir, выполните команду `recompile`. В следующем примере мы добавили пустую строку в файл `drop.ex`, чтобы показать, как выполняется повторная компиляция:

```
iex(2)> recompile
Compiling 1 file (.ex)
:ok
```

Если исходный код не изменялся, вы увидите ответ `:noop` (по operation – пустая операция) вместо `:ok`.

Если вы не пользуетесь инструментом Mix, чего мы настоятельно не рекомендуем, вам придется запустить IElixir в каталоге с файлом, который требуется скомпилировать, и затем вызвать команду с именем файла:

```
iex(1)> c("drop.ex")
[Drop]
```

Если теперь заглянуть в каталог с файлом *drop.ex*, можно увидеть новый файл с именем *Elixir.Drop.beam*.

После компиляции модуля появляется возможность использовать функции из него:

```
iex(2)> Drop.fall_velocity(20)
19.79898987322333
iex(3)> Drop.mps_to_mph(Drop.fall_velocity(20))
44.289078952755766
```

Они работают точно так же, как функции, которые были определены ранее, но теперь вы можете закрыть интерактивную оболочку, вновь запустить ее и повторно использовать скомпилированные функции:

```
iex(4)>
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
a
$ iex -S mix
Erlang/OTP 19 [erts-8.0] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe]
Defining Modules | 15
[kernel-poll:false]

Interactive Elixir (1.3.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> Drop.mps_to_mph(Drop.fall_velocity(20))
44.289078952755766
```

В процессе программирования на Elixir основное время занимают разработка функций в модулях и их соединение в большие программы.



Если вы не помните, в каком каталоге находитесь, выполните команду `pwd()`. Если вам потребуется перейти в другой каталог, выполните команду `cd(pathname)`. В следующем примере команда `cd("../")` вернет вас в первоначальный каталог:

```
iex(1)> pwd()
/Users/elixir/code/ch02/ex1-drop
```

```

iex(2)> cd("lib")
/Users/elixir/code/ch02/ex1-drop/lib
iex(3)> pwd()
/Users/elixir/code/ch02/ex1-drop/lib
iex(4)> cd("../")
/Users/elixir/code/ch02/ex1-drop
iex(5)>

```



Если вдруг обнаружится, что, работая в IEx, вы все время повторяете одни и те же последовательности команд, вы можете воспользоваться командой `c`, чтобы скомпилировать такие группы команд IEx. Для этого поместите последовательности команд в файлы с расширением `.exs` (от Elixir script). Если вызвать функцию `c()` и передать ей такой файл, Elixir выполнит все команды, находящиеся в файле.

Компиляция Elixir и система времени выполнения Erlang

Интерактивная оболочка Elixir интерпретирует каждую введенную команду, независимо от того, выполнялась она прежде или нет. Когда вы требуете от Elixir скомпилировать файл, содержимое файла преобразуется в некоторый код, не нуждающийся в повторной интерпретации, что существенно увеличивает эффективность выполнения.

Таким «некоторым кодом», в случае с Elixir, является файл Erlang BEAM. Он содержит код, который может выполняться процессором BEAM, ключевым элементом системы времени выполнения Erlang (Erlang Runtime System, ERTS). Аббревиатура BEAM расшифровывается как Bogdan's Erlang Abstract Machine (абстрактная машина Богдана для Erlang). Это виртуальная машина, интерпретирующая оптимизированный код BEAM. Может показаться, что такое решение не так эффективно, как традиционная компиляция в машинный код, выполняемый компьютером непосредственно, но оно близко напоминает другие виртуальные машины. (Наиболее типичными представителями виртуальных машин являются Oracle Java Virtual Machine, или JVM, и Microsoft .NET Framework.)

Виртуальная машина Erlang оптимизирует некоторые важные аспекты, облегчая конструирование надежных и масштабируемых приложений. Встроенный планировщик берет на себя все хлопоты по распределению операций между несколькими процессорами на одном компьютере. Вам не придется думать о том, сколько процессоров может занять ваше приложение, — вы просто пишете независимые процессы, а Erlang автоматически распределяет их между процессорами. Erlang также немного иначе организует ввод и вывод, избегая стиля, опирающегося на установление соединений, который блокирует другие операции. И кроме того, виртуальная машина использует свою стратегию сборки мусора, хорошо согласующуюся с ее стилем работы, сокращая до минимума паузы в ра-

боте приложения. (Механизм сборки мусора освобождает память, которая была нужна процессу в одной точке и стала не нужна в другой.)

Когда вы начнете создавать свои программы на Elixir, вы будете распространять их в виде комплектов скомпилированных файлов BEAM. Однако вам не придется собирать каждый из них в интерактивной оболочке, как мы делали это выше. Команда `elixirc` позволит вам скомпилировать все файлы Elixir сразу и объединить компиляцию со сборкой, а команда `elixir` с файлами `.exs` даст вам возможность выполнить код на Elixir как сценарий за пределами оболочки IEx.

От модулей к свободным функциям

Если вам нравится стиль определения функций с помощью `fn`, но хочется обеспечить сохранность кода в модулях, где его проще отлаживать, вы можете объединить все самое лучшее из двух миров, используя оператор захвата `&` для ссылки на уже определенную функцию. Ниже показано, как извлечь функцию с единственным аргументом, используя синтаксис: *Имя_модуля.имя_функции/арность*. Арность (*arity*) — это количество аргументов, принимаемых функцией, в случае с `Drop.fall_velocity` это 1:

```
iex(1)> fun=&Drop.fall_velocity/1
&Drop.fall_velocity/1
iex(2)> fun.(20)
19.79898987322333
```

То же самое можно проделать внутри модуля. Если в ссылке указывается функция из этого же модуля, префикс с именем модуля можно опустить. То есть в данном случае часть `Drop.` можно выбросить из определения и оставить только `&fall_velocity/1`.

Деление кода на модули

Модуль `Drop` включает два разных вида функций. Функция `fall_velocity()` полностью соответствует имени модуля, `Drop`, реализуя вычисления, опирающиеся на высоту, с которой сбрасывается объект. Однако функции `mps_to_mph` и `mps_to_kph` не имеют прямого отношения к свободному падению. Это универсальные функции преобразования единиц измерений, которые могут пригодиться также в других контекстах и в действительности должны быть объявлены в отдельном модуле. Примеры 2.2 и 2.3, оба в папке *ch02/ex2-split*, демонстрируют, как этого добиться.

Пример 2.2 ❖ Модуль вычисления скорости свободно падающего объекта

```
defmodule Drop do
  def fall_velocity(distance) do
    :math.sqrt(2 * 9.8 * distance)
  end
end
```

Пример 2.3 ❖ Модуль функций преобразования скоростей

```
defmodule Convert do
  def mps_to_mph(mps) do
    2.23693629 * mps
  end

  def mps_to_kph(mps) do
    3.6 * mps
  end
end
```

Скомпилируем их, и после этого разделенные функции станут доступными для использования:

```
$ iex -S mix
Erlang/OTP 19 [erts-8.0] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe]
[kernel-poll:false]

Compiling 2 files (.ex)
Generated drop app
Interactive Elixir (1.3.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> Drop.fall_velocity(20)
19.79898987322333
iex(2)> Convert.mps_to_mph(Drop.fall_velocity(20))
44.289078952755766
```

Теперь код читается намного проще, но как быть, если эти функции понадобятся вызывать в третьем модуле? Модули, использующие функции из других модулей, должны явно заявлять об этом. Пример 2.4 (*ch02/ex3-combined*) демонстрирует модуль, использующий функции из обоих модулей, Drop и Convert.

Пример 2.4 ❖ Модуль, объединяющий логику из модулей Drop и Convert

```
defmodule Combined do
  def height_to_mph(meters) do
    Convert.mps_to_mph(Drop.fall_velocity(meters))
  end
end
```

Такой прием работает, только если модуль Combined имеет доступ к модулям Convert и Drop, что гарантируется по умолчанию, если они

находятся в том же каталоге, причем то же правило действует в оболочке IEx.

Функция в модуле `Combined` помогает сократить ввод с клавиатуры:

```
$iex -S mix
Erlang/OTP 19 [erts-8.0] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe]
[kernel-poll:false]

Compiling 3 files (.ex)
Generated combined app
Interactive Elixir (1.3.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> Combined.height_to_mph(20)
44.289078952755766
```



Имеющие опыт программирования на Erlang, вероятно, привыкли к «толстостенным» модулям, содержимое которых становится доступно только при явном использовании директив `-export` и `-import`. Elixir избрал другой путь, в нем доступно все, что имеется снаружи, кроме функций, явно объявленных приватными с помощью инструкции `defp`.

Комбинирование функций с помощью оператора конвейера

Существует еще один способ комбинирования функций: с помощью оператора `|>`, который называют *оператором конвейера* (pipe operator). Оператор конвейера, который иногда называют *прямым конвейером* (pipe forward), позволяет передать выражение в первом аргументе следующей функции. Пример 2.5 (*ch02/ex4-pipe*) демонстрирует, как использовать этот оператор.

Пример 2.5 ❖ Использование оператора конвейера

```
defmodule Combined do
  def height_to_mph(meters) do
    Drop.fall_velocity(meters) |> Convert.mps_to_mph
  end
end
```

Обратите внимание, что в сравнении с примером 2.4 порядок следования функций в коде изменился: теперь `Drop.fall_velocity(meters)` располагается перед `Convert.mps_to_mph`. Если читать оператор `|>` как «передается в», тогда логика работы может показаться более ясной. Вы можете выстроить в цепочку сразу несколько функций преобразования, которые иначе пришлось бы глубоко вкладывать в вызовы друг друга, и получить код, который читается проще, как мы полагаем.



Оператор конвейера передает в следующую функцию только один результат, в ее первом аргументе. Если справа от оператора конвейера вам понадобится использовать функцию, принимающую несколько параметров, просто укажите остальные параметры, как если бы первый уже находился на своем месте.

Импортирование функций

Пока вы указываете полное квалифицированное имя функции, Elixir автоматически будет стараться отыскать ее в коде. Однако если вы постоянно используете функции из какого-то определенного модуля, можно сэкономить на нажатиях клавиш, формально импортировав его.

Пример 2.6 (*ch02/ex5-import*) демонстрирует простейший случай использования директивы `import`, которая включает все функции (и макросы, которых, впрочем, там пока нет) из модуля `Convert`.

Пример 2.6 ❖ Модуль, объединяющий логику `Drop` и `Convert` и импортирующий `Convert`

```
defmodule Combined do
  import Convert

  def height_to_mph(meters) do
    mps_to_mph(Drop.fall_velocity(meters))
  end
end
```

Инструкция `import Convert` сообщает Elixir, что все функции и макросы (кроме тех, чьи имена начинаются с символа подчеркивания) из модуля `Convert` должны быть доступны без использования имени этого модуля в качестве префикса.

Операция импортирования модуля Erlang, как показано в примере 2.7, выглядит почти так же, за исключением того, что имя модуля предваряется двоеточием и само имя начинается с буквы в нижнем регистре.

Пример 2.7 ❖ Импортирование Erlang-модуля `math`

```
defmodule Drop do
  import :math

  def fall_velocity(distance) do
    sqrt(2 * 9.8 * distance)
  end
end
```

Импортирование модулей целиком может вызывать конфликты имен функций, уже используемых в вашем собственном модуле. Чтобы этого избежать, Elixir позволяет с помощью аргумента `only` указать, какие функции импортировать. Например, ниже показано, как импортировать только функцию `sqrt` с аргументом `1`:

```
defmodule Drop do
  import :math, only: [sqrt: 1]

  def fall_velocity(distance) do
    sqrt(2 * 9.8 * distance)
  end
end
```

Если требуется импортировать модуль для использования в единственной функции, директиву `import` можно вставить непосредственно в инструкцию `def` или `defp` с определением этой функции. Ее действие не будет распространяться за пределы функции:

```
defmodule Drop do
  def fall_velocity(distance) do
    import :math, only: [sqrt: 1]

    sqrt(2 * 9.8 * distance)
  end
end
```



Если понадобится импортировать все функции, кроме нескольких, можно передать директиве `import` аргумент `except`:

```
import :math, except: [sin: 1, cos: 1]
```

Используйте `import` с осторожностью. Да, эта директива помогает сократить объем вводимого кода, но при ее применении становится сложнее определить, из какого модуля вызывается та или иная функция.

Значения по умолчанию для аргументов

Если потребуется организовать вычисления с участием не только Земли, но и других астрономических тел (и мы будем делать это на протяжении следующих глав), можно создать функцию `fall_velocity/2`, принимающую два параметра, высоту и постоянную ускорения свободного падения:

```
defmodule Drop do
  def fall_velocity(distance, gravity) do
    :math.sqrt(2 * gravity * distance)
  end
end
```

С этой функцией можно вычислять скорость свободно падающих тел на Земле, для которой ускорение свободного падения равно 9,8, и на Луне, где ускорение свободного падения равно 1,6:

```
$ iex -S mix
Erlang/OTP 19 [erts-8.0] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe]
[kernel-poll:false]

Compiling 1 file (.ex)
Generated drop app
Interactive Elixir (1.3.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> Drop.fall_velocity(20, 9.8)
19.79898987322333
iex(2)> Drop.fall_velocity(20, 1.6)
8.0
```

Если предполагается выполнять расчеты в основном для Земли, в таких случаях Elixir позволяет указать значение по умолчанию для параметра ускорения свободного падения после пары обратных слешей, как показано в примере 2.8 (*ch02/ex6-defaults*).

Пример 2.8 ❖ Функция с параметром, имеющим значение по умолчанию

```
defmodule Drop do
  def fall_velocity(distance, gravity \\ 9.8) do
    :math.sqrt(2 * gravity * distance)
  end
end
```

Теперь для вычисления скорости свободно падающего объекта на Земле можно передать функции только первый аргумент, а на других планетах – оба аргумента:

```
iex(3)> recompile
Compiling 1 file (.ex)
:ok
iex(4)> Drop.fall_velocity(20)
19.79898987322333
iex(5)> Drop.fall_velocity(20, 1.6)
8.0
```

Документирование кода

Программы могут прекрасно работать и без документации. Однако работать с такими проектами очень тяжело.

Многие программисты думают, что они пишут код, взглянув на который любой сможет сразу понять, что он делает, но в действитель-

ности это не совсем так, и даже совсем не так. Программный код, чуть более сложный, чем в предыдущих примерах, может выглядеть непонятным и запутанным для других разработчиков. Даже вам, обдумавшему и написавшему свой код, спустя какое-то время этот код будет казаться таинственным и непостижимым.

Создателям Elixir хорошо знакомы эти проблемы, поэтому они прекрасно понимают важность и необходимость хорошей документации.

Самый простой способ включить описание в код – добавить в него комментарий. Комментарии могут начинаться с символа `#` и простираются до конца строки. Некоторые комментарии занимают всю строку, другие – лишь часть ее, в самом конце. Пример 2.9 демонстрирует обе разновидности комментариев.

Пример 2.9 ❖ Комментарии в коде

```
# вспомогательные функции!
```

```
defmodule Combined do
```

```
  def height_to_mph(meters) do # принимает метры, возвращает мили в час  
    Convert.mps_to_mph(Drop.fall_velocity(meters))
```

```
  end
```

```
end
```

Компилятор Elixir просто игнорирует текст, следующий за символом `#`, до конца строки, но люди, исследующие код, смогут прочитать его.

Неформальные комментарии, как в примере выше, очень полезны, и разработчики часто включают такие комментарии, помогающие им следить за тем, что они делают, пока пишут код. Такие комментарии могут помогать или не помогать понять код другим разработчикам или даже вам самим, вернувшимся к своему же коду после долгого перерыва. Более формальные комментарии требуют больше времени и внимания, чем вам хотелось бы выделить в пылу программирования, но они заставляют вас спросить себя: кто мог бы заняться исследованием вашего кода в будущем и что он хотел бы узнать.

Поддержка документирования в Elixir не только позволяет включать в код банальные комментарии, но и предлагает набор инструментов для создания документации, которую можно читать в оболочке IEx или отдельно, с помощью веб-браузера.

Документирование функций

Модуль `Drop` содержит одну функцию: `fall_velocity/1`. Возможно, вы знаете, что она принимает высоту в метрах и возвращает скорость

в метрах в секунду для объекта, сброшенного в вакууме на Земле, но код ничего этого не говорит. В примере 2.10 показано, как исправить этот недостаток с помощью тега `@doc`.

Пример 2.10 ❖ Документирование функции вычисления скорости свободно падающего объекта

```
defmodule Drop do
  @doc """
  Вычисляет скорость свободно падающего объекта на Земле, как
  если бы он падал в вакууме (то есть без учета сопротивления
  воздуха). Параметр distance определяет высоту в метрах, с которой
  падает объект, а возвращаемое значение выражает скорость в метрах
  в секунду.
  """

  def fall_velocity(distance) do
    import :math, only: [sqrt: 1]

    sqrt(2 * 9.8 * distance)
  end
end
```

Если после компиляции этого модуля вызвать функцию `h()` в IEx, она сообщит полезную информацию о функции:

```
iex(1)> recompile
Compiling 1 file (.ex)
:ok
iex(2)> h Drop.fall_velocity
def fall_velocity(distance)
```

Вычисляет скорость свободно падающего объекта на Земле, как если бы он падал в вакууме (то есть без учета сопротивления воздуха). Параметр `distance` определяет высоту в метрах, с которой падает объект, а возвращаемое значение выражает скорость в метрах в секунду.

Так намного лучше, но что, если пользователь передаст в аргументе функции строку «twenty» («двадцать») вместо числа 20? Поскольку Elixir является языком с динамической типизацией, исходный код никак не подчеркивает, что значением для `distance` должно быть число, иначе функция вернет ошибку.

Добавить эту информацию можно с помощью тега `@spec`. Это может показаться немного странным, потому что подобное описание в какой-то степени дублирует объявление функции. В данном случае описание выглядит просто, как показано в примере 2.11.

Пример 2.11 ❖ Документирование функции вычисления скорости свободно падающего объекта

```
defmodule Drop do
```

```
  @doc """
```

```
  Вычисляет скорость свободно падающего объекта на Земле, как
  если бы он падал в вакууме (то есть без учета сопротивления
  воздуха). Параметр distance определяет высоту в метрах, с которой
  падает объект, а возвращаемое значение выражает скорость в метрах
  в секунду.
  """
```

```
  @spec fall_velocity(number()) :: float()
```

```
  def fall_velocity(distance) do
```

```
    import :math, only: [sqrt: 1]
```

```
    sqrt(2 * 9.8 * distance)
```

```
  end
```

```
end
```

Эта спецификация сообщает, что функция может принимать любые числа (`number()`) – целые или вещественные, – но результат всегда является вещественным (`float()`) числом.

Теперь вы сможете воспользоваться функцией `s()`, чтобы увидеть информацию о типах аргументов и возвращаемого значения функции:

```
iex(1)> s(Drop.fall_velocity)
```

```
@spec fall_velocity(number()) :: float()
```

Можно также попробовать выполнить вызов `s(Drop)`, чтобы увидеть все спецификации, присутствующие в модуле `Drop`.



В этой главе демонстрируются только типы `float()` и `number()` (объединяющий типы `integer()` и `float()`). Полный перечень типов приводится в приложении А.

Документирование модулей

До сих пор в этой главе демонстрировались очень простые модули, но в них уже достаточно кода, чтобы начать документирование модулей, как показано в файлах, в папке `ch02/ex7-docs`. В примере 2.12 приводится модуль `Drop` с дополнительной информацией о его версии (`@vsn`), авторе и причинах создания.

Пример 2.12 ❖ Документирование модуля с функцией вычисления скорости свободно падающего объекта

```
defmodule Drop do
  @moduledoc """
  Функции вычисления скорости свободно падающего объекта в вакууме.

  *Introducing Elixir*, Second Edition, O'Reilly Media, Inc., 2017.
  Copyright 2017 by Simon St.Laurent and J. David Eisenberg.
  """

  @vsn 0.1

  @doc """
  Вычисляет скорость свободно падающего объекта на Земле, как
  если бы он падал в вакууме (то есть без учета сопротивления
  воздуха). Параметр distance определяет высоту в метрах, с которой
  падает объект, а возвращаемое значение выражает скорость в метрах
  в секунду.
  """

  @spec fall_velocity(number()) :: number()

  def fall_velocity(distance) do
    import :math, only: [sqrt: 1]

    sqrt(2 * 9.8 * distance)
  end
end
```

Теперь с помощью функции `h` вы сможете узнать больше о модуле:

```
iex(1)> h Drop
```

```
Drop
```

Функции вычисления скорости свободно падающего объекта в вакууме.

Introducing Elixir, Second Edition, O'Reilly Media, Inc., 2017.
Copyright 2017 by Simon St.Laurent and J. David Eisenberg.

Наличие хорошей документации пригодится всем, кто примет-ся за чтение вашего кода (и даже *вам*, когда вам придется вернуться к своему коду спустя несколько месяцев). Эту документацию можно даже использовать для создания веб-страниц, описывающих модули и функции. Для этого вам понадобится инструмент ExDoc. ExDoc распознает разметку Markdown (<http://daringfireball.net/projects/markdown/>), благодаря чему у вас есть возможность выделять текст и включать в описание, кроме всего прочего, списки и ссылки. Дополнительная информация по использованию ExDoc приводится в приложении В.

Глава 3

Атомы, кортежи и сопоставление с образцом

Программы на Elixir в основном состоят из наборов запросов и средств их обработки. Elixir включает инструменты, упрощающие эффективную обработку таких запросов и помогающие писать читаемый код (по крайней мере, для программистов), который действует достаточно быстро.

Атомы

Атомы – ключевой компонент Elixir. Технически это всего лишь еще один тип данных, но они оказывают существенное влияние на стиль программирования на языке Elixir.

Обычно атомы – это фрагменты текста, начинающиеся с двоеточия, такие как `:ok`, `:earth` и `:Today`. Они могут также включать символы подчеркивания (`_`) и «коммерческого at» (`@`), например: `:this_is_a_short_sentence` или `:me@home`. Если вам понадобится использовать в атомах пробелы, заключите текст после двоеточия в одиночные или двойные кавычки, например: `:'Today is a good day'`. Атомы в форме одного слова, состоящего только из букв нижнего регистра, обычно читаются проще.

Значениями атомов являются они сами:

```
iex(1)> :test  
:test
```

В самих атомах нет ничего интересного. Интересными они становятся в сочетании с другими типами и приемами сопоставления с образцом, где они помогают конструировать простые, но мощные логические структуры.

Сопоставление с образцом и атомы

В примерах из главы 2 Elixir использовал механизм сопоставления с образцом, но там все было очень просто. Имя функции интерпретировалось как ключ, который может изменяться, а после добавления числового аргумента Elixir понимал, что вы имеете в виду. Однако сопоставление с образцом в Elixir обладает более широкими возможностями, позволяя выполнять сопоставление с аргументами и именами функций.

Например, представьте, что требуется реализовать вычисление скорости свободно падающего объекта не только на Земле, где постоянная ускорения свободного падения равна $9,8 \text{ м/с}^2$, но также на Луне ($1,6 \text{ м/с}^2$) и на Марсе ($3,71 \text{ м/с}^2$). В примере 3.1 (*ch03/ex1-atoms*) демонстрируется один из вариантов решения этой задачи.

Пример 3.1 ❖ Сопоставление с атомами и именами функций

```
defmodule Drop do
```

```
  def fall_velocity(:earth, distance) do
    :math.sqrt(2 * 9.8 * distance)
  end
```

```
  def fall_velocity(:moon, distance) do
    :math.sqrt(2 * 1.6 * distance)
  end
```

```
  def fall_velocity(:mars, distance) do
    :math.sqrt(2 * 3.71 * distance)
  end
```

```
end
```

Этот код выглядит так, как будто функция `fall_velocity` определена трижды и имеются три точки входа в одну и ту же функцию. Однако, поскольку для выбора версии функции, подлежащей вызову, Elixir использует сопоставление с образцом, они не являются повторяющимися определениями. Как и в обычном языке человеческого общения, эти определения называются *предложениями* (clauses). Все предложения для данного имени функции должны быть сгруппированы в одном модуле.

После определения такого модуля у нас появляется возможность вычислять скорости объектов, сброшенных с заданной высоты, на Земле, на Луне и на Марсе. В данном случае вместо команды `recompile`, которая компилирует весь код в файлах `.ex` в проекте, можно просто перезагрузить единственный модуль командой `r()`:

```
iex(1)> r(Drop)
warning: redefining module Drop (current version loaded from
  _build/dev/lib/drop/ebin/Elixir.Drop.beam)
  lib/drop.ex:1

{:reloaded, Drop, [Drop]}
iex(2)> Drop.fall_velocity(:earth, 20)
19.79898987322333
iex(3)> Drop.fall_velocity(:moon, 20)
8.0
iex(4)> Drop.fall_velocity(:mars, 20)
12.181953866272849
```

Если попытаться найти скорость по образцу, не имеющему соответствующего предложения, Elixir сообщит об ошибке:

```
iex(5)> Drop.fall_velocity(:jupiter, 20)
** (FunctionClauseError) no function clause matching in Drop.fall_velocity/2
  drop.ex:3: Drop.fall_velocity(:jupiter, 20)
```

Вы очень скоро поймете, что атомы являются очень важным средством увеличения удобочитаемости кода на Elixir.



Если понадобится сопоставить с образцом значение переменной, добавьте символ крышки (^) перед именем переменной.

Логические атомы

Для представления логических значений в Elixir используются значения `true` и `false`. Несмотря на то что в их основе лежат атомы, `:true` и `:false`, эти имена настолько распространены в программировании, что их можно записывать без двоеточий. Elixir возвращает эти значения при выполнении операций сравнения:

```
iex(1)> 3 < 2
false
iex(2)> 3 > 2
true
iex(3)> 10 == 10
true
iex(4)> :true == true
```

```
true
iex(5)> :false == false
true
```

В Elixir имеются также специальные операторы для работы с этими атомами (которые, как и операторы сравнения, возвращают эти же атомы):

```
iex(6)> true and true
true
iex(7)> true and false
false
iex(8)> true or false
true
iex(9)> false or false
false
iex(10)> not true
false
```

Операторы `and` и `or` принимают два аргумента. Оператор `and` возвращает `true`, только если оба аргумента имеют значение `true`. Оператор `or` возвращает `true`, если хотя бы один из аргументов имеет значение `true`. При сравнении более сложных выражений, чем `true` и `false`, заключение их в круглые скобки будет мудрым решением.

Elixir автоматически использует сокращенную схему вычислений логических выражений. Например, если первый аргумент оператора `and` имеет значение `false`, он не будет вычислять второй аргумент и сразу вернет `false`. Аналогично, если первый аргумент оператора `or` имеет значение `true`, он не будет вычислять второй аргумент и сразу вернет `true`.

Оператор `not` проще, он принимает единственный аргумент и возвращает `true` для аргумента `false` и `false` – для `true`. В отличие от других операторов, располагающихся между аргументами, оператор `not` предшествует своему единственному аргументу.

Если попытаться применить любой из этих операторов к любым другим атомам, Elixir вернет сообщение о недопустимом аргументе:

```
iex(11)> not :bonkers
** (ArgumentError) argument error
:erlang.not(:bonkers)
```



По аналогии со значениями `true` и `false` Elixir позволяет записывать атом `:nil` как `nil`. Существуют также другие атомы, имеющие общепонятный смысл, такие как `:ok` и `:error`, но они больше являются соглашением, чем формальной частью языка, и для них не предполагается какая-то специальная интерпретация. При их использовании требуется добавлять двоеточие.

Ограничители

Функция `fall_velocity` замечательно справляется со своей работой, но есть одна шероховатость. Если она получит отрицательное значение высоты, функция вычисления квадратного корня (`sqrt`) завершится с ошибкой:

```
iex(1)> Drop.fall_velocity(:earth, -20)
** (ArithmeticError) bad argument in arithmetic expression
    (stdlib) :math.sqrt(-392.0)
    drop.ex:4: Drop.fall_velocity/2
```

Как вы понимаете, нельзя закопаться на глубину 20 метров, отпустить предмет и надеяться, что он устремится вверх, к поверхности Земли, с ускорением свободного падения. Впрочем, эту проблему можно легко исправить, возвращая другой вид ошибки.

В языке Elixir есть возможность ограничить данные, которые могут передаваться той или иной функции, с помощью *ограничителей* (`guards`). Ограничители определяются с помощью ключевого слова `when` и дают возможность настроить сопоставление с образцом на основе содержимого аргументов, а не их вида. Ограничители должны оставаться максимально простыми, использовать очень узкий круг встроенных функций и ограничиваться только проверками данных, не имеющими побочных эффектов, — но они все еще могут преобразовывать ваш код.



В приложении А приводится список функций, безопасных для использования в ограничителях.

Ограничивающие выражения возвращают `true` или `false`, как описывалось выше, и побеждает первое, вернувшее результат `true`. То есть можно написать `when true` для ограничителя, который будет вызываться всегда, когда до него дойдет выполнение, или добавить `when false`, чтобы заблокировать некоторый код (например, во время отладки).

В данном простом случае можно предотвратить вызов функции квадратного корня с отрицательным аргументом, добавив ограничители в предложения `fall_velocity`, как показано в примере 3.2 (*ch03/ex2-guards*).

Пример 3.2 ❖ Добавление ограничителей в предложения функции

```
defmodule Drop do
```

```
  def fall_velocity(:earth, distance) when distance >= 0 do
    :math.sqrt(2 * 9.8 * distance)
```

```

end

def fall_velocity(:moon, distance) when distance >= 0 do
  :math.sqrt(2 * 1.6 * distance)
end

def fall_velocity(:mars, distance) when distance >= 0 do
  :math.sqrt(2 * 3.71 * distance)
end

end

```

Выражение `when` описывает условие или комплекс условий в заголовке функции. В данном случае условие требует лишь, чтобы значение `distance` было больше или равно нулю. В Elixir условие «больше или равно» записывается как `>=`, а «меньше или равно» – как `<=`. Если скомпилировать этот модуль и вызвать функцию с положительным значением высоты, результат получится тем же, как и раньше. Но для отрицательного аргумента результат получится иным:

```

iex(1)> recompile
Compiling 1 file (.ex)
:ok
iex(2)> Drop.fall_velocity(:earth, 20)
19.79898987322333
iex(3)> Drop.fall_velocity(:earth, -20)
** (FunctionClauseError) no function clause matching in Drop.fall_velocity/2
    drop.ex:3: Drop.fall_velocity(:earth, -20)

```

Благодаря действию ограничителя интерпретатор Elixir не смог найти функцию, готовую принять отрицательный аргумент. Возможно, такое сообщение об ошибке не выглядит значительным достижением, но с добавлением новых уровней кода сообщение «не обработано» может оказаться полезнее, «произошла ошибка в вычислениях».

Более очевидным и простым примером применения ограничителей мог бы служить код, возвращающий абсолютное значение. Да, в Elixir есть встроенная функция `abs()`, тем не менее давайте рассмотрим пример 3.3, поясняющий, как она действует.

Пример 3.3 ❖ Вычисление абсолютного значения с применением ограничителей

```

defmodule MathDemo do

  def absolute_value(number) when number < 0 do
    -number
  end

```

```
def absolute_value(number) when number == 0 do
  0
end

def absolute_value(number) when number > 0 do
  number
end

end
```

Если вызвать `Mathdemo.absolute_value()` с отрицательным (меньше нуля) аргументом, Elixir вызовет первое предложение, которое вернет инвертированное значение отрицательного аргумента. Если аргумент равен (`==`) нулю, Elixir вызовет второе предложение, возвращающее 0. Наконец, если аргумент имеет положительное значение (больше нуля), Elixir вызовет третье предложение, просто возвращающее полученное число. (Первые два предложения обрабатывают все значения, не являющиеся положительными, поэтому ограничитель в последнем предложении не нужен и от него можно избавиться, как будет показано в примере 3.4.)



Все примеры, начиная с этого момента, построены с помощью Mix и должны запускаться командой `iex -S mix`. Она автоматически скомпилирует код, если понадобится. Для экономии места в книге и чтобы не повторять одно и то же несколько раз, мы будем опускать команду запуска IEx и сообщения, выводимые компилятором.

Опробуем этот код в деле:

```
iex(1)> MathDemo.absolute_value(-20)
20
iex(2)> MathDemo.absolute_value(0)
0
iex(3)> MathDemo.absolute_value(20)
20
```

Такой способ вычислений может показаться избыточно громоздким. Но не волнуйтесь – в Elixir имеется простая логика переключения, которую можно использовать внутри функций. Однако ограничители играют важную роль в выборе из нескольких предложений функции, которые оказываются особенно полезными в рекурсии, с которой мы познакомимся в главе 4.

Elixir пробегает через предложения функции в том порядке, в каком они определены в исходном коде, и останавливается на первом, соответствующем всем условиям. Если обнаружится, что интерпретатор выбирает неправильное предложение, попробуйте переупорядочить предложения в исходном коде или изменить условия в ограничителях.

Кроме того, когда выражение в ограничителе проверяет только одно значение, вы легко можете заменить ограничители использованием сопоставления с образцом. Функция `absolute_value()` в примере 3.4 действует точно так же, как в примере 3.3.

Пример 3.4 ❖ Вычисление абсолютного значения с использованием ограничителей и сопоставления с образцом

```
def module MathDemo do
```

```
  def absolute_value(number) when number < 0 do
    -number
  end
```

```
  def absolute_value(number) do
    number
  end
```

```
end
```

В подобных случаях вам решать, какой подход использовать: более простой или более явный.



В одном ограничителе можно использовать несколько сравнений. Если разделить их оператором `or`, ограничитель вернет `true`, если выполнится хотя бы одно условие. Если разделить их оператором `and`, ограничитель вернет `true`, только если выполнятся все условия.

Обозначайте подчеркиванием все, что не важно

Ограничители помогают точнее определить, как обрабатывать входящие аргументы. Но иногда большая точность просто не нужна. Не все аргументы являются существенными для некоторой операции, особенно это касается аргументов, через которые передаются сложные структуры данных. Можно, конечно, создавать переменные для таких аргументов и никогда не использовать их, но тогда компилятор будет выводить предупреждения (подозревая, что вы допустили ошибку), и эти предупреждения могут вызывать беспокойство у других, использующих ваш код, — почему он заботится не обо всех передаваемых ему аргументах.

Вы можете, например, решить не заботиться о том, какой *плането*¹ (объект планетарной массы, к таким объектам относят планеты,

¹ От англ. *planeto*, planetary mass object. — Прим. перев.

астероиды и луны) указал пользователь вашей функции вычисления скорости, и просто использовать ускорение свободного падения для Земли. В таком случае можно написать код, как в примере 3.5 (*ch03/ex3-underscore*).

Пример 3.5 ❖ Объявление переменной и ее игнорирование

```
defmodule Drop do
```

```
  def fall_velocity(planemo, distance) when distance >= 0 do
    :math.sqrt(2 * 9.8 * distance)
  end
end
```

```
end
```

Этот пример скомпилируется, но компилятор выведет предупреждение, а попытка вычислить скорость, например, на Марсе даст неверный результат:

```
iex(1)> r(Drop)
```

```
r(Drop)
```

```
warning: redefining module Drop (current version loaded from
  _build/dev/lib/drop/ebin/Elixir.Drop.beam)
lib/drop.ex:1
```

```
warning: variable planemo is unused
lib/drop.ex:3
```

```
{:reloaded, Drop, [Drop]}
```

```
iex(2)> Drop.fall_velocity(:mars,20)
```

```
19.79898987322333
```

На Марсе скорость должна была получиться ближе к 12, чем к 19, то есть компилятор был прав, предупредив вас.

Иногда, однако, действительно требуется позаботиться лишь о некоторых аргументах. В таких случаях можно использовать простой символ подчеркивания (`_`). Подчеркивание играет двойную роль: сообщает компилятору, чтобы он не беспокоил вас предупреждениями, и говорит любому, кто будет читать ваш код, что вы не предполагаете использовать этот аргумент. Вы можете попытаться присвоить значение переменной с именем, состоящим из единственного символа подчеркивания, но попытка прочитать ее вызовет ошибку. Elixir считает, что эта переменная никогда не связана со значением:

```
iex(3)> _ = 20
```

```
20
```

```
iex(4)> _
```

```
** (CompileError) iex:4 unbound variable _
```

Если вы действительно желаете, чтобы ваш код был ориентирован исключительно на Землю и игнорировал любые попытки выполнить расчеты для других планет, его можно было бы записать, как показано в примере 3.6.

Пример 3.6 ❖ Преднамеренно игнорирует аргумент с помощью символа подчеркивания

```
defmodule Drop2 do

  def fall_velocity(_, distance) when distance >= 0 do
    :math.sqrt(2 * 9.8 * distance)
  end

end
```

На этот раз компилятор не выведет предупреждения, а любой, кто займется чтением вашего кода, поймет, что первый аргумент не имеет смысла:

```
iex(4)> r(Drop2)
warning: redefining module Drop2 (current version loaded from
  _build/dev/lib/drop/ebin/Elixir.Drop2.beam)
  lib/drop2.ex:1

{:reloaded, Drop2, [Drop2]}
iex(5)> Drop2.fall_velocity(:you_dont_care, 20)
19.79898987322333
```

Символ подчеркивания можно использовать несколько раз, по количеству игнорируемых аргументов. В сопоставлениях с образцом он соответствует всему, чему угодно, и никогда не имеет значения.



Имеется также возможность давать переменным имена, начинающиеся с символа подчеркивания, — например, `_плането` — и компилятор не будет выводить предупреждений, обнаружив, что эти переменные нигде не используются. Однако такие переменные связываются со значениями, и на них можно ссылаться. Соответственно, если более одного раза использовать одно и то же имя, пусть и начинающееся с подчеркивания, в списке параметров, компилятор сообщит об ошибке двукратного использования одного и того же имени.

Структуры данных: кортежи

Кортежи в языке Elixir позволяют объединить несколько элементов в один составной тип данных. Это упрощает передачу сообщений между компонентами и позволяет создавать собственные сложные

типы данных. Кортёжи могут содержать любые данные, поддерживаемые языком Elixir, включая числа, атомы, другие кортежи, а также списки и строки, с которыми вы встретитесь в последующих главах.

Кортёжи устроены просто – это группа элементов, заключенная в фигурные скобки:

```
iex(1)> {:earth, 20}  
{:earth, 20}
```

Кортёж может содержать один элемент или сотню. В практике чаще встречаются кортежи с количеством элементов от двух до пяти. Часто (но не всегда) в начало кортежа добавляют атом, который служит неформальным идентификатором структуры данных, хранящейся в кортеже.

В Elixir имеются встроенные функции для доступа к содержимому кортежей на уровне элементов. Извлечь значение элемента, например, можно с помощью функции `elem`, записать значения в новый кортеж можно с помощью функции `put_elem`, а узнать количество элементов в кортеже можно с помощью функции `tuple_size`. В Elixir (в отличие от Erlang) отсчет элементов в кортежах начинается с нуля, то есть первому элементу соответствует индекс 0, второму 1 и т. д.:

```
iex(2)> tuple={:earth,20}  
{:earth,20}  
iex(3)> elem(tuple,1)  
20  
iex(4)> new_tuple=put_elem(tuple,1,40)  
{:earth,40}  
iex(5)> tuple_size(new_tuple)  
2
```

В сочетании с механизмом сопоставления с образцом кортежи позволяют писать более удобочитаемый код.

Сопоставление с образцом и кортежи

Кортёжи упрощают задачу упаковки нескольких аргументов в один контейнер и позволяют принимающей функции решать, что делать с ними. Сопоставление с образцом для кортежей действует во многом так же, как для атомов, за исключением, конечно, пары фигурных скобок, заключающих каждую группу аргументов, как показано в примере 3.7 (*ch03/ex4-tuples*).

Пример 3.7 ❖ Инкапсуляция аргументов в кортеж`defmodule Drop do`

```

  def fall_velocity({:earth, distance}) when distance >= 0 do
    :math.sqrt(2 * 9.8 * distance)
  end

  def fall_velocity({:moon, distance}) when distance >= 0 do
    :math.sqrt(2 * 1.6 * distance)
  end

  def fall_velocity({:mars, distance}) when distance >= 0 do
    :math.sqrt(2 * 3.71 * distance)
  end
end

```

`end`

Аргумент функции изменился: эта версия определяется как `fall_velocity/1`, а не `fall_velocity/2`, потому что кортеж считается единственным аргументом. Версия с кортежем действует во многом похоже на версию с атомом, но требует добавлять дополнительные фигурные скобки при вызове функции:

```

iex(1)> Drop.fall_velocity({:earth, 20})
19.79898987322333
iex(2)> Drop.fall_velocity({:moon, 20})
8.0
iex(3)> Drop.fall_velocity({:mars, 20})
12.181953866272849

```

Зачем могло бы понадобиться использовать эту форму, учитывая, что все ее отличие заключается в необходимости ввода лишних фигурных скобок? Это не совсем так. Использование кортежей открывает массу новых возможностей. В кортеж можно упаковать, например, множество аргументов, разные атомы и даже функции, созданные с помощью `fn()`. Возможность передачи единственного кортежа вместо груды аргументов дает языку Elixir значительную гибкость, особенно когда дело доходит до передачи сообщений между разными процессами.

Обработка кортежей

Существует много способов обработки кортежей, не только простым сопоставлением с образцом, как было показано в примере 3.7. Принимая кортеж в единственной переменной, с ним можно многое сделать. Начнем с простого – используем кортеж для вызова приватной версии функции. Эта часть примера 3.8 (*ch03/ex5-tuplesMore*) может по-

казаться знакомой, так как это та же самая функция `fall_velocity/2` из примера 3.1.

Пример 3.8 ❖ Инкапсуляция аргументов в кортеж и передача их приватной функции

```
defmodule Drop do
  def fall_velocity({planemo, distance}) when distance >= 0 do
    fall_velocity(planemo, distance)
  end

  defp fall_velocity(:earth, distance) do
    :math.sqrt(2 * 9.8 * distance)
  end

  defp fall_velocity(:moon, distance) do
    :math.sqrt(2 * 1.6 * distance)
  end

  defp fall_velocity(:mars, distance) do
    :math.sqrt(2 * 3.71 * distance)
  end
end
```

Использование объявления `defp` для создания приватной версии означает, что только `fall_velocity/1`, версия с кортежем, будет доступна извне. Функция `fall_velocity/2` доступна только внутри модуля. В данном примере в этом нет особой необходимости, но такой прием – «сделать одну версию общедоступной, а остальные версии, с другими аргументами, оставить приватными» – часто используется в ситуациях, когда требуется сделать функцию доступной, но чтобы внутренние ее механизмы нельзя было использовать извне.

Если вы вызовете эту функцию `fall_velocity/1` (принимающую кортеж, и потому аргументы должны быть заключены в фигурные скобки), она вызовет приватную `fall_velocity/2`, которая вернет значение в `fall_velocity/1`, а та, в свою очередь, вернет его вам. Результат получится тем же, что и прежде:

```
iex(1)> Drop.fall_velocity({:earth, 20})
19.79898987322333
iex(2)> Drop.fall_velocity({:moon, 20})
8.0
iex(3)> Drop.fall_velocity({:mars, 20})
12.181953866272849
```

Существует несколько разных способов извлечения данных из кортежей. Можно ссылаться на элементы кортежей по их порядко-

вым номерам, используя встроенный макрос `elem/2`, который принимает кортеж и числовой индекс. Первый элемент кортежа имеет индекс 0, второй – индекс 1 и т. д.:

```
def fall_velocity(where) do
  fall_velocity(elem(where,0), elem(where,1))
end
```

Также для извлечения элементов из кортежа можно использовать механизм сопоставления с образцом:

```
def fall_velocity(where) do
  {planemo, distance} = where
  fall_velocity(planemo,distance)
end
```

Результат последней строки станет возвращаемым значением функции `fall_velocity/1`.

Сопоставление с образцом в данном случае немного отличается от аналогичного подхода в примере 3.8. Функция принимает кортеж в аргументе и связывает его с переменной `where`. (Если окажется, что `where` – не кортеж, инструкция `{planemo, distance} = where` вызовет ошибку.) Зная структуру кортежа, мы легко можем извлечь из него данные, используя сопоставление с образцом внутри функции. Переменные `planemo` и `distance` будут связаны со значениями, содержащимися в кортеже `where`, и затем смогут использоваться в вызове `fall_velocity/2`.

Глава 4

Логика и рекурсия

До сих пор мы рассматривали логичные, но очень простые примеры на Elixir. Механизм сопоставления с образцом управляет потоком выполнения программы и в ответ на запросы возвращает соответствующие ответы. Даже притом, что этого вполне достаточно для решения многих задач, иногда бывает необходимо реализовать нечто более сложное, особенно когда в работе используются большие и более сложные структуры данных.

Логика внутри функций

Сопоставление с образцом и ограничители – мощные инструменты, но иногда проще выполнить ряд сравнений внутри функции, а не создавать новых предложений. Создатели Elixir согласны с этим мнением и предусмотрели две конструкции для оценки условий внутри функций: выражение `case` и реже используемые выражения `cond` и `if`.

Конструкция `case` позволяет использовать сопоставление с образцом и ограничители внутри предложения. Она читается очень естественно, когда требуется сравнить единственное значение (или набор значений) с несколькими вариантами. Конструкция `cond` просто вычисляет серию выражений, без сопоставления с образцом. Обычно код с конструкцией `cond` получается более удобочитаемым, когда разные варианты определяются разными комбинациями значений. Конструкция `if` вычисляет только одно выражение.

Все эти конструкции возвращают значение, которое может использоваться в последующем коде.

Конструкция `case`

Конструкция `case` дает возможность выполнить сопоставление с образцом внутри функции. Если вы считаете, что функция с несколь-

кими предложениями в примере 3.2 трудно читается, можете создать версию, которая показана в примере 4.1 (*ch04/ex1-case*).

Пример 4.1 ❖ Перенос сопоставления с образцом внутрь функции, в виде конструкции `case`

```
defmodule Drop do

  def fall_velocity(planemo, distance) when distance >= 0 do
    case planemo do
      :earth -> :math.sqrt(2 * 9.8 * distance)
      :moon  -> :math.sqrt(2 * 1.6 * distance)
      :mars  -> :math.sqrt(2 * 3.71 * distance)
    end
  end
end
```

Конструкция `case` сравнит атом в `planemo` со значениями в списке (образцами), в порядке их перечисления. Сопоставление прекращается после обнаружения первого же совпадения. За каждым образцом, имеющим вид `a ->`, следует выражение, которое можно читать как «производит». Конструкция `case` возвращает результат выражения, следующего за совпавшим атомом, а так как `case` — последняя конструкция в функции, возвращающая значение, функция вернет ее результат.



Используйте символ подчеркивания (`_`) для определения образца, совпадающего «с чем угодно». Но имейте в виду, что этот вариант всегда должен стоять последним — любые другие варианты, следующие за ним, никогда не будут проверяться.

Эта версия функции должна возвращать те же самые результаты:

```
iex(1)> Drop.fall_velocity(:earth, 20)
19.79898987322333
iex(2)> Drop.fall_velocity(:moon, 20)
8.0
iex(3)> Drop.fall_velocity(:mars, -20)
** (FunctionClauseError) no function clause matching in Drop.fall_velocity/2
    (drop) lib/drop.ex:3: Drop.fall_velocity(:mars, -20)
```

Конструкция `case` переключается между планетами, а ограничитель в определении функции предохранит ее от отрицательных значений, производя (правильное) сообщение об ошибке, как показано в строке 3.

Значение, возвращаемое конструкцией `case`, можно также использовать для устранения повторяющегося кода и сделать логику про-

граммы более ясной. В данном случае единственное отличие между тремя вариантами вычислений заключается в ускорении свободного падения Земли, Луны и Марса. В примере 4.2 (*ch04/ex2-case*) показано, как заставить конструкцию `case` возвращать ускорение свободного падения и использовать его в выражении в конце функции.

Пример 4.2 ❖ Использование значения, возвращаемого конструкцией `case`, для устранения повторяющегося кода

```
defmodule Drop do
  def fall_velocity(planemo, distance) when distance >= 0 do
    gravity = case planemo do
      :earth -> 9.8
      :moon  -> 1.6
      :mars  -> 3.71
    end
    :math.sqrt(2 * gravity * distance)
  end
end
```

На этот раз функция присваивает переменной `gravity` значение, возвращаемое конструкцией `case`. Теперь формула `math:sqrt(2 * gravity * distance)` в последней строке функции, производящая возвращаемое значение функции, стала более ясной и понятной.

С инструкцией `case` также можно использовать ограничители, хотя в этом случае они выглядят менее элегантно, как показано в примере 4.3 (*ch04/ex3-case*). Такой подход имел бы больше смысла, если бы для разных планет предъявлялись разные требования к высоте.

Пример 4.3 ❖ Перемещение ограничителей в инструкцию `case`

```
defmodule Drop do
  def fall_velocity(planemo, distance) do
    gravity = case planemo do
      :earth when distance >= 0 -> 9.8
      :moon  when distance >= 0 -> 1.6
      :mars  when distance >= 0 -> 3.71
    end
    :math.sqrt(2 * gravity * distance)
  end
end
```

Эта реализация произведет тот же результат, только сообщение в случае ошибки изменится с `FunctionClauseError` на `CaseClauseError`:

```
iex(3)> r(Drop)
warning: redefining module Drop (current version defined in memory)
lib/drop.ex:1

{:reloaded, Drop, [Drop]}
iex(4)> Drop.fall_velocity(:earth, 20)
Logic Inside of Functions | 41
19.79898987322333
iex(5)> Drop.fall_velocity(:moon, 20)
8.0
iex(6)> Drop.fall_velocity(:mars, -20)
** (CaseClauseError) no case clause matching: :mars
(drop) lib/drop.ex:4: Drop.fall_velocity/2
```

Сообщение об ошибке правильно сообщает, что конструкции `case` не удалось найти соответствующую ветку для `:mars`, но оно вводит в заблуждение, потому что проблема заключается не в `:mars`, а в ограничителе, проверяющем переменную `distance`. Если Elixir сообщает, что инструкция `case` не нашла совпадения, когда совпадение у вас перед глазами, проверьте ограничитель.

Конструкция `cond`

Конструкция `cond` во многом схожа с инструкцией `case`, но она не использует сопоставление с образцом. При желании вы можете добавить в нее условие, выполняющееся при любых обстоятельствах, — выражение в конце, возвращающее `true`. Эта конструкция часто упрощает реализацию логики, основанной на более широких сравнениях, чем простое сопоставление с образцом.

Допустим, к примеру, что точность функции `fall_velocity` для наших нужд слишком велика. И вместо фактической скорости требуется описать относительную скорость объекта, сбрасываемого с башни заданной высоты. Вы можете добавить конструкцию `cond` (см. пример 4.4 в папке *ch04/ex4-cond*), которая делает это, в конец примера 4.2.

Пример 4.4 ❖ Добавление конструкции `cond` для преобразования чисел в атомы

```
defmodule Drop do

  def fall_velocity(planemo, distance) when distance >= 0 do
    gravity = case planemo do
      :earth -> 9.8
      :moon  -> 1.6
      :mars  -> 3.71
    end
  end
end
```

```

velocity = :math.sqrt(2 * gravity * distance)

cond do
  velocity == 0           -> :stable
  velocity < 5            -> :slow
  velocity >= 5 and velocity < 10 -> :moving
  velocity >= 10 and velocity < 20 -> :fast
  velocity >= 20          -> :speedy
end
end
end

```

Конструкция `cond` в этом примере возвращает значение (атом, описывающий скорость), опираясь на множество ограничителей в ней. Так как это последнее значение, получаемое в функции, оно становится ее возвращаемым значением.

Результаты немного отличаются от предыдущих испытаний:

```

iex(6)> r(Drop)
warning: redefining module Drop (current version defined in memory)
lib/drop.ex:1

{:reloaded, Drop, [Drop]}
iex(7)> Drop.fall_velocity(:earth, 20)
:fast
iex(8)> Drop.fall_velocity(:moon, 20)
:moving
iex(9)> Drop.fall_velocity(:mars, 20)
:fast
iex(10)> Drop.fall_velocity(:earth, 30)
:speedy

```

При необходимости значение, произведенное конструкцией `cond`, можно сохранить в переменной. Для этого достаточно заменить `cond` до в первой строке на что-нибудь вроде `description = cond do`.



Elixir вычисляет инструкции `cond` и `if`, опираясь на понятие «истинности». Истинными считаются все значения, кроме `nil` и `false`.

if или else

Для простейших случаев Elixir предлагает функцию `if`, которая проверяет только одно условие, и позволяет использовать `else` вслед за ней, если в случае невыполнения условия также требуется выполнить какие-то действия.

Пример 4.5 (*ch04/ex5-if*) посылает предупреждение в стандартный вывод (в данном случае в оболочку `IEEx`), если скорость падения объекта слишком высока. Он использует ближайшего родственника

конструкции `cond` – функцию `if`, чтобы принять решение о необходимости вывода дополнительного сообщения.

Пример 4.5 ❖ Отправка дополнительного предупреждения, если скорость слишком высока

```
defmodule Drop do
  def fall_velocity(planemo, distance) when distance >= 0 do
    gravity = case planemo do
      :earth -> 9.8
      :moon  -> 1.6
      :mars  -> 3.71
    end

    velocity = :math.sqrt(2 * gravity * distance)

    if velocity > 20 do
      IO.puts("Look out below!")
    else
      IO.puts("Reasonable...")
    end

    velocity
  end
end
```

Новая инструкция `if` проверяет переменную `velocity`, сравнивая ее с числом 20. Если она оказывается больше 20, вызывается функция `IO.puts`, имеющая побочный эффект: вывод сообщения на экран. Если значение переменной `velocity` меньше или равно 20, инструкция `else` выведет менее настораживающее сообщение (обращение к переменной `velocity` в конце гарантирует возврат ее значения из функции):

```
iex(1)> Drop.fall_velocity(:earth, 50)
Look out below!
31.304951684997057
iex(2)> Drop.fall_velocity(:moon, 100)
Reasonable...
17.88854381999832
```

Инструкции `if` можно записывать несколькими разными способами. Если инструкция `if` достаточно компактна, ее можно записать в одну строку:

```
iex(3)> x=20
20
iex(4)> if x>10 do :large end
:large
```

Более того, в эту же строку можно даже добавить `else`:

```
iex(5)> if x>10 do :large else :small end
:large
```

Существует альтернативная форма записи – с добавлением двоеточия после `do`:

```
iex(6)> if x>10, do: :large, else: :small
:large
```

Кому-то из вас инструкция `unless` может показаться более выразительной, чем `if`, которая проверяет *невыполнение* указанного условия:

```
iex(7)> unless x>10, do: :small, else: :large
:large
```

Присваивание значений переменным в конструкциях `case` и `if`

Во всех ветвях, создаваемых в инструкциях `case`, `cond` и `if`, допускается связывать переменные со значениями. Обычно это весьма ценная возможность, но может порождать проблемы при присваивании значений разным переменным в разных ветвях. Продемонстрируем эту проблему в примере 4.6 (*ch04/ex6-broken*).

Пример 4.6 ❖ Конструкция `cond` с ошибкой

```
defmodule Broken do
  def bad_cond(test_val) do
    cond do
      test_val < 0 -> x=1
      test_val >= 0 -> y=2
    end
    x+y
  end
end
```

Компилятор Elixir заметит проблему и предложит решение. Вывод ниже переформатирован, чтобы уместить строки по ширине книжной страницы:

```
$ iex -S mix
Erlang/OTP 19 [erts-8.0] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe]
[kernel-poll:false]

Compiling 1 file (.ex)
```

warning: the variable "x" is unsafe as it has been set inside a case/cond/receive/if/||. Please explicitly return the variable value instead.

For example:

```
case int do
  1 -> atom = :one
  2 -> atom = :two
end
```

should be written as

```
atom =
  case int do
    1 -> :one
    2 -> :two
  end
```

Unsafe variable found at:

```
lib/broken.ex:10
```

;; similar warning for variable "y"

Generated broken app

Interactive Elixir (1.3.1) - press Ctrl+C to exit (type h() ENTER for help)

```
iex(1)>
```

Если проигнорировать это предупреждение и попытаться использовать функцию с ошибкой, во время выполнения появится сообщение:

```
iex(1)> Broken.bad_cond(20)
```

```
** (ArithmeticError) bad argument in arithmetic expression
(broken) lib/broken.ex:10: Broken.bad_cond/1
```



Elixir также снимает жесткие ограничения языка Erlang на допустимые операции в инструкциях `if` и `cond`. Erlang позволяет использовать только то, что допустимо в ограничителях, чтобы исключить любые побочные эффекты. Elixir не накладывает таких ограничений.

Самый желательный побочный эффект: IO.puts

До примера 4.5 все наши фрагменты кода на Elixir производили только один результат. Мы передавали аргумент или аргументы функции и получали возвращаемое значение. Это самый ясный подход к решению задач: вы можете полагаться, что все будет работать точно так же снова и снова, потому что предшествующие вычисления не оставляют никаких следов в системе.

Пример 4.5 демонстрирует отступление от этой модели, создавая побочный эффект, который сохраняется после выполнения функции. Этим побочным эффектом является лишь сообщение, появляющееся в оболочке (или в стандартном выводе, если код на Elixir выполняется за пределами оболочки). Приложения, которые передают информацию нескольким пользователям или хранят ее дольше, чем в короткий период обработки, нуждаются в более мощных побочных эффектах, таких как сохранение информации в базе данных.

При программировании на языке Elixir предлагается использовать побочные эффекты, *только* когда они действительно необходимы. Приложению, реализующему интерфейс к базе данных, например, действительно необходимо выполнять операции чтения и записи с базой данных. Приложению, взаимодействующему с пользователями, необходимо выводить информацию на экран (или другой интерфейс).

Побочные эффекты также очень полезны для трассировки логики в процессе разработки. Самый простой способ увидеть, что делает программа, пока вы не познакомились со встроенными инструментами трассировки и отладки, – вставить в программу инструкции, сообщающие о ее состоянии в наиболее интересных для вас точках. Этот код не должен оставаться в окончательной версии программы, но он поможет вам в процессе разработки понять, как действует ваш код.

Функция `IO.puts` позволяет посылать информацию в консоль или, когда программа выполняется за пределами консоли, в другие места. Пока мы будем использовать ее для отправки сообщений из программ в консоль. Пример 4.5 продемонстрировал простейший способ использования функции `IO.puts`, которая просто выводит сообщение, переданное ей в двойных кавычках:

```
IO.puts("Look out below!")
```

`IO.puts` добавляет символ перевода строки в конец, сообщая консоли, что следующее сообщение должно быть выведено с начала следующей строки. Благодаря этому вывод на экран выглядит более удобочитаемым. Если вам понадобится вывести сообщение без символа перевода строки в конце, используйте функцию `IO.write`. Если потребуется вывести значение переменной, не являющееся строкой, используйте `IO.inspect`.



Elixir категорически запрещает использовать любые операции с побочными эффектами в выражениях-ограничителях. Если бы побочные эффекты были допустимы в ограничителях, они могли бы производиться всякий раз, когда вычисляется выражение-ограничитель. Функ-

ция `IO.puts`, вероятно, не сможет сделать ничего ужасного, но описанное правило также запрещает ее применение в ограничителях.

Простая рекурсия

Основным инструментом выполнения повторяющихся действий является *рекурсия*: функция, которая вызывает саму себя, пока (как предполагается) не будет достигнут конец вычислений. Звучит сложно, но в действительности все намного проще.

Существуют два основных вида рекурсии. В некоторых случаях окончание рекурсии определяется естественными причинами, когда процесс достигает конца обрабатываемой коллекции или другого естественного предела. В других ситуациях, когда естественный предел отсутствует, приходится следить за результатом и завершать обработку в нужный момент. Овладев этими двумя основными формами, вы сможете создавать более сложные их вариации.



Существует третья форма, в которой последовательность рекурсивных вызовов никогда не достигает конца. Эта форма называется «бесконечным циклом» и часто считается ошибкой, которой следует избегать. Однако, как вы увидите в главе 9, даже бесконечная рекурсия может пригодиться на практике, например когда требуется организовать бесконечный цикл приема и обработки сообщений.

Обратный отсчет

Простейшим примером рекурсии с естественным пределом является обратный отсчет, как, например, в процедуре запуска ракеты. Рекурсия начинается с некоторым значением в счетчике, который затем последовательно уменьшается до нуля. Когда счетчик достигает нуля, рекурсия завершается (и ракета покидает стартовый стол).

Чтобы реализовать такую рекурсию в Elixir, нужно передать функции начальное число. Если число больше нуля, она объявляет полученное число и вызывает саму себя с тем же числом минус единица. Если число равно нулю (или меньше), она сообщает **blastoff!** (пуск!) и завершается. Пример 4.7 (*ch04/ex7-countdown*) демонстрирует один из вариантов реализации такого поведения.

Пример 4.7 ❖ Обратный отсчет

```
defmodule Count do
```

```
  def countdown(from) when from > 0 do
    IO.inspect(from)
```



```

    countdown(from-1)
end

def countdown(from) do
  IO.puts("blastoff!")
end

end

```

Последнее предложение функции могло бы иметь ограничитель `when from <= 0`, но он необходим, только чтобы человеку, читающему код, было понятнее, когда произойдет пуск. Избыточные ограничители могут вызывать путаницу в будущем, поэтому в данном случае краткость является лучшим выбором. Правда, при этом вы получите предупреждение, что `from` не используется в последнем предложении:

```

$ iex -S mix
Erlang/OTP 19 [erts-8.0] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe]
[kernel-poll:false]

Compiling 1 file (.ex)
warning: variable from is unused
  lib/count.ex:8

Generated count app
Interactive Elixir (1.3.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> Count.countdown(2)
2
1
blastoff!
:ok

```

В первый раз Elixir выбрал первое предложение функции `countdown(from)` и передал ей значение 2. Это предложение вывело число 2 и символ перевода строки, а затем вызвало функцию `countdown` еще раз, передав значение 1. Это привело к повторному вызову первого предложения. Оно вывело 1 и символ перевода строки, а затем снова вызвало функцию `countdown` – на этот раз передав значение 0.

Значение 0 обеспечило вызов второго предложения, которое вывело сообщение `blastoff!` и завершилось. После перечисления трех значений функция достигла естественного предела.



Завершение рекурсии также можно было бы реализовать с помощью инструкции `if` в функции `countdown(from)` с единственным предложением, однако это не совсем обычно для Elixir. Мы считаем, что ограничители более очевидны в таких ситуациях, впрочем, вы можете думать иначе.

Прямой отсчет

Прямой отсчет немного сложнее, потому что не имеет естественной конечной точки, и, используя этот подход, вы не сможете смоделировать код в примере 4.7. Вместо обратного счетчика мы должны использовать *аккумулятор*. Аккумулятор — это дополнительный аргумент, хранящий последнее достигнутое значение, который передается в рекурсивный вызов функции. (При необходимости можно организовать поддержку нескольких аргументов-аккумуляторов, однако часто одного вполне достаточно.) В примере 4.8 (*ch04/ex8-countup*) демонстрируется функция `countup()`, добавленная в модуль `count`. Эта функция позволяет вести прямой отсчет до указанного числа.

Пример 4.8 ❖ Прямой отсчет

```
defmodule Count do

  def countup(limit) do
    countup(1, limit)
  end

  defp countup(count, limit) when count <= limit do
    IO.inspect(count)
    countup(count+1, limit)
  end

  # использовать подчеркивание,
  # чтобы подавить предупреждение "unused variable"

  defp countup(_count, _limit) do
    IO.puts("finished!")
  end

end
```

Эта функция производит следующие результаты:

```
iex(1)> Count.countup(2)
1
2
finished!
:ok
```

Функция `countup/2`, выполняющая основную работу, определена как приватная функция и не экспортируется модулем. Это не является обязательным требованием — вы можете объявить ее общедоступной, чтобы дать возможность выполнять прямой отсчет до любо-

го произвольного значения, – но это не является распространенной практикой. Приватность внутренних рекурсивных функций уменьшает вероятность, что кто-то по ошибке использует их в непредусмотренных ситуациях. В данном случае это не имеет никакого значения, но в других ситуациях, особенно связанных с изменением данных, такое решение может оказаться весьма кстати.

Когда мы вызываем `countup/1`, она вызывает `countup/2` с аргументом `1` (текущее значение счетчика) и значением верхнего предела.

Если текущее значение счетчика меньше или равно верхнему пределу, первое предложение функции `countup/2` сообщает значение счетчика с помощью `IO.puts`. Затем оно снова вызывает себя, увеличивая счетчик, но оставляя верхний предел в неприкосновенности.

Если текущее значение счетчика больше верхнего предела, срабатывает ограничитель первого предложения и вызывается второе предложение: оно сообщает "Finished" (Конец) и завершает выполнение.



Ограничители в данном примере помогают избежать попадания в бесконечный цикл. Вы можете передать функции `countup/1` ноль, отрицательное или вещественное число, и она завершится в нужный момент. Однако если вместо `>=` или `<=` в ограничителе использовать оператор `==` или `===`, можно столкнуться с серьезной проблемой.

Рекурсия с возвратом значения

Примеры со счетчиками, представленными выше, реализуются очень просто – они лишь демонстрируют, как работает рекурсия, отбрасывая любые возвращаемые значения. Да, там есть возвращаемые значения – вызовы `IO.puts` возвращают атом `:ok`, но они не имеют практического применения. Более типично, когда рекурсивный вызов использует возвращаемое значение.

Классическим примером рекурсии является вычисление факториалов. Факториал – это произведение всех положительных чисел, равных или меньше аргумента. Факториал числа `1` равен `1`; для числа `1` должно возвращаться само число `1`. Факториал числа `2` равен `2`; $2 \times 1 = 2$. Самое интересное начинается с числа `3`, где $3 \times 2 \times 1 = 6$. Для числа `4`: $4 \times 3 \times 2 \times 1 = 24$. С увеличением аргумента результат начинает быстро увеличиваться.

Однако существует более общий шаблон вычислений. Факториал любого целого числа можно найти умножением этого целого числа на факториал целого числа, на единицу меньше данного. Это делает по-

иск факториала отличным кандидатом на реализацию с применением рекурсии, когда результат вычисления факториала меньшего числа используется для вычисления факториала большего числа. Этот подход напоминает логику обратного отсчета, но программа должна не просто вести обратный отсчет, но еще и собирать результат, как показано в примере 4.9 (*ch04/ex9-factorial-down*).

Пример 4.9 ❖ Поиск факториала с использованием логики обратного отсчета

```
defmodule Fact do
  def factorial(n) when n > 1 do
    n * factorial(n - 1)
  end

  def factorial(n) when n <= 1 do
    1
  end
end
```

Первое предложение функции `factorial` используется для обработки чисел больше 1. Оно возвращает значение – произведение числа *n* на факториал следующего целого числа, меньше текущего. Второе предложение возвращает значение 1, когда *n* достигает 1. Использование оператора `<=` в ограничителе для этого предложения, вместо `==`, обеспечивает более высокую надежность при получении вещественных или отрицательных аргументов, хотя в этом случае ответ получается неверным: факториалы можно вычислять только для целых чисел от 1 и выше:

```
iex(1)> Fact.factorial(1)
1
iex(2)> Fact.factorial(3)
6
iex(3)> Fact.factorial(4)
24
iex(4)> Fact.factorial(40)
815915283247897734345611269596115894272000000000
```

Код работает, но кому-то из вас его работа может показаться неочевидной. Да, функция выполняет обратный отсчет и собирает значения, но если вы хотите увидеть механику работы во всех подробностях, добавьте несколько вызовов `IO.puts`, как показано в примере 4.10 (*ch04/ex10-factorial-down-instrumented*).

Пример 4.10 ❖ Исследование механики работы рекурсивного алгоритма вычисления факториала

```
defmodule Fact do
  def factorial(n) when n > 1 do
    IO.puts("Calling from #{n}.")
    result = n * factorial(n - 1)
    IO.puts("#{n} yields #{result}.")
    result
  end

  def factorial(n) when n <= 1 do
    IO.puts("Calling from 1.")
    IO.puts("1 yields 1.")
    1
  end
end
```

Здесь добавлен избыточный код. Чтобы вывести результат рекурсивного вызова и вернуть это значение для следующего рекурсивного вызова, нам пришлось сохранить данный результат в переменной `result`. Вызовы `IO.puts` позволяют увидеть, какие значения производятся в результате. Далее, поскольку значение последнего выражения в предложении функции становится его возвращаемым значением, в конец добавлено дополнительное обращение к переменной `result`. Второе предложение, для 1, имеет аналогичную структуру, с той лишь разницей, что оно может просто сообщить `1 yields 1`. (для 1 получено 1), потому что это всегда так.

Скомпилировав и запустив этот пример, вы увидите примерно следующее:

```
iex(1)> 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
```

Несмотря на то что вызовы выполняют обратный отсчет, как предполагает логика работы функции, сообщения с результатами не выводятся, пока обратный отсчет не закончится, после чего они появляются в порядке нарастания результата.

Причина такого поведения в том, что вызов функции не возвращает значения, пока обратный отсчет не завершится. До тех пор Elixir создает новые кадры стека, соответствующие вызовам функции. Каждый кадр стека можно представить как постановку вызова функции на паузу до момента, пока следующий рекурсивный вызов не вернет значения. Как только вызов с аргументом 1 просто вернет значение, прервав цепь рекурсивных вызовов, Elixir сможет «раскрутить» кадры стека в обратном направлении и вычислить `result`. В это время производится вывод результатов – «X yields Y.» – в порядке раскручивания кадров стека.

«Раскручивание» также свидетельствует о том, что рекурсия в примерах 4.9 и 4.10 не является *хвостовой рекурсией*. Когда Elixir встречает код, который последним действием выполняет рекурсивный вызов, он может оптимизировать работу со стеком и отказаться от создания новых кадров. Это может быть неважно для однократных вычислений, но позволяет получить существенный выигрыш, когда предполагается, что код будет работать продолжительное время.

Добиться оптимизации хвостовой рекурсии в задаче поиска факториалов можно, применив методику прямого отсчета. Вы получите тот же результат (по крайней мере, для целых чисел), но порядок вычислений будет иным, как показано в примере 4.11 (*ch04/ex11-factorial-up*).

Пример 4.11 ❖ Поиск факториала с использованием логики прямого отсчета

```
defmodule Fact do
  def factorial(n) do
    factorial(1, n, 1)
  end

  defp factorial(current, n, result) when current <= n do
    new_result = result * current
    IO.puts("#{current} yields #{new_result}.")
    factorial(current + 1, n, new_result)
  end

  defp factorial(_current, _n, result) do
    IO.puts("finished!")
    result
  end
end
```

Так же, как в предыдущем примере реализации прямого отсчета, главная функция (здесь `factorial/1`) вызывает приватную функцию

`factorial/3`. В данном случае используются два аккумулятора. Аккумулятор `current` хранит текущее значение счетчика, а `result` – результат предыдущего умножения. Когда значение `current` оказывается больше ограничивающего значения `n`, первый ограничитель терпит неудачу, и вызывается второе предложение функции, которое выводит `"finished"` и возвращает `result`. (Чтобы избежать предупреждений компилятора, мы добавили символ подчеркивания в начало имен переменных `current` и `n`, потому что последнее предложение не использует их.)

Поскольку последним действием, которое выполняет `factorial/3`, является рекурсивный вызов самой себя, этот вызов является хвостовым. В результате Elixir может минимизировать объем информации, хранимой на стеке, без ущерба для работы кода.

Этот пример дает те же результаты, но производит вычисления в ином порядке:

```
iex(1)> Fact.factorial(4)
1 yields 1.
2 yields 2.
3 yields 6.
4 yields 24.
finished!
24
```

Даже притом, что этому коду приходится хранить больше значений, во время выполнения он потребляет меньше памяти. По достижении конечного значения ему не требуется выполнять никаких других вычислений. Это конечное значение и есть результат вычислений. Обратите также внимание, что при подобном подходе упростилась также структура вызовов `IO.puts`. Если их удалить или закомментировать, оставшийся код не потребует изменений.

Глава 5

Взаимодействие с человеком

Инструменты языка Erlang для работы со строками были полностью переделаны в Elixir, благодаря чему появилась поддержка Юникода (UTF-8), и теперь строки превратились в нечто большее, чем простые списки символов. В главе 4 вы познакомились с функцией `IO.puts`, выполняющей вывод строк, но вам предстоит узнать намного больше, чтобы уметь организовать взаимодействие с человеком и, возможно, с другими приложениями. Эта глава, по крайней мере, покажет вам, как конструировать более удобные интерфейсы для тестирования кода, чем простой вызов функций из `IO`.



Если вы еще не остыли после обсуждения рекурсии в главе 4 и вам не терпится продолжить исследование этой темы, можете прямо сейчас перейти к главе 6, где в начале и в середине вы найдете продолжение дискуссии.

Строки

Атомы прекрасно подходят для обмена сообщениями внутри приложения, но они не предназначены для взаимодействия с внешним окружением процессов Erlang. Чтобы собрать фразу или скомпоновать какую-то информацию, необходимо нечто более гибкое. Строки — вот та структура, что вам нужна. Вы уже немного знакомы со строками, которые мы использовали в роли аргументов функции `IO.puts` в примере 4.5:

```
IO.puts("Look out below!")
```


Содержимое в двойных кавычках (Look out below!) и есть строка. Строка – это последовательность символов. Если понадобится включить двойную кавычку в строку, экранируйте ее символом обратного слеша, например: \". Комбинация \n обозначает символ перевода строки. Чтобы включить символ обратного слеша, его нужно повторить дважды: \\. Полный список экранированных и других последовательностей вы найдете в приложении А.

Если строка создается в интерактивной оболочке, Elixir выведет ее с сохранением экранированных последовательностей. Чтобы увидеть, что «в действительности» содержится в строке, используйте IO.puts:

```
iex(1)> x = "Quote - \" in a string. \n Backslash, too: \\ . \n"
"Quote - \" in a string. \n Backslash, too: \\ . \n"
iex(2)> IO.puts(x)
Quote - " in a string.
Backslash, too: \ .
:ok
```



Если вы начали ввод строки и забыли добавить в конец закрывающую кавычку, после нажатия **Enter** оболочка IEx просто перенесет текст на новую линию в той же строке ввода. Такое поведение позволяет добавлять символы перевода строки, но оно может сбивать с толку. Если вам покажется, что вы застряли непонятно где, просто введите " – и оболочка перейдет в нормальный режим работы.

Elixir поддерживает операции создания новых строк. Простейшей из них является операция *конкатенации*, объединяющая две строки в одну. Для этой цели в Elixir используется необычный, но функциональный оператор <>:

```
iex(3)> "el" <> "ixir"
"elixir"
iex(4)> a="el"
"el"
iex(5)> a <> "ixir"
"elixir"
```

В Elixir имеется также механизм интерполяции строк, который добавляет в строку все, что заключено в фигурных скобках {}. Мы уже пользовались им в примере 4.10 для вывода значений переменных:

```
IO.puts("#{n} yields #{result}.")
```

Встретив комбинацию #{ } в строке, Elixir вычисляет выражение в фигурных скобках, преобразует полученное значение в строку, если необходимо, и вставляет фрагмент в основную строку. Интерполяция

выполняется только один раз. Даже если переменная, используемая в строке, изменится, это не повлияет на содержимое интерполированной строки:

```
iex(6)> a = "this"
"this"
iex(7)> b = "The value of a is #{a}."
"The value of a is this."
iex(8)> a = "that"
"that"
iex(9)> b
"The value of a is this."
```

В фигурные скобки можно поместить все, что возвращает значение: переменную, вызов функции или выражение. Мы обычно используем переменные, но у вас могут быть свои потребности. Как и в любых других вычислениях, если значение в фигурных скобках не может быть вычислено, выводится сообщение об ошибке.

Интерполяция может применяться только к значениям, которые сами являются строками или имеют естественное строковое представление (например, числа). Если вам потребуется интерполировать какое-то другое значение, его следует заключить в вызов функции `inspect`:

```
iex(10)> x = 7 * 5
35
iex(11)> "x is now #{x}"
"x is now 35"
iex(12)> y = {4, 5, 6}
{4,5,6}
iex(13)> "y is now #{y}"
** (Protocol.UndefinedError) protocol String.Chars not
implemented for {4, 5, 6}
    (elixir) lib/string/chars.ex:3:
      String.Chars.impl_for!/1
    (elixir) lib/string/chars.ex:17:
      String.Chars.to_string/1
iex(14)> "y is now #{inspect y}"
"y is now {4,5,6}"
```

Кроме того, в Elixir имеются два оператора сравнения строк, оператор `==` и оператор `===` (точное, или строгое, равенство). Оператор `==` самый простой в этой паре, хотя второй дает те же результаты:

```
iex(15)> "el" == "el"
true
iex(16)> "el" == "ixir"
```

```
false
iex(17)> "el" === "el"
true
iex(18)> "el" === "ixir"
false
```

В Elixir отсутствуют функции, изменяющие строки на месте, так как это плохо соответствует модели, согласно которой значения переменных не меняются. Однако имеется набор функций для поиска содержимого в строках и деления или дополнения строк, которые все вместе позволяют извлекать информацию из строк и объединять ее в новую строку.

Чтобы узнать больше о поддержке строк, обязательно загляните в документацию Elixir с описанием модулей `String` и `Regex` (поддержка регулярных выражений).

Многострочные строки

Многострочные строки, которые иногда называют *встроенными документами* (heredocs), позволяют создавать строки, содержащие символы перевода строки. В главе 2 мы уже упоминали их как один из способов создания документации, но их можно использовать также для других целей.

В отличие от обычных строк, многострочные строки открываются и закрываются тремя двойными кавычками:

```
iex(1)> multi = """
...(1)> This is a multiline
...(1)> string, also called a heredoc.
...(1)> """
"This is a multiline\nstring, also called a heredoc.\n"
iex(2)> IO.puts(multi)
This is a multiline
string, also called a heredoc.
:ok
```

Кроме необычного способа ввода, в остальном многострочные строки обрабатываются так же, как обычные строки.

Юникод

Elixir прекрасно справляется со строками Юникода (UTF-8). Функция `String.length/1` возвращает количество графем Юникода, содержащихся в ее аргументе. Это число не всегда совпадает с количеством

байтов, занимаемых строкой в памяти, так как для хранения многих символов Юникода требуется несколько байтов памяти. Чтобы узнать количество байтов, используйте функцию `byte_size/1`:

```
iex(1)> str="서울 - 대한민국" # Сеул, республика Корея
"서울 - 대한민국"
iex(2)> String.length(str)
9
iex(3)> byte_size(str)
21
```

СПИСКИ СИМВОЛОВ

Обработка строк в Elixir выполняется совершенно иначе, чем в Erlang. В Erlang все строки интерпретируются как списки символов; со списками этого вида вы познакомитесь в главе 6. Так как многим программам на Elixir приходится использовать в своей работе библиотеки Erlang, Elixir предоставляет поддержку списков символов, совместимых с Erlang.



Списки символов работают медленнее и занимают больше памяти, чем строки, поэтому их следует использовать в последнюю очередь.

Чтобы создать список символов, нужно заключить строку в одиночные кавычки:

```
iex(1)> x = 'ixir'
'ixir'
```

Для конкатенации списков символов используется оператор `++` вместо `<>`:

```
iex(2)> 'el' ++ 'ixir'
'elixir'
```

Список символов можно преобразовать в строку с помощью функции `List.to_string/1`, а строку — в список символов — с помощью функции `String.to_char_list/1`:

```
iex(3)> List.to_string('elixir')
"elixir"
iex(4)> String.to_char_list("elixir")
'elixir'
```

В других случаях, не связанных с использованием библиотек Erlang, старайтесь придерживаться строк. (В главе 6 подробно обсуждаются приемы работы со списками, которые могут пригодиться для

работы с данными, которые желательно интерпретировать как списки, подобные спискам символов.)

Строковые метки

Elixir предлагает еще один способ создания строк, списков символов и регулярных выражений, которые можно применять к двум другим форматам. *Строковые метки* (string sigils) сообщают интерпретатору: «Это содержимое должно иметь указанный тип».

Строковые метки начинаются с символа тильды ~, за которым следует одна из следующих букв: *s* (двоичная строка), *c* (список символов), *r* (регулярное выражение) или *w* (список слов). Если за тильдой следует буква нижнего регистра, интерполяция и экранирование выполняются как обычно. Если за тильдой следует буква верхнего регистра (*S*, *C*, *R* или *W*), строка создается в точности в том виде, в каком она указана, без экранирования и интерполяции. После буквы, помимо двойных кавычек, для ограничения строки можно использовать слеш, квадратные скобки, вертикальную черту (*|*), круглые скобки, фигурные скобки и угловые скобки.

Звучит сложно, но в действительности все просто. Например, если требуется создать строку, содержащую экранированные последовательности для обработки другими инструментами, ее можно записать так:

```
iex(1)> pass_through = ~S"This is a {#msg}, she said.\n This is only a {#msg}."
"This is a {#msg}, she said.\n This is only a {#msg}."
iex(2)> IO.puts(pass_through)
This is a {#msg}, she said.\n This is only a {#msg}.
:ok
```

С помощью символов *w* и *W* создаются списки слов. Эта метка принимает строку и разбивает ее на список слов по пробельным символам:

```
iex(3)> ~w/Elixir is great!/
["Elixir", "is", "great!"]
```



Можно также создать свою строковую метку, для поддержки своего уникального формата. Описание этой и многих других возможностей вы найдете на веб-сайте Elixir (<https://github.com/elixir-lang/elixir/releases>)¹.

¹ Аналогичное описание на русском языке можно найти по адресу: <https://elixirschool.com/ru/lessons/basics/sigils/>. – Прим. перев.

Запрос информации у пользователя

Многие приложения на Elixir действуют подобно оптовым продавцам – незаметно для покупателей предоставляя товары и услуги розничным продавцам, взаимодействующим с покупателями напрямую. Однако иногда требуется написать интерфейсный код, чуть более удобный, чем командная строка IEEx. Маловероятно, что вам придется писать приложения на Elixir, основным интерфейсом которых является командная строка, но этот интерфейс может очень даже пригодиться для отладки вашего кода. (Коль скоро вы работаете с Elixir, высока вероятность, что вы не будете иметь ничего против интерфейса командной строки.)

Вы *можете* смешивать операции ввода/вывода с основной логикой программы, но вообще подобного рода фасад принято помещать в отдельный модуль. В данном случае мы напомним модуль `Ask`, который будет работать с модулем `Drop` из примера 3.8.

Ввод символов

Функция `IO.getn` дает возможность принять несколько символов от пользователя. Она может пригодиться, например, когда имеется список вариантов: вы выводите этот список на экран и ждете ввода пользователя. В примере 5.1 (*ch05/ex1-ask*) роль такого списка вариантов играет список планет, пронумерованных от 1 до 3. То есть требуется ввести только один символ.

Пример 5.1 ❖ Вывод меню и ожидание ввода одного символа

```
defmodule Ask do
```

```
  def chars() do
    IO.puts("""
    Which planemo are you on?
      1. Earth
      2. Moon
      3. Mars
    """)

    IO.getn("Which? > ")
```

```
  end
end
```

Большую часть кода занимает само меню. Ключевым здесь является вызов функции `IO.getn` в конце. В первом аргументе ей передается строка приглашения к вводу, а во втором – количество символов для ввода.

Второй аргумент имеет значение по умолчанию 1. Функция позволяет пользователю ввести все, что ему вздумается, и ждет нажатия клавиши **Enter**, но вернет только один первый символ (или указанное во втором аргументе количество первых символов) в виде строки:

```
iex(1)> c("ask.ex")
[Ask]
iex(2)> Ask.chars
Which planemo are you on?
1. Earth
2. Earth's Moon
3. Mars

Which? > 3
"3"
iex(3)>
nil
iex(4)>
```

Функция `IO.getn` вернула строку "3", содержащую символ, введенный пользователем перед нажатием клавиши **Enter**. Однако, как можно судить по значению `nil`, появившемуся ниже, и дополнительному приглашению к вводу интерактивной оболочки, нажатие клавиши **Enter** обрабатывается оболочкой `IEEx`. Это может приводить к странным результатам, если пользователь введет больше символов, чем требуется:

```
iex(5)> Ask.chars
Which planemo are you on?
1. Earth
2. Earth's Moon
3. Mars

Which? > 23456
"2"
iex(6)> 3456
3456
iex(7)>
```

Иногда программе требуется именно такое поведение функции `IO.getn`, но чаще, по крайней мере при работе в командной оболочке `IEEx`, желательно прочитать всю введенную строку и выбрать из нее то, что необходимо.

Чтение строк текста

`Elixir` предлагает несколько разных функций, которые приостанавливают работу программы в ожидании ввода пользователя. Функция

`IO.gets` ждет, пока пользователь закончит ввод строки, заканчивающейся символом перевода строки. Получив такую строку, вы можете извлечь из нее то, что требуется, не оставив в буфере ввода ничего лишнего. В примере 5.2 (*ch05/ex2-ask*) показано, как пользоваться этой функцией, однако извлечение информации оказалось немного сложнее в реализации, чем нам хотелось бы.

Пример 5.2 ❖ Ввод строки, полученной от пользователя

```
defmodule Ask do

  def line() do
    planemo=get_planemo()
    distance=get_distance()
    Drop.fall_velocity({planemo, distance})
  end

  defp get_planemo() do
    IO.puts("""
    Which planemo are you on?
      1. Earth
      2. Earth's Moon
      3. Mars
    """)

    answer = IO.gets("Which? > ")
    value=String.first(answer)
    char_to_planemo(value)
  end

  defp get_distance() do
    input = IO.gets("How far? (meters) > ")
    value = String.strip(input)
    String.to_integer(value)
  end

  defp char_to_planemo(char) do
    case char do
      "1" -> :earth
      "2" -> :moon
      "3" -> :mars
    end
  end
end
```

Функция `line` просто вызывает три другие функции. Она вызывает `get_planemo`, чтобы вывести меню и получить ответ пользователя, затем вызывает `get_distance`, чтобы узнать у пользователя высоту

падения. Далее она вызывает `Drop.fall_velocity`, чтобы получить и вернуть скорость объекта, падающего в пустоте, в момент достижения поверхности после сброса с указанной высоты на заданной планете.

Функция `get_planemo` использует `IO.puts` и многострочную строку для представления информации, и вызывает `IO.gets`, чтобы получить информацию от пользователя. В отличие от `IO.getn`, функция `IO.gets` возвращает всю строку, введенную пользователем, включая символ перевода строки, и ничего не оставляет в буфере ввода:

```
defp get_planemo() do
  IO.puts("""
  Which planemo are you on?
  1. Earth
  2. Earth's Moon
  3. Mars
  """)

  answer = IO.gets("Which? > ")
  value=String.first(answer)
  char_to_planemo(value)
end
```

Последние две строки обрабатывают результат. Данное приложение интересует только первый символ из введенной строки. Самый простой путь извлечь его – воспользоваться встроенной функцией `String.first`, возвращающей первый символ из строки. Функция `Drop.fall_velocity()` не распознает планеты по номерам 1, 2 или 3; ей нужен атом `:earth`, `:moon` или `:mars`. Поэтому функция `get_planemo` завершается вызовом функции `char_to_planemo()`, выполняющей требуемое преобразование:

```
defp char_to_planemo(char) do
  case char do
    "1" -> :earth
    "2" -> :moon
    "3" -> :mars
  end
end
```

Инструкция `case` сопоставляет аргумент с предопределенными строками. Атом, возвращаемый инструкцией `case`, становится возвращаемым значением функции `get_planemo/0`, которое, в свою очередь, становится возвращаемым значением функции `line/0` и используется в дальнейших вычислениях.

Получить высоту падения немного проще:

```
defp get_distance() do
  input = IO.gets("How far? (meters) > ")
  value = String.strip(input)
  String.to_integer(value)
end
```

Переменная `input` принимает ответ пользователя на вопрос «How far?» (Как высоко?). Затем, с помощью `String.strip`, из `input` удаляются все пробельные символы в начале и в конце, в том числе символ перевода строки. В заключение вызывается функция `String.to_integer`, которая извлекает целочисленное значение из строки `value`. Использование функции `String.to_integer` – не лучший выбор, но в данном случае это вполне приемлемо.

Следующий пример демонстрирует, что описанный выше код производит правильные результаты для правильного ввода:

```
iex(1)> c("ask.ex")
[Ask]
iex(2)> c("drop.ex")
[Drop]
iex(3)> Ask.line
Which planemo are you on?
1. Earth
2. Earth's Moon
3. Mars

Which? > 1
How far? (meters) > 20
19.79898987322333
iex(4)> Ask.line
Which planemo are you on?
1. Earth
2. Earth's Moon
3. Mars

Which? > 2
How far? (meters) > 20
8.0
```

В главе 10 мы еще вернемся к этому коду и познакомимся с приемами обработки ошибок в случаях, когда пользователь вводит неожиданные данные.



Функция `get_planemo()` работает, но ее можно записать в более идиоматической форме. Во-первых, инструкцию `case` можно заменить тремя однострочными функциями:

```
defp char_to_planemo("1"), do: :earth
defp char_to_planemo("2"), do: :moon
defp char_to_planemo("3"), do: :mars
```

Кроме того, вместо промежуточных переменных в `get_planemo` и `get_distance/1` можно использовать оператор конвейера `|>`:

```
defp get_planemo() do
  IO.puts("""elixir enum.unzip
  Which planemo are you on?
    1. Earth
    2. Earth's Moon
    3. Mars
  """)

  IO.gets("Which? > ")
  |> String.first()
  |> char_to_planemo()
end

defp get_distance() do
  IO.gets("How far? (meters) > ")
  |> String.strip()
  |> String.to_integer()
end
```

Эта идиоматическая версия хранится в каталоге `ch05/ex3-ask`. Если вы считаете неудобным писать однострочные функции или использовать оператор `|>`, то можете этого не делать, но должны знать и уметь читать подобный синтаксис, потому что он часто будет встречаться вам в программах других разработчиков.

Глава 6

Списки

Язык Elixir имеет превосходную поддержку списков, длинных последовательностей однотипных (или нет) значений. Встроенная поддержка списков помогает оценить достоинства рекурсии и позволяет выполнить большой объем работы с минимальными усилиями.

Основы списков

Списки в Elixir – это упорядоченные коллекции элементов. Обычно списки обрабатываются по порядку, от первого элемента (*головы списка*) до последнего, однако имеется возможность извлечь из списка конкретный элемент. В Elixir имеются также встроенные функции для работы со списками, когда не требуется выполнять обхода всей последовательности элементов.

Списки в Elixir заключаются в квадратные скобки, а элементы отделяются друг от друга запятыми. Ниже показан пример списка чисел:

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

Элементы могут иметь любой тип, включая числа, атомы, кортежи, строки и другие списки. Проще всего работать со списками, которые содержат однотипные элементы, но сам язык Elixir не накладывает ограничений на возможность смешивания в списках самых разных значений. Также он не ограничивает количества элементов в списке – единственным ограничением является объем доступной памяти.

Списки поддерживают сопоставление с образцом, так же как другие структуры данных в Elixir:

```
iex(1)> [1, x, 4, y] = [1, 2, 4, 8]
[1, 2, 4, 8]
iex(2)> x
```

```
2  
iex(3)> y  
8
```



Для хранения структур с разнотипными данными, в которых элементы следуют в определенном порядке, обычно принято использовать кортежи, а списки, как правило, применяют для хранения последовательностей однотипных структур неопределенной длины. Кортежи предполагают хранение данных в определенном порядке и могут также содержать списки, то есть если у вас есть структура, включающая один-два расширяемых элемента, используйте для их хранения списки.

Списки могут содержать другие списки, что иногда может приводить к неожиданным результатам. Если, к примеру, вы захотите добавить содержимое одного списка в другой список, дело может закончиться добавлением еще одного уровня вложенности:

```
iex(4)> insert = [2, 4, 8]  
[2, 4, 8]  
iex(5)> full = [1, insert, 16, 32]  
[1, [2, 4, 8], 16, 32]
```

Эту проблему (если это действительно проблема) можно исправить вызовом функции `List.flatten/1`:

```
iex(6)> neat = List.flatten(full)  
[1, 2, 4, 8, 16, 32]
```

Это также означает, что если вы хотите сложить списки, вы должны решить, что должно получиться в результате: список списков или один список, включающий содержимое добавляемых списков. Чтобы создать список списков, достаточно просто вставить списки в список:

```
iex(7)> a = [1, 2, 4]  
[1, 2, 4]  
iex(8)> b = [8, 16, 32]  
[8, 16, 32]  
iex(9)> list_of_lists = [a, b]  
[[1, 2, 4], [8, 16, 32]]
```

Создать единственный плоский список из нескольких списков можно с помощью функции `Enum.concat/2` или эквивалентного ей оператора `++`:

```
iex(10)> combined = Enum.concat(a, b)  
[1, 2, 4, 8, 16, 32]  
iex(11)> combined2 = a ++ b  
[1, 2, 4, 8, 16, 32]
```

Обе операции дают один и тот же результат: общий, плоский список.



Оператор `++` является правоассоциативным, то есть при объединении нескольких списков их порядок в результате может отличаться от порядка следования в коде.

Если требуется объединить несколько списков, это можно сделать с помощью функции `Enum.concat/1`, которая принимает список списков и возвращает единый список с их содержимым:

```
iex(12)> c = [64, 128, 256]
[64, 128, 256]
iex(13)> combined3 = Enum.concat([a, b, c])
[1, 2, 4, 8, 16, 32, 64, 128, 256]
```

Деление списков на головы и хвосты

Списки дают удобный способ хранить большие массивы однотипных данных, но главное их достоинство в языке Elixir – простота использования в рекурсивных алгоритмах. Списки естественно укладываются в модель рекурсии с обратным отсчетом, о которой рассказывалось в разделе «Обратный отсчет» в главе 4: можно продолжать обрабатывать список рекурсивно до исчерпания элементов. Многие языки, реализующие поддержку итераций по спискам, позволяют определить количество элементов в списке и выполнять их обход последовательно. В Elixir применяется иной подход, позволяющий обработать первый элемент в списке, *голову*, и извлечь остальную часть списка, *хвост*, чтобы его можно было передать в следующий, рекурсивный вызов.

Извлечение головы и хвоста производится с помощью механизма сопоставления, путем использования специального синтаксиса слева:

```
[head | tail] = [1,2,4]
```

Две переменные, разделенные вертикальной чертой (`|`), или оператором *соединения* (`cons`), и образующие конструктор списка, будут связаны с головой и хвостом списка справа. В консоли Elixir просто выведет содержимое списка справа, а не фрагменты, созданные механизмом сопоставления, но если попробовать выполнить обход списка, вы увидите истинные результаты:

```
iex(1)> list = [1, 2, 4]
[1, 2, 4]
iex(2)> [h1 | t1] = list
[1, 2, 4]
```

```

iex(3)> h1
1
iex(4)> t1
[2, 4]
iex(5)> [h2 | t2] = t1
[2, 4]
iex(6)> h2
2
iex(7)> t2
[4]
iex(8)> [h3 | t3] = t2
[4]
iex(9)> h3
4
iex(10)> t3
[]
iex(11)> [h4 | t4] = t3
** (MatchError) no match of right hand side value: []

```

Строка 2 разбила первоначальный список на две меньшие части. Переменная `h1` получила значение первого элемента списка, а переменная `t1` — часть списка, *кроме* первого элемента. Строка 5 повторила операцию с меньшим списком, разбив `t1` на `h2` и `t2`. В `t2` все еще хранится список, как показано в строке 7, но он содержит единственный элемент. Строка 8 разбивает этот одноэлементный список, помещая значение единственного элемента в `h3` и *пустой* список в `t3`.

Что получится, если попытаться разбить пустой список, как показано в строке 11? Elixir сообщит об ошибке, "no match..." (отсутствует соответствие с правой стороны). К счастью, это не означает, что рекурсивный обход списков неизбежно будет наткаться на ошибку. Отсутствие соответствия естественным образом остановит рекурсивный процесс, то есть именно то, что чаще всего требуется.



Операция извлечения головы и хвоста возвращает первый и оставшиеся элементы списка, и вы можете использовать эту особенность для обхода списка в прямом направлении. Если вам действительно потребуется выполнить обход списка в обратном направлении, используйте функцию `Enum.reverse`, чтобы перевернуть список, и затем выполните обход перевернутого списка.

Обработка содержимого списков

Нотация голова–хвост была специально создана для рекурсивной обработки. Фактически такой стиль работы следует шаблону, в котором список принимается как аргумент и затем передается другой (обыч-

но приватной) функции с аргументом-аккумулятором. Простейшим примером могут служить вычисления на основе содержимого списка. В примере 6.1 (*ch06/ex1-product*) демонстрируется практическая реализация этого шаблона, где выполняется перемножение элементов списка.

Пример 6.1 ❖ Вычисление произведения значений в списке

```
defmodule Overall do
  def product([]) do
    0
  end

  def product(list) do
    product(list, 1)
  end

  def product([], accumulated_product) do
    accumulated_product
  end

  def product([head | tail], accumulated_product) do
    product(tail, head * accumulated_product)
  end
end
```

Функция `product/1` в этом модуле является точкой входа. Она передает список (если он не пустой), плюс аккумулятор функции `product/2`, которая выполняет основную работу. Если вам потребуется проверить полученный список, чтобы убедиться, что он соответствует ожиданиям, эту работу лучше проделать в функции `product/1` и позволить функции `product/2` сосредоточиться на рекурсивной обработке.



А что, произведение элементов для пустого списка действительно равно нулю? Возможно, для случая с пустым списком имело бы больше смысла возбудить ошибку. Философия «позволь потерпеть аварию», исповедуемая языком Elixir, с которой вы познакомитесь позже, прекрасно подходит для таких ситуаций. В конечном счете вам решать, какой вариант выбрать – с возбуждением ошибки или без нее.

Функция `product/2` имеет два предложения. Первое соответствует пустому списку и вызывается в конце рекурсивного процесса, когда в списке не осталось элементов для обработки или когда получен пустой список. Оно возвращает свой второй аргумент, аккумулятор.

Второе предложение вызывается, когда получен непустой список. Сначала операция сопоставления (`[head|tail]`) отделяет первый элемент от остальной части списка. Затем снова вызывается `product/2`, которой передаются остаток (если имеется) списка и новый аккумулятор, который умножается на значение первого элемента. В результате получается произведение всех значений в списке:

```
iex(1)> Overall.product([1, 2, 3, 5])
30
```

Вроде бы все правильно, но как действительно протекал процесс? После того как `product/1` вызвала `product/2`, было выполнено пять итераций, включая обработку пустого списка, как показано в табл. 6.1.

Таблица 6.1. Рекурсивная обработка простого списка в `product/2`

Полученный список	Полученное произведение	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)	Нет	Нет

Последнее значение `accumulated_product`, 30, было получено предложением, обрабатывающим пустой список, и возвращено как результат функции `product/2`. Когда `product/1` получила это значение, она также вернула его и завершилась.



Поскольку строки в одиночных кавычках в действительности являются списками, можно проделать интересный трюк: ввести `Overall.product('funny')`. В этом случае функция `product/1` будет интерпретировать коды символов как числовые значения и вернет 17472569400.

Создание списка из головы и хвоста

Иногда обработка списка заключается в выполнении вычислений на основе значений в списке, но нередко требуется изменять или преобразовывать списки. Так как в действительности нельзя изменить сам список, подобные преобразования и модификации подразумевают создание нового списка. С этой целью можно использовать тот же синтаксис голова/хвост с вертикальной чертой, но уже с правой стороны от оператора сопоставления, а не с левой. Вы можете попробовать этот прием в консоли:

```
iex(1)> x = [1 | [2, 3]]
[1, 2, 3]
```

Elixir интерпретирует `[1|[2,3]]` как операцию создания нового списка. Если значение справа от вертикальной черты является списком, голова будет добавлена перед ним. В данном случае получается простой список чисел. Существуют и другие формы, которые вы должны знать:

```
iex(2)> y = [1, 2 | [3]]
[1, 2, 3]
iex(3)> z = [1, 2 | 3]
[1, 2 | 3]
```

В строке 2 первые два элемента в голове не заключены в список, но конструктор благополучно объединил голову и хвост. (Если эти два элемента заключить в квадратные скобки, конструктор списка посчитает, что первым элементом нового списка должен быть список, то есть `[[1,2] | [3]]` даст в результате `[[1,2],3]`.)

Строка 3 демонстрирует, что случается, когда хвост не является списком (не заключен в квадратные скобки), – в результате получается список, который называется *неправильным списком* (*improper list*) и содержит конструктор со странным хвостом. Пока вы не овладели языком Elixir в достаточной мере, старайтесь избегать подобных ситуаций, так как попытки обрабатывать подобные списки приводят к ошибкам времени выполнения. Кроме того, в практике очень редко встречаются ситуации, требующие создания неправильных списков.

Чаще всего конструкторы списков используются в рекурсивных функциях. Пример 6.2 (*ch06/ex2-drop*) создает несколько кортежей, представляющих планеты и высоты. С помощью модуля `Drop` из примера 3.8 он создает список скоростей для соответствующих ситуаций.

Пример 6.2 ❖ Вычисление серии скоростей падения, с ошибкой

```
defmodule ListDrop do
  def falls(list) do
    falls(list, [])
  end

  def falls([], results) do
    results
  end
end
```

```
def falls([head|tail], results) do
  falls(tail, [Drop.fall_velocity(head) | results])
end

end
```

Большая часть этого кода должна показаться вам знакомой, кроме того что переменная `results` получает список вместо числа и последняя строка в `falls/2` создает список вместо единственного значения. Однако если попробовать выполнить этот пример, вы увидите одну небольшую проблему:

```
$ iex -S mix
Erlang/OTP 19 [erts-8.0] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe]
[kernel-poll:false]

Compiling 2 files (.ex)
Generated list_drop app
Interactive Elixir (1.3.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>> ListDrop.falls([{:earth, 20}, {:moon, 20}, {:mars, 20}])
[12.181953866272849, 8.0, 19.79898987322333]
```

Полученные скорости следуют в обратном порядке: ускорение свободного падения на Земле больше, чем на Марсе, соответственно, на Земле объект должен набрать более высокую скорость. Что же произошло? Проблема кроется в последней функции `falls/2`: она читает список от начала до конца и создает новый список от конца до начала. Она помещает значения в список в противоположном порядке. К счастью, как демонстрирует пример 6.3 (*ch06/ex3-drop*), эту проблему легко исправить. Достаточно вызвать `Enum.reverse/1` в предложении функции `falls/2`, обрабатывающем пустой список.

Пример 6.3 ❖ Вычисление серии скоростей падения, с исправленной ошибкой

```
defmodule ListDrop do

  def falls(list) do
    falls(list, [])
  end

  def falls([], results) do
    Enum.reverse(results)
  end

  def falls([head|tail], results) do
    falls(tail, [Drop.fall_velocity(head) | results])
  end

end
```

Теперь все работает правильно:

```
iex(2)> r(ListDrop)
warning: redefining module ListDrop (current version loaded from
  _build/dev/lib/list_drop/ebin/Elixir.ListDrop.beam)
  lib/list_drop.ex:1
iex(3)> ListDrop.falls([{:earth, 20}, {:moon, 20}, {:mars, 20}])
[19.79898987322333, 8.0, 12.181953866272849]
```



Вызов `Enum.reverse/1` можно также вставить в функцию `falls/1`, играющую роль точки входа. В любом случае будет получен правильный результат, однако мы предпочли бы вернуть правильный результат из `falls/2`.

Смешивание списков и кортежей

По мере погружения в язык Elixir и используя все более сложные структуры данных, вы можете заметить, что все чаще обрабатываете списки кортежей или что удобнее было бы превратить два списка в один, содержащий кортежи, или наоборот. Модуль `Enum` включает простое решение для преобразований такого рода.

Простейшими инструментами являются функции `Enum.zip/2` и `Enum.unzip/1`. Первая превращает два списка одинакового размера в список кортежей, а вторая – список кортежей в два списка:

```
iex(1)> list1 = ["Hydrogen", "Helium", "Lithium"]
["Hydrogen", "Helium", "Lithium"]
iex(2)> list2 = ["H", "He", "Li"]
["H", "He", "Li"]
iex(3)> element_list = Enum.zip(list1, list2)
[{"Hydrogen", "H"}, {"Helium", "He"}, {"Lithium", "Li"}]
iex(4)> separate_lists = Enum.unzip(element_list)
[["Hydrogen", "Helium", "Lithium"], ["H", "He", "Li"]]
```

Два списка, `list1` и `list2`, имеют разное содержимое, но одинаковое количество элементов. Функция `List.zip/1` возвращает список кортежей, содержащих соответствующие пары элементов из исходных списков. Функция `List.unzip/1` принимает этот список 2-элементных кортежей и преобразует его в кортеж с двумя списками.

Создание списка списков

Простая рекурсия реализуется относительно просто, но в процессе обработки списков нередко возникает необходимость преобразовывать их в списки списков. Треугольник Паскаля – классический математический инструмент – относительно легко создать, но он де-

монстрирует широкие возможности, доступные при работе со списками. На вершине треугольника находится число 1, каждая следующая строка содержит суммы двух чисел выше:

```

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

```

Если эти числа покажутся вам знакомыми, то, скорее всего, потому, что они являются биномиальными коэффициентами, которые используются в разложении степени суммы $(x+y)$. Это только начало математического чуда!

Треугольник Паскаля легко вычисляется на языке Elixir. Для этого, например, можно использовать приемы работы со списками, обсуждавшиеся в этой главе, интерпретируя каждый ряд как список, а весь треугольник – как список списков. Первый ряд – с 1 в вершине – можно представить в виде списка $[0, 1, 0]$. Дополнительные нули упрощают вычисление сумм.



Следующий пример не является самой красивой, компактной и эффективной реализацией. Его главная цель – объяснить некоторые дополнительные приемы работы со списками.

На первом шаге пример 6.4 (*ch06/ex4-pascal*) вычисляет ряды по отдельности. Этот простой рекурсивный процесс выполняет обход предыдущего списка и на его основе создает новый список.

Пример 6.4 ❖ Вычисление строки

```

defmodule Pascal do

  def add_row(initial) do
    add_row(initial, 0, [])
  end

  def add_row([], 0, final) do
    [0 | final]
  end

  def add_row([h | t], last, new) do
    add_row(t, h, [last + h | new])
  end

end

```

Функция `add_row/1` запускает процесс, посылая текущую строку, 0 и пустой список, который можно рассматривать как контейнер для

результатов, то есть как аккумулятор. Функция `add_row/3` имеет два предложения. Первое вызывается, когда список со слагаемыми пустой и возвращает список с результатом, в начало которого добавляется `0`.

Основная работа выполняется во втором предложении функции `add_row/3`. Принимая аргументы, оно использует шаблон `[h | t]`, чтобы сохранить голову списка в `h` (число) и хвост в `t` (список, который может быть пустым, если это было последнее число в списке). Оно также принимает значение последнего обработанного числа `last` и конструируемый новый список `new`.

Оно рекурсивно вызывает `add_row/3`. В этом новом вызове хвост `t` становится старым списком (списком с числами для суммирования); значение `h` становится числом `last`; и третьим аргументом – новый список, начинающийся найденной суммой и завершающийся остальной частью конструируемого списка `new`.



Так как списки в треугольнике симметричны, нет необходимости переворачивать их с помощью `Enum.reverse/1`. Впрочем, вы можете сделать это, если пожелаете.

Этот код легко проверить в консоли, но не забывайте, что в тестовые списки следует добавлять нули по краям:

```
iex(1)> Pascal.add_row([0, 1, 0])
[0, 1, 1, 0]
iex(2)> Pascal.add_row([0, 1, 1, 0])
[0, 1, 2, 1, 0]
iex(3)> Pascal.add_row([0, 1, 2, 1, 0])
[0, 1, 3, 3, 1, 0]
```

Теперь, получив возможность создавать новые ряды из старых, реализуем создание набора рядов, начиная с вершины треугольника, как показано в примере 6.5 (*ch06/ex4-pascal*). Функция `add_row/3` фактически реализует прием обратного отсчета до конца списка, а `triangle/3` должна выполнить прямой отсчет до заданного числа рядов. Функция `triangle/1` запускает вычисления, определяя начальный ряд, устанавливая счетчик равным 1 (потому что начальный ряд является первым) и передавая количество рядов для создания.

Функция `triangle/3` имеет два предложения. Первое останавливает рекурсию после создания достаточного количества рядов и переворачивает список. (Отдельные ряды в нашем случае симметричны, но сам треугольник – нет.) Второе предложение выполняет основную работу по созданию новых рядов. Оно извлекает предыдущий ряд (`previous`) из списка `list` и передает его функции `add_row/1`, которая возвращает новый ряд. Затем оно вызывает себя с новым списком,

увеличивает счетчик `count`, и так, пока не будет вызвано предложение, останавливающее рекурсию.

Пример 6.5 ❖ Вычисление полного треугольника Паскаля

```
defmodule Pascal do

  def triangle(rows) do
    triangle([[0,1,0]], 1, rows)
  end

  def triangle(list, count, rows) when count >= rows do
    Enum.reverse(list)
  end

  def triangle(list, count, rows) do
    [previous | _] = list
    triangle([add_row(previous) | list], count + 1, rows)
  end

  def add_row(initial) do
    add_row(initial, 0, [])
  end

  def add_row([], 0, final) do
    [0 | final]
  end

  def add_row([h | t], last, new) do
    add_row(t, h, [last + h | new])
  end
end
```

Этот пример успешно работает:

```
iex(4)> r(Pascal)
warning: redefining module Pascal (current version loaded from
  _build/dev/lib/pascal/ebin/Elixir.Pascal.beam)
  lib/pascal.ex:1

{:reloaded, Pascal, [Pascal]}
iex(5)> Pascal.triangle(4)
[[0, 1, 0], [0, 1, 1, 0], [0, 1, 2, 1, 0], [0, 1, 3, 3, 1, 0]]
iex(6)> 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 является одним из лучших представителей списков, состоящих из списков, с которыми вам придется работать, но сам подход к обработке подобных многоуровневых списков, продемонстрированный здесь, является распространенной тактикой обработки и создания списков данных.

Глава 7

Пары имя/значение

Кортежи и списки – мощные инструменты для создания сложных структур данных, но в нашем рассказе отсутствуют еще два ключевых элемента. Кортежи, по сути, являются анонимными структурами. Зависимость от конкретного порядка следования и количества элементов в кортежах может создавать немалые проблемы. Списки страдают похожими проблемами: типичные приемы обработки списков в Elixir опираются на тот факт, что списки являются обычными последовательностями (часто) однотипных элементов.

Но иногда возникает желание обращаться к элементам данных по именам, а не по номерам или шаблонам, определяющим их местоположение. Для этого в Elixir имеется много разных возможностей.



Отображения и структуры появились на последних этапах развития Elixir. Они основаны непосредственно на особенностях Erlang, появившихся в версии R17. С течением времени отображения и структуры, возможно, займут лидирующее положение, но вам может также понадобиться сохранить совместимость со старым кодом на языке Erlang.

Списки ключей

Иногда возникает потребность обрабатывать списки двухэлементных кортежей, которые можно рассматривать как списки пар «ключ и значение», где ключ является атомом. Elixir отображает их в формате *списков ключей*, и вы также можете вводить их в этом формате:

```
iex(1)> planemo_list = [{:earth, 9.8}, {:moon, 1.6}, {:mars, 3.71}]
[earth: 9.8, moon: 1.6, mars: 3.71]
iex(2)> atomic_weights = [hydrogen: 1.008, carbon: 12.011, sodium: 22.99]
[hydrogen: 1.008, carbon: 12.011, sodium: 22.99]
iex(3)> ages = [david: 59, simon: 40, cathy: 28, simon: 30]
[david: 59, simon: 40, cathy: 28, simon: 30]
```


Список ключей всегда хранит данные последовательно и может содержать повторяющиеся ключи. Модуль `Keyword` в Elixir позволяет читать, удалять и вставлять значения по их ключам.

Функция `Keyword.get/3` извлекает значение из списка по заданному ключу. В третьем необязательном аргументе можно передать функции `Keyword.get` значение по умолчанию, которое будет возвращено в случае отсутствия указанного ключа в списке. Функция `Keyword.fetch!/2` возбуждает ошибку, если ключ не найден. Функция `Keyword.get_values/2` вернет все значения для заданного ключа:

```
iex(4)> Keyword.get(atomic_weights, :hydrogen)
1.008
iex(5)> Keyword.get(atomic_weights, :neon)
nil
iex(6)> Keyword.get(atomic_weights, :carbon, 0)
12.011
iex(7)> Keyword.get(atomic_weights, :neon, 0)
0
iex(8)> Keyword.fetch!(atomic_weights, :neon)
** (KeyError) key :neon not found in:
    [hydrogen: 1.008, carbon: 12.011, sodium: 22.99]
    (elixir) lib/keyword.ex:312: Keyword.fetch!/2
iex(8)> Keyword.get_values(ages, :simon)
[40,30]
```

Функция `Keyword.has_key?/2` позволяет проверить наличие ключа в списке:

```
iex(9)> Keyword.has_key?(atomic_weights, :carbon)
true
iex(10)> Keyword.has_key?(atomic_weights, :neon)
false
```

Функция `Keyword.put_new/3` добавляет новое значение. Если ключ уже существует, его значение остается без изменений:

```
iex(11)> weights2 = Keyword.put_new(atomic_weights, :helium, 4.0026)
[helium: 4.0026, hydrogen: 1.008, carbon: 12.011, sodium: 15.999]
iex(12)> weights3 = Keyword.put_new(weights2, :helium, -1)
[helium: 4.0026, hydrogen: 1.008, carbon: 12.011, sodium: 22.99]
```

Функция `Keyword.put/3` изменяет значение существующего ключа. Если ключ отсутствует, он будет создан. Если ключ существует, функция удалит все записи с этим ключом и добавит в список новую запись:

```
iex(13)> ages2 = Keyword.put(ages, :chung, 19)
[chung: 19, david: 59, simon: 40, cathy: 28, simon: 30]
```

```
iex(14)> ages3 = Keyword.put(ages2, :simon, 22)
[simon: 22, chung: 19, david: 59, cathy: 28]
```



Все эти функции копируют списки или создают новые, модифицированные версии списка. Исходный список, как можно догадаться, остается без изменений.

Функция `Keyword.delete/2` удаляет все записи с указанным ключом. Удалить только первую запись можно с помощью `Keyword.delete_first/2`:

```
iex(15)> ages2
[chung: 19, david: 59, simon: 40, cathy: 28, simon: 30]
iex(16)> ages4 = Keyword.delete(ages2, :simon)
[chung: 19, david: 59, cathy: 28]
```

Списки кортежей с несколькими ключами

Если создать список кортежей с атомными весами, включающих названия элементов и их химические знаки, первый и второй элементы таких кортежей можно использовать как ключи:

```
iex(1)> atomic_info = [{:hydrogen, :H, 1.008}, {:carbon, :C, 12.011},
... (1)> {:sodium, :Na, 22.99}]
[{:hydrogen, :H, 1.008}, {:carbon, :C, 12.011}, {:sodium, :Na, 22.99}]
```

Если у вас имеются данные, структурированные таким способом, для работы с ними можно использовать функции `List.keyfind/4`, `List.keymember?/3`, `List.keyreplace/4`, `List.keystore/4` и `List.keydelete/3`. Все эти функции принимают список в первом аргументе. Во втором аргументе они принимают искомый ключ, а в третьем — индекс элемента в кортеже, который должен использоваться как ключ, где `0` соответствует первому элементу:

```
iex(2)> List.keyfind(atomic_info, :H, 1)
{:hydrogen, :H, 1.008}
iex(3)> List.keyfind(atomic_info, :carbon, 0)
{:carbon, :C, 12.011}
iex(4)> List.keyfind(atomic_info, :F, 1)
nil
iex(5)> List.keyfind(atomic_info, :fluorine, 0, {})
{}
iex(6)> List.keymember?(atomic_info, :Na, 1)
true
iex(7)> List.keymember?(atomic_info, :boron, 0)
false
iex(8)> atomic_info2 = List.keystore(atomic_info, :boron, 0,
```

```

...(8)> {:boron, :B, 10.081})
[{:hydrogen, :H, 1008}, {:carbon, :C, 12.011}, {:sodium, :Na, 22.99},
 {:boron, :B, 10.081}]
iex(9)> atomic_info3 = List.keyreplace(atomic_info2, :B, 1,
...(9)> {:boron, :B, 10.81})
[{:hydrogen, :H, 1008}, {:carbon, :C, 12.011}, {:sodium, :Na, 22.99},
 {:boron, :B, 10.81}]
iex(10)> atomic_info4 = List.keydelete(atomic_info3, :fluorine, 0)
[{:hydrogen, :H, 1008}, {:carbon, :C, 12.011}, {:sodium, :Na, 22.99},
 {:boron, :B, 10.81}]
iex(11)> atomic_info5 = List.keydelete(atomic_info3, :carbon, 0)
[{:hydrogen, :H, 1008}, {:sodium, :Na, 22.99}, {:boron, :B, 10.81}]

```

Строки 2 и 3 демонстрируют поиск по названию химического элемента (позиция 0) или химическому знаку (позиция 1). По умолчанию попытка найти несуществующий ключ возвращает `nil` (строка 4), но есть возможность вернуть любое другое значение по своему выбору (строка 5). Строки 6 и 7 демонстрируют использование `List.member?`.

Чтобы добавить новое значение, необходимо передать в последнем аргументе полный кортеж, как показано в строке 8. Мы умышленно ввели неверное значение атомного веса бора (`boron`). В строке 9 мы исправили эту ошибку с помощью `List.keyreplace`.



Для замены целого кортежа можно также использовать функцию `List.keyreplace`. Заменить запись для бора записью для цинка можно так:

```

iex(9)> atomic_info3 = List.keyreplace(atomic_info2, :B,
...(9)> 1, {:zinc, :Zn, 65.38})

```

Строки 10 и 11 показывают, что происходит при попытке удалить несуществующую и существующую записи вызовом `List.keydelete`.

Словари

Если заранее известно, что ключи будут уникальными, можно использовать *словарь* (`HashDict`), который, по сути, является ассоциативным массивом. Словари не являются списками, но мы включили их в эту главу, потому что все функции для работы со списками ключей в равной степени применимы к словарям `HashDict`. В отличие от списков ключей, словари `HashDict` позволяют работать с огромными объемами данных без потери производительности. Чтобы задействовать словарь, нужно явно создать его вызовом функции `HashDict.new`:

```

iex(1)> planemo_hash = Enum.into([earth: 9.8, moon: 1.6, mars: 3.71],
...(1)> HashDict.new())
#HashDict<[earth: 9.8, mars: 3.71, moon: 1.6]>
iex(2)> HashDict.has_key?(planemo_hash, :moon)
true
iex(3)> HashDict.has_key?(planemo_hash, :jupiter)
false
iex(4)> HashDict.get(planemo_hash, :jupiter)
nil
iex(5)> HashDict.get(planemo_hash, :jupiter, 0)
0
iex(6)> planemo_hash2 = HashDict.put_new(planemo_hash, :jupiter, 99.9)
#HashDict<[moon: 1.6, mars: 3.71, jupiter: 99.9, earth: 9.8]>
iex(7)> planemo_hash3 = HashDict.put_new(planemo_hash2, :jupiter, 23.1)
#HashDict<[moon: 1.6, mars: 3.71, jupiter: 99.9, earth: 9.8]>
iex(8)> planemo_hash4 = HashDict.put(planemo_hash3, :jupiter, 23.1)
#HashDict<[moon: 1.6, mars: 3.71, jupiter: 23.1, earth: 9.8]>
iex(9)> planemo_hash5 = HashDict.delete(planemo_hash4, :saturn)
#HashDict<[moon: 1.6, mars: 3.71, jupiter: 23.1, earth: 9.8]>
iex(10)> planemo_hash6 = HashDict.delete(planemo_hash4, :jupiter)
#HashDict<[moon: 1.6, mars: 3.71, earth: 9.8]>

```

В строке 6 мы преднамеренно указали неправильное ускорение свободного падения для Юпитера. Строка 7 показывает, что функция `HashDict.put_new/2` не обновляет существующего значения; для этой цели следует использовать `HashDict.put`, как показано в строке 8. Строка 9 показывает, что попытка удалить из словаря несуществующий ключ никак не изменяет словарь.



В настоящее время не рекомендуется использовать словари `HashDict`, так как их поддержка будет убрана из версии Elixir 2.0. Используйте вместо них отображения, которые к тому же работают намного быстрее.

От списков к отображениям

Списки ключей дают удобную возможность хранения содержимого в виде списков по ключу, но внутри Elixir все еще вынужден выполнять обход всего списка. В этом нет ничего страшного, если у вас есть планы использовать такие списки как обычные списки, но это влечет ненужное падение производительности, если программа обращается к элементам только по ключам.

После многих лет существования этой проблемы сообщество Erlang, наконец, добавило новый набор инструментов в версию R17:

отображения. (Начальная реализация не отличается полнотой, но неплохо подходит для начала.) Поддержка новых возможностей немедленно была добавлена в Elixir, хотя и с некоторыми особенностями в синтаксисе, характерными для языка Elixir.

Создание отображений

Чтобы создать пустое отображение, достаточно использовать конструкцию `%{}`:

```
iex(1)> new_map = %{  
%{}
```

Нередко, однако, отображения должны создаваться хотя бы с несколькими начальными значениями. Сделать это в Elixir можно двумя способами. Использовать тот же синтаксис `%{}`, с несколькими объявлениями внутри:

```
iex(2)> planemo_map = %{:earth => 9.8, :moon => 1.6, :mars => 3.71}  
%{earth: 9.8, mars: 3.71, moon: 1.6}
```

Это отображение хранит три ключа (атома): `:earth`, `:moon` и `:mars`, — которым соответствуют значения 9.8, 1.6 и 3.71 соответственно. Самым примечательным в этом синтаксисе является возможность использовать в роли ключей значения любых типов. Например, можно использовать числа:

```
iex(3)> number_map=%{2 => "two", 3 => "three"}  
%{2 => "two", 3 => "three"}
```

Однако на практике чаще используются атомы, а кроме того, Elixir предлагает еще более компактный синтаксис создания отображений, с ключами-атомами:

```
iex(4)> planemo_map_alt = %{earth: 9.8, moon: 1.6, mars: 3.71}  
%{earth: 9.8, mars: 3.71, moon: 1.6}
```

На попытки создать отображение в строках 2 и 4 оболочка IEEx выводит одинаковый ответ, и сам Elixir использует более компактный синтаксис, если это возможно.

Изменение отображений

Если понадобится изменить величину ускорения свободного падения для какой-то планеты, это легко сделать, как показано ниже:

```
iex(5)> altered_planemo_map = %{planemo_map | earth: 12}  
%{earth: 12, mars: 3.71, moon: 1.6}
```

или:

```
iex(6)> altered_planemo_map = %{planemo_map | :earth => 12}
%{earth: 12, mars: 3.71, moon: 1.6}
```

При желании можно изменить несколько пар ключ/значение, используя синтаксис `%{planemo_map | earth: 12, mars: 3}` или `%{planemo_map | :earth => 12, :mars => 3}`. Синтаксис с вертикальной чертой (`|`) работает только для существующих ключей. Если ключ отсутствует, Elixir возбudit ошибку `KeyError`.

Вам может также потребоваться добавить в отображение еще одну пару ключ/значение. Изменить само отображение, естественно, не получится, но с помощью библиотечной функции `Map.put_new` легко можно создать новое отображение, включающее исходное отображение, плюс дополнительную пару ключ/значение:

```
iex(7)> extended_planemo_map = Map.put_new(planemo_map, :jupiter, 23.1)
%{earth: 9.8, jupiter: 23.1, mars: 3.71, moon: 1.6}
```

Чтение отображений

Elixir позволяет извлекать информацию из отображений с помощью сопоставления с образцом. Один и тот же синтаксис применяется и к переменным, и к вызовам функций. Требуется узнать ускорение свободного падения на Земле?

```
iex(8)> %{earth: earth_gravity} = planemo_map
%{earth: 9.8, mars: 3.71, moon: 1.6}
iex(9)> earth_gravity
9.8
```

Если попытаться запросить несуществующий ключ, интерпретатор возбudit ошибку. Если вам потребуется образец для сопоставления, соответствующий «любому отображению», просто используйте пустое отображение, `%{}`.

От отображений к структурам

Одним из недостатков кортежей, списков ключей и отображений является их слабая структурированность. При использовании кортежей вы вынуждены запоминать порядок следования элементов данных в них. Списки ключей и отображения позволяют добавить новый ключ в любой момент или ввести ключ с орфографической ошибкой, и Elixir не сообщит об этом. Эти проблемы решают *структуры*. Они основаны на отображениях, поэтому порядок следования пар ключ/

значение в них не поддерживается, но структуры точно знают имена своих ключей и не позволяют использовать недопустимые ключи.

Объявление структур

Чтобы получить возможность использовать структуру, необходимо передать интерпретатору специальное объявление: объявление `defstruct` (фактически это макрос, как вы увидите далее в книге), заключенное в объявление `defmodule`:

```
defmodule Planemo do
  defstruct name: :nil, gravity: 0, diameter: 0, distance_from_sun: 0
end
```

Здесь объявляется структура с именем `Planemo`, содержащая поля `name`, `gravity`, `diameter` и `distance_from_sun` с их значениями по умолчанию. Следующее объявление определяет структуру, представляющую башню для сброса объектов и измерения их скорости:

```
defmodule Tower do
  defstruct location: "", height: 20, planemo: :earth, name: ""
end
```

Создание и чтение экземпляров структур

Найдите представленные выше объявления в каталоге *ch07/ex1-struct* и скомпилируйте их в IEEx. Теперь вы готовы использовать структуры для хранения данных. Как показано ниже, в строке 1, когда новая структура создается с применением пустых фигурных скобок {}, в ней сохраняются значения по умолчанию, но если указать значения, как показано в строке 4, будут использованы они:

```
$ iex -S mix
Erlang/OTP 19 [erts-8.0] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe]
[kernel-poll:false]
```

```
Compiling 2 files (.ex)
```

```
Generated tower app
```

```
Interactive Elixir (1.3.1) - press Ctrl+C to exit (type h()) ENTER for help
```

```
iex(1)> tower1 = %Tower{}
```

```
%Tower{height: 20, location: "", name: "", planemo: :earth}
```

```
iex(2)> tower2 = %Tower{location: "Grand Canyon"}
```

```
%Tower{height: 20, location: "Grand Canyon", name: "", planemo: :earth}
```

```
iex(3)> tower3 = %Tower{location: "NYC", height: 241, name: "Woolworth Building"}
```

```
%Tower{height: 241, location: "NYC", name: "Woolworth Building",
planemo: :earth}
```

```
iex(4)> tower4 = %Tower{location: "Rupes Altat 241", height: 500,
```

```

...(4)> planemo: :moon, name: "Piccolini View"}
%Tower{height: 500, location: "Rupes Altat 241", name: "Piccolini View",
planemo: :moon}
iex(5)> tower5 = %Tower{planemo: :mars, height: 500,
...(5)> name: "Daga Vallis", location: "Valles Marineris"}
%Tower{height: 500, location: "Valles Marineris", name: "Daga Vallis",
planemo: :mars}
iex(6)> tower5.name
"Daga Vallis"

```

Эти определения башен демонстрируют разные способы использования синтаксиса структур для создания переменных, а также для управления значениями по умолчанию:

- строка 1 просто создает `tower1` со значениями по умолчанию – значения элементов этой структуры можно будет изменить позже;
- строка 2 создает `tower2` с собственным значением элемента `location`, но для остальных элементов использует значения по умолчанию;
- строка 3 переопределяет значения по умолчанию для элементов `location`, `height` и `name`, но элемент `planemo` оставляет в неприкосновенности;
- строка 4 переопределяет все значения по умолчанию;
- строка 5 также переопределяет все значения по умолчанию и дополнительно демонстрирует, что *порядок следования пар имя/значение не имеет никакого значения*; Elixir сам расположит их, как надо.

После сохранения значений в структуры вы сможете извлекать их, используя точечную нотацию, как показано в строке 6, которая может показаться знакомой тем, кто имеет опыт использования других языков программирования.

Использование структур в сопоставлениях с образцом

Так как структуры – это отображения, механизм сопоставления с образцом обрабатывает структуры точно так же, как отображения:

```

iex(7)> %Tower{planemo: p, location: where} = tower5
%Tower{height: 500, location: "Valles Marineris", name: "Daga Vallis",
planemo: :mars}
iex(8)> p
:mars
iex(9)> where
"Valles Marineris"

```


Использование структур в функциях

Есть возможность сопоставлять структуры, полученные в виде аргументов. Самый легкий способ – просто сопоставить структуру, как показано в примере 7.1 (*ch07/ex2-struct-match*).

Пример 7.1 ❖ Метод сопоставления полной записи

```
defmodule StructDrop do
```

```
  def fall_velocity(t = %Tower{}) do
    fall_velocity(t.planemo, t.height)
  end

  def fall_velocity(:earth, distance) when distance >= 0 do
    :math.sqrt(2 * 9.8 * distance)
  end

  def fall_velocity(:moon, distance) when distance >= 0 do
    :math.sqrt(2 * 1.6 * distance)
  end

  def fall_velocity(:mars, distance) when distance >= 0 do
    :math.sqrt(2 * 3.71 * distance)
  end
end
```

```
end
```

Здесь используется образец, совпадающий только со структурами `Tower` и помещающий совпавшую структуру в переменную `t`. Затем, так же как в примере 3.8, вызывается функция `fall_velocity/2` с отдельными аргументами для вычислений, но на этот раз в вызове используется синтаксис структур:

```
iex(10)> r(StructDrop)
warning: redefining module StructDrop (current version loaded from
  _build/dev/lib/struct_drop/ebin/Elixir.StructDrop.beam)
  lib/struct_drop.ex:1

{:reloaded, StructDrop, [StructDrop]}
[StructDrop]
iex(11)> StructDrop.fall_velocity(tower5)
60.909769331364245
iex(12)> StructDrop.fall_velocity(tower1)
19.79898987322333
```

Функция `StructDrop.fall_velocity/1`, показанная в примере 7.2, извлекает поле `planemo` и связывает его с переменной `planemo`¹. Ана-

¹ Elixir допускает совпадение имен переменных и полей, как в данном примере.

логично она извлекает поле `height` и связывает его с переменной `distance`. Затем возвращает скорость объекта, сброшенного с этой высоты, – в точности как в предыдущих примерах в книге.

Аналогично сопоставление с образцом можно использовать для извлечения полей из структуры, как показано в примере 7.2 (*ch07/ex3-struct-components*).

Пример 7.2 ❖ Метод извлечения полей структуры сопоставлением с образцом

```
defmodule StructDrop do

  def fall_velocity(%Tower{planemo: planemo, height: distance}) do
    fall_velocity(planemo, distance)
  end

  def fall_velocity(:earth, distance) when distance >= 0 do
    :math.sqrt(2 * 9.8 * distance)
  end

  def fall_velocity(:moon, distance) when distance >= 0 do
    :math.sqrt(2 * 1.6 * distance)
  end

  def fall_velocity(:mars, distance) when distance >= 0 do
    :math.sqrt(2 * 3.71 * distance)
  end

end
```

Вы можете взять любую структуру `Tower` из созданных выше и передать ее этой функции, а она сообщит вам скорость, которую получит объект, сброшенный с вершины указанной башни.

Наконец, можно сопоставить и структуру целиком, и ее отдельные поля. Пример 7.3 (*ch07/ex4-struct-multi*) демонстрирует использование этого смешанного подхода для создания более детального ответа, содержащего не только скорость падения.

Пример 7.3 ❖ Метод одновременного сопоставления структуры целиком и отдельных ее компонентов

```
defmodule StructDrop do

  def fall_velocity(t = %Tower{planemo: planemo, height: distance}) do
    IO.puts("From #{t.name}'s elevation of #{distance} meters on #{planemo},")
    IO.puts("the object will reach #{fall_velocity(planemo, distance)} m/s")
    IO.puts("before crashing in #{t.location}")
  end

end
```

```
def fall_velocity(:earth, distance) when distance >= 0 do
  :math.sqrt(2 * 9.8 * distance)
end

def fall_velocity(:moon, distance) when distance >= 0 do
  :math.sqrt(2 * 1.6 * distance)
end

def fall_velocity(:mars, distance) when distance >= 0 do
  :math.sqrt(2 * 3.71 * distance)
end

end
```

Если передать функции `StructDrop.fall_velocity/1` структуру `Tower`, она извлечет отдельные поля, необходимые для вычислений, и сохранит структуру целиком в переменной `t`, чтобы потом вывести более интересное сообщение:

```
iex(1)> StructDrop.fall_velocity(tower5)
From Daga Vallis's elevation of 500 meters on mars,
the object will reach 60.90976933136424520399 m/s
before crashing in Valles Marineris
:ok

iex(2)> StructDrop.fall_velocity(tower3)
From Woolworth Building's elevation of 241 meters on earth,
the object will reach 68.72845116834803036454 m/s
before crashing in NYC
:ok
```

Добавление поведения в структуры

Elixir позволяет добавлять поведение в структуры (и фактически в любые типы данных) с помощью *протоколов*. Например, вам может понадобиться проверить допустимость структуры. Очевидно, что для разных видов структур проверка будет отличаться. Например, экземпляр структуры `Planemo` можно считать допустимым, если ускорение свободного падения (`gravity`), диаметр (`diameter`) и расстояние до Солнца (`distance_from_sun`) имеют неотрицательные значения. Экземпляр структуры `Tower` можно считать допустимым, если высота (`height`) имеет неотрицательное значение и определена планета (`planemo` не равно `nil`).

Пример 7.4 демонстрирует определение протокола для проверки допустимости. Файлы для этого примера находятся в папке `ch07/ex5-protocol`.

Пример 7.4 ❖ Определение протокола для проверки допустимости структур

```
defprotocol Valid do
  @doc "Возвращает true, если данные можно считать допустимыми"
  def valid?(data)
end
```

Самая интересная строка здесь – это `def valid?(data)`. Фактически это неполное определение функции. Для любого типа данных, допустимость которого вам потребуется проверить, необходимо будет написать законченную функцию с именем `valid?`. Давайте посмотрим, как это сделать, на примере структуры `Planemo`:

```
defmodule Planemo do
  defstruct name: nil, gravity: 0, diameter: 0, distance_from_sun: 0
end

defimpl Valid, for: Planemo do
  def valid?(p) do
    p.gravity >= 0 and p.diameter >= 0 and
    p.distance_from_sun >= 0
  end
end
```

И тут же опробуем новое поведение.

```
iex(1)> p = %Planemo{}
%Planemo{diameter: 0, distance_from_sun: 0, gravity: 0, name: nil}
iex(2)> Valid.valid?(p)
true
iex(3)> p2 = %Planemo{name: :weirdworld, gravity: -2.3}
%Planemo{diameter: 0, distance_from_sun: 0, gravity: -2.3, name: :weirdworld}
iex(4)> Valid.valid?(p2)
false
iex(5)> t = %Tower{}
%Tower{height: 20, location: "", name: "", planemo: :earth}
iex(6)> Valid.valid?(t)
** (Protocol.UndefinedError) protocol Valid not implemented for
    %Tower{height: 20, location: "", name: "", planemo: :earth}
    valid_protocol.ex:1: Valid.impl_for!/1
    valid_protocol.ex:3: Valid.valid?/1
```

Строки 1 и 2 демонстрируют создание и проверку допустимого экземпляра структуры `Planemo`; строки 3 и 4 демонстрируют результаты для недопустимого экземпляра. Строка 6 показывает, что эту проверку невозможно применить к структуре `Tower`, так как функция `valid?` для нее еще не реализована. Ниже приводится дополнительный код для структуры `Tower` (*ch07/ex6-protocol*):

```
defmodule Tower do
  defstruct location: "", height: 20, planemo: :earth, name: ""
end

defimpl Valid, for: Tower do
  def valid?(%Tower{height: h, planemo: p}) do
    h >= 0 and p != nil
  end
end
```

А вот результаты проверки:

```
iex(6)> r(Tower)
warning: redefining module Tower (current version loaded from
  _build/dev/lib/valid/ebin/Elixir.Tower.beam)
lib/tower.ex:1

{:reloaded, Tower, [Tower]}
iex(7)> Valid.valid?(t)
true
iex(8)> t2 = %Tower{height: -2, location: "underground"}
%Tower{height: -2, location: "underground", name: "", planemo: :earth}
iex(9)> Valid.valid?(t2)
false
```

Расширение существующих протоколов

Используя `inspect` для исследования содержимого `Tower`, вы получаете обобщенный вывод:

```
iex(10)> t3 = %Tower{location: "NYC", height: 241, name: "Woolworth Building"}
%Tower{height: 241, location: "NYC", name: "Woolworth Building",
  planemo: :earth}
iex(11)> inspect t3
"%Tower{height: 241, location: \"NYC\", name: \"Woolworth Building\",
  planemo: :earth}"
```

Хотели бы улучшить вывод? Это возможно, если реализовать протокол `Inspect` для структур `Tower`. В примере 7.5 приводится код, который нужно добавить в *tower.ex* (в папке *ch07/ex7-inspect*).

Пример 7.5 ❖ Реализация протокола `Inspect` для структуры `Tower`

```
defimpl Inspect, for: Tower do
  import Inspect.Algebra
  def inspect(item, _options) do
    metres = concat(to_string(item.height), "m:")
    msg = concat([metres, break, item.name, ",", break,
```

```

    item.location, ",", break,
    to_string(item.planemo)])
end
end

```

Модуль `Inspect.Algebra` реализует «красивую печать», используя алгебраический подход (отсюда такое необычное название модуля). В простейшем случае он позволяет объединить с помощью `concat` документы, которые могут отделяться друг от друга разрывами строк (`break`). Все разрывы строк в документе будут замещены пробелами или разрывом строки, если для следующего элемента окажется недостаточно места в конце строки.

Функция `inspect/2` принимает в первом аргументе элемент для исследования. Во втором аргументе можно передать структуру с параметрами, позволяющими управлять формированием вывода функцией `inspect/2`.

Первый вызов `concat` объединяет высоту и сокращенное обозначение для метров без пробела между ними. Второй вызов `concat` объединяет все элементы в список, благодаря чему функция возвращает отформатированный документ. Так как `Valid` – это протокол, а не модуль, мы должны скомпилировать файл:

```

iex(12)> c("lib/valid.ex")
warning: redefining module Valid (current version loaded from
  _build/dev/consolidated/Elixir.Valid.beam)
  lib/valid.ex:1

[Valid]
iex(13)> inspect t3
"241m: Woolworth Building, NYC, earth"

```

Глава 8



Функции высшего порядка и генераторы списков

Функции высшего порядка (higher-order functions), или функции, принимающие в аргументах другие функции, помогают языку Elixir проявиться во всем его блеске. Нельзя сказать, что в других языках невозможно реализовать подобные функции, — это возможно почти в любых языках, но в Elixir функции высшего порядка интерпретируются как естественная часть языка, а не как искусственное и чужеродное образование.

Простые функции высшего порядка

Вернемся к главе 2 и вспомним, как мы использовали `fn` для создания функций:

```
iex(1)> fall_velocity = fn(distance) -> :math.sqrt(2 * 9.8 * distance) end
#Function<6.106461118/1 in :erl_eval.expr/5>
iex(2)> fall_velocity.(20)
19.79898987322333
iex(3)> fall_velocity.(200)
62.609903369994115
```

Elixir позволяет не только связывать функции с переменными, но и передавать их в аргументах другим функциям. Это означает, что можно создавать функции, поведение которых может изменяться в зависимости от параметров вызова, причем намного более причудливо, чем при использовании обычных параметров. В примере 8.1

(*ch08/ex1-hof*) представлена очень простая функция высшего порядка, принимающая другую функцию в аргументе.

Пример 8.1 ❖ Очень простая функция высшего порядка

```
defmodule Hof do
  def tripler(value, function) do
    3 * function(value)
  end
end
```

Имена аргументов неконкретные, но вполне приемлемые. Функция `tripler/2` принимает значение (`value`) и функцию (`function`). Она передает указанное значение этой функции, умножает возвращаемое значение на три и возвращает результат. Испытаем эту функцию в интерактивной оболочке:

```
iex(1)> my_function = fn(value) -> 20 * value end
#Function<6.106461118/1 in :erl_eval.expr/5>
iex(2)> Hof.tripler(6, my_function)
360
```

В строке 1 определяется еще одна простая функция, принимающая единственный аргумент (и возвращающая произведение аргумента на число 20), и сохраняется в переменной `my_function`. Затем в строке 2 вызывается функция `Hof.tripler/2` со значением 6 и функцией `my_function` в аргументах. Функция `Hof.tripler/2`, в свою очередь, передает значение полученной функции и получает в ответ 120. Затем она утраивает это число и возвращает 360.

Связывание функции с переменной при желании можно опустить и просто включить объявление `fn` в вызов `Hof.tripler/2`:

```
iex(3)> Hof.tripler(6, fn(value) -> 20 * value end)
360
```

Такой код трудно читать, особенно если определяется сложная функция. Но данный пример тривиально прост, и мы можем с его помощью показать, что такое возможно.

Elixir дает еще один способ определения функций: с использованием оператора захвата `&`, когда `&1` соответствует первому аргументу, `&2` – второму и т. д. Применив данную нотацию, предыдущий пример можно упростить:

```
iex(4)> ampersand_function = &(20 * &1)
#Function<6.106461118/1 in :erl_eval.expr/5>
iex(5)> Hof.tripler(6, ampersand_function)
```



```
360
iex(6)> Hof.tripler(6, &(20 * &1))
360
```



Это очень мощный прием, но при его использовании вы легко можете пережить самих себя (судя по нашему опыту!). Так же, как в обычном коде, вы должны убедиться в соответствии ожиданиям количества параметров и их типов. Дополнительная гибкость и мощность могут создавать новые проблемы, если не проявлять осторожности.

Объявление `fn` имеет еще ряд хитростей, которые вы должны знать. Вы можете использовать `fn` для сохранения контекста на случай, если исходный контекст изменится или исчезнет:

```
iex(7)> x = 20
20
iex(8)> my_function2 = fn(value) -> x * value end
#Function<6.106461118/1 in :erl_eval.expr/5>
iex(9)> x = 0
0
iex(10)> my_function2.(6)
120
```

В строке 7 переменной с именем `x` присваивается значение, а в строке 8 эта переменная используется в определении `fn` функции. В строке 9 значение переменной `x` изменяется, но строка 10 показывает, что функция `my_function2` все еще «помнит» прежнее значение `x` — число 20. Даже притом, что значение `x` изменилось, объявление `fn` сохранило прежнее значение и может использовать его. (Это называется *замыканием*.)

И снова можно использовать форму записи с амперсандом:

```
iex(11)> x = 20
20
iex(12)> my_function3 = &(x * &1)
#Function<6.106461118/1 in :erl_eval.expr/5>
iex(13)> x = 0
0
iex(14)> Hof.tripler(6, my_function3)
360
```

Аналогично функциям высшего порядка можно передавать функции из любых модулей, даже из встроенных. В этом нет никаких сложностей:

```
iex(15)> Hof.tripler(:math.pi, &:math.cos(&1))
-3.0
```

В данном случае функция `Hof.tripler` принимает значение `pi` и функцию `:math.cos/1` из встроенного модуля `math`. Так как аргументность этой функции равна 1, это нужно обозначить с помощью `&1`. Как известно, косинус от π равен -1 , поэтому `tripler` вернула -3.0 .

Создание новых списков с помощью функций высшего порядка

Списки – одни из лучших объектов для применения функций высшего порядка. Чаще всего функции применяются ко всем элементам списка, чтобы создать, отсортировать или разбить список на меньшие части. Но для этого не требуется писать много сложного кода: встроенные модули `Elixir – List` и `Enum` – предлагают разнообразные функции высшего порядка, перечисленные в приложении А, которые принимают функцию и список и выполняют некоторые операции с ними. С той же целью можно использовать *генераторы списков* (`list comprehensions`). На первый взгляд, модули `List` и `Enum` могут показаться проще и привычнее в использовании, но, как вы увидите дальше, генераторы списков обеспечивают более выразительный и краткий синтаксис.



Функции в модуле `Enum` способны обрабатывать любые коллекции данных (например, строки в файле); функции из модуля `List` применимы только к спискам.

Получение информации о списке

Простейшей из этих функций является `Enum.each/2`, которая всегда возвращает атом `:ok`. Это может показаться странным, но `Enum.each/2` – это функция, которую следует вызывать, если и только если требуется сделать что-то с самим списком, как побочный эффект, например вывести содержимое списка в консоль. Для этого определим список и простую функцию, применяющую `IO.puts/1`, которая здесь присваивается переменной `print`, и затем передадим их функции `Enum.each/2`:

```
iex(1)> print = fn(value) -> IO.puts("#{value}") end
#Function<6.106461118/1 in :erl_eval.expr/5>
iex(2)> list = [1, 2, 4, 8, 16, 32]
iex(3)> Enum.each(list, print)
1
2
```

```
4
8
16
32
:ok
```

Функция `Enum.each/2` последовательно перебирает элементы списка и для каждого вызывает функцию из переменной `print`. Функция `IO.puts`, вызываемая внутри `print`, выводит значение элемента списка, добавляя небольшой отступ. Когда `Enum.each/2` достигает конца списка, она возвращает значение `:ok`, которое также отображается в консоли.



Большинство демонстрационных примеров в этой главе оперирует одним и тем же списком в переменной `list`, содержащим `[1, 2, 4, 8, 16, 32]`.

Обработка элементов списка с помощью функций

Также может потребоваться создать новый список из результатов обработки элементов исходного списка. Можно, к примеру, возвести в квадрат все значения в списке, создав функцию, возвращающую квадрат своего аргумента и передав ее в `Enum.map/2`. Вместо атома `:ok` эта функция возвращает новый список с результатами работы переданной ей функции:

```
iex(4)> square = &(&1 * &1)
#Function<6.106461118/1 in :erl_eval.expr/5>
iex(5)> Enum.map(list, square)
[1, 4, 16, 64, 256, 1024]
```



Сгенерировать список последовательных целых чисел (или символов) можно с помощью нотации `start..end`. Обычно она используется с целыми числами, но ее можно применить также к символам (и получить неожиданные результаты):

```
iex(6)> Enum.map(1..3, square)
[1, 4, 9]
iex(7)> Enum.map(-2..2, square)
[4, 1, 0, 1, 4]
iex(8)> Enum.map(?a..?d, square)
[9409, 9604, 9801, 10000]
```

Существует другой способ получить тот же результат, что дает `Enum.map/2`, — с применением генераторов списков:

```
iex(9)> for value <- list, do: value * value  
[1, 4, 16, 64, 256, 1024]
```

Эта инструкция возвращает такой же список, но имеет другой (более гибкий) синтаксис.

Этот генератор списка можно прочесть так: «Для каждого значения `value` в списке `list` создать элемент со значением `value * value` в новом списке». В генераторах списков можно также использовать синтаксис диапазонов:

```
for value <- 1..3, do: value * value.
```

Фильтрация значений в списках

Модуль `Enum` включает несколько функций для фильтрации содержимого списков, опираясь на функцию, переданную в параметре. Например, `Enum.filter/2` возвращает список, включающий элементы исходного списка, для которых указанная функция вернула `true`. Так, если потребуется отфильтровать список, оставив в результате только целые числа, которые можно представить четырьмя двоичными разрядами, то есть числа от 0 до 15, можно определить функцию и сохранить в переменной `four_bits`:

```
iex(10)> four_bits = fn(value) -> (value >= 0) and (value < 16) end  
#Function<6.106461118/1 in :erl_eval.expr/5>
```

Если применить ее к списку `[1, 2, 4, 8, 16, 32]`, объявленному выше, в результате останутся только четыре первых значения:

```
iex(11)> Enum.filter(list, four_bits)  
[1, 2, 4, 8]
```

И снова тот же эффект можно получить с помощью генератора списков. В данном случае не требуется определять функцию – достаточно использовать справа выражение, напоминающее ограничитель (записанное без `when`):

```
iex(12)> for value <- list, value >= 0, value < 16, do: value  
[1, 2, 4, 8]
```



Если, напротив, потребуется получить список значений, не соответствующих условию, используйте функцию `Enum.partition/2`, которая возвращает кортеж, включающий соответствующие и не соответствующие значения в двух отдельных списках. Пример можно увидеть в разделе «Разбиение списков» ниже.

За пределами возможностей генераторов списков

Генераторы `for` имеют компактный и выразительный синтаксис, но в них отсутствуют некоторые ключевые особенности, доступные при использовании рекурсивной методики обработки. Они возвращают списки, но во многих случаях требуется обработать список и вернуть что-то другое, например логическое значение, кортеж или число. В генераторах списков также отсутствует поддержка аккумуляторов, и они не имеют возможности прервать обработку раньше времени по некоторому условию.

Можно, конечно, написать свои рекурсивные функции для обработки списков, но в модулях `Enum` и `List` уже есть функции для большинства случаев, которым можно передать свою функцию и список и получить требуемый результат.

Проверка списков

Иногда достаточно просто знать, соответствуют ли все (или некоторые) элементы списка определенному критерию. Например, все они принадлежат определенному типу или для всех их выполняется некоторое условие?

Функции `Enum.all?/2` и `Enum.any?/2` позволяют проверять списки на соответствие правилам, которые вы можете оформить в виде функции. Если ваша функция вернет `true` для всех элементов списка, обе эти функции вернут `true`. `Enum.any?/2` также вернет `true`, если хотя бы для одного элемента в списке ваша функция вернет `true`. Обе вернут `false`, если ваша функция вернет `false`:

```
iex(1)> int? = fn(value) -> is_integer(value) end
#Function<erl_eval.6.17052888>
iex(2)> Enum.all?(list, int?)
true
iex(3)> Enum.any?(list, int?)
true
iex(4)> greater_than_ten? = &(&1 > 10)
#Function<6.106461118/1 in :erl_eval.expr/5>
iex(5)> Enum.all?(list, greater_than_ten?)
false
iex(6)> Enum.any?(list, greater_than_ten?)
true
```

`Enum.all?/2` можно считать аналогом функции `and`, потому что она прекращает проверку списка, как только встретит результат `false`.

Аналогично `Enum.any?`/2 можно считать аналогом `or` – как только она встретит результат `true`. Если вам требуется только проверить элементы в списке, эти две функции высшего порядка помогут вам избежать создания рекурсивного кода.



По принятым соглашениям функциям, возвращающим логическое значение (их еще называют «предикатами»), дают имена, оканчивающиеся знаком вопроса. Мы следовали этому соглашению в предыдущем примере. Функции, имена которых начинаются с `is_` (такие как `is_integer`), обычно используются в роли ограничителей.

Разбиение списков

Фильтрация списков – удобная возможность, но иногда требуется знать, какие элементы не прошли фильтра, а иногда требуется просто разделить элементы на два списка.

Функция `Enum.partition`/2 возвращает кортеж с двумя списками. Первый список содержит элементы из исходного списка, соответствующие условию, выраженному в виде вашей функции, а второй – элементы, не соответствующие этому условию. Определив переменную `compare`, как показано в предыдущей демонстрации, в строке 4, с функцией, возвращающей `true`, когда элемент списка имеет значение больше 10, вы легко сможете разбить исходный список на два списка – содержащий числа больше 10 и содержащий числа меньше или равные 10:

```
iex(7)> Enum.partition(list, compare)
{[16, 32], [1, 2, 4, 8]}
```

Иногда бывает необходимо разбить список по первому значению, не соответствующему условию. Функции `Enum.take_while`/2 и `Enum.drop_while`/2 создают новый список, содержащий часть исходного списка до или после граничного значения. Эти функции не являются фильтрами, и, чтобы прояснить это, в следующем примере используется другой список, не тот, что в остальной части главы:

```
iex(8)> test = &(&1 < 4)
#Function<6.106461118/1 in :erl_eval.expr/5>
iex(9)> Enum.drop_while([1, 2, 4, 8, 4, 2, 1], test)
[4, 8, 4, 2, 1]
iex(10)> Enum.take_while([1, 2, 4, 8, 4, 2, 1], test)
[1, 2]
```

Обе функции перебирают элементы списка в направлении от головы к хвосту и останавливаются, достигнув значения, для которого

функция, указанная вами в первом аргументе, вернет `false`. После этого функция `Enum.drop_while/2` возвращает то, что осталось в исходном списке, включая элемент, не прошедший проверку. Но она не отфильтровывает элементов, следующих в списке дальше, которые также могли бы быть отброшены, если бы встретились раньше в списке. Функция `Enum.take_while/2` возвращает уже пройденную часть списка, исключая элемент, не прошедший проверку.

Свертка списков

Добавление аккумулятора в процесс обработки списка позволяет не только превратить один список в другой, но и открывает двери к более сложным приемам обработки. Функции `List.foldl/3` и `List.foldr/3` дают возможность передать функцию, начальное значение аккумулятора и список. На этот раз вам придется определить функцию не с одним аргументом, как в примерах выше, а с двумя. В первом аргументе функции будет передано значение текущего элемента списка, а во втором – аккумулятор. Результат функции станет новым значением аккумулятора.

Определение функции для использования в операциях свертки выглядит немного иначе из-за наличия двух аргументов:

```
iex(11)> sumsq = fn(value, accumulator) -> accumulator + value * value end
#Function<12.54118792/2 in :erl_eval.expr/5>
```

Эта функция вычисляет квадраты значений в списке и суммирует их в аккумуляторе `accumulator`. Ее можно передать в функцию `List.foldl/3` с нулевым начальным значением аккумулятора, чтобы получить сумму квадратов чисел в списке:

```
iex(12)> List.foldl([2, 4, 6], 0, sumsq)
56
```

Процедура свертки имеет одну важную особенность. Вы можете выбрать направление обхода списка: от головы к хвосту (`List.foldl/3`) или от хвоста к голове (`List.foldr/3`). Если порядок обхода не влияет на результат, используйте `List.foldl/3`, так как она реализует хвостовую рекурсию и в большинстве ситуаций действует более эффективно. В предыдущем примере, если оставить вопрос эффективности, абсолютно безразлично, какую функцию выбрать, `List.foldl/3` или `List.foldr/3`, потому что от перемены мест слагаемых сумма не меняется.

Однако представьте, что в операции свертки используется следующая функция:

```
iex(13)> divide = fn(value, accumulator) -> value / accumulator end
#Function<12.106461118/2 in :erl_eval.expr/5>
```

Эта функция делит свой первый аргумент, значение из списка, на второй, аккумулятор, передаваемый функцией свертки.

Функция `divide` представляет один из тех случаев, когда направление свертки (и начальное значение аккумулятора) влияет на результат. Кроме того, в данном случае свертка производит немного иной результат, чем можно было бы ожидать при простом делении. Например, для списка `[1, 2, 4, 8, 16, 32]` можно было бы подумать, что свертка слева направо даст в результате $1/2/4/8/16/32$, а свертка справа налево – $32/16/8/4/2/1$, по крайней мере если использовать 1 как начальное значение аккумулятора. Однако в действительности результат получается иным:

```
iex(14)> divide = fn(value, accumulator) -> value / accumulator end
#Function<12.106461118/2 in :erl_eval.expr/5>
iex(15)> 1/2/4/8/16/32
3.0517578125e-5
iex(16)> List.foldl(list, 1, divide)
8.0
iex(17)> 32/16/8/4/2/1
0.03125
iex(18)> List.foldr(list, 1, divide)
0.125
```

Этот код выглядит слишком простым, чтобы содержать ошибку, так в чем же дело? В табл. 8.1 демонстрируются результаты вычислений по шагам для случая `List.foldl(list, 1, divide)`, а в табл. 8.2 – для случая `List.foldr(list, 1, divide)`.

Таблица 8.1. Рекурсивное деление списка в направлении слева направо с помощью `List.foldl/3`

Значение из списка	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 (16/4)	8

Таблица 8.2. Рекурсивное деление списка в направлении справа налево с помощью `List.foldr/3`

Значение из списка	Аккумулятор	Результат деления
32	1	32
16	32 (32/1)	0.5
8	0.5 (16/32)	16
4	16 (8/0.5)	0.25
2	0.25 (4/16)	8
1	8 (2/0.25)	0.125

При движении по списку шаг за шагом получается совсем другой результат. В данном случае поведение простой функции `divide` значительно отличается для значений выше и ниже 1, из-за чего результаты свертки списка с ее помощью отличаются от ожидаемых.



В данном случае свертка с помощью `List.foldl` воспроизводит выражение $32 / (16 / (8 / (4 / (2 / (1 / 1)))))$, а свертка с помощью `List.foldr` — выражение $1 / (2 / (4 / (8 / (16 / (32 / 1)))))$. Круглые скобки отмечают шаги в процессе свертки, а завершающее значение 1 в обоих случаях — это начальное значение аккумулятора.

Свертка — удивительно мощная операция. Выше был представлен простой пример с единственным значением в аккумуляторе. Но если в роли аккумулятора использовать кортеж, в нем можно будет хранить любые виды данных и даже выполнять многократные операции с ним. Вероятно, вы не будете использовать в своих свертках однострочные функции, но теперь вы знаете, что это возможно.

Глава 9

Процессы

Elixir – это функциональный язык, но программы на Elixir редко структурируются исключительно на основе простых функций. Чаще используется другая организационная единица – *процесс*, независимый компонент (построенный на функциях), которая посылает и принимает сообщения. Программы разворачиваются как набор процессов, взаимодействующих друг с другом. Этот подход существенно упрощает разработку программ, действующих сразу на нескольких процессорах или компьютерах, а также делает возможным обновление программ без остановки всей системы.

Однако для использования этих преимуществ необходимо знать, как запускать (и завершать) процессы, как посылать сообщения между ними и как использовать механизм сопоставления с образцом для обработки входящих сообщений.

Интерактивная оболочка – это процесс

До сих пор в этой книге мы работали в рамках единственного процесса – интерактивной оболочки Elixir. Ни один из прежде представленных примеров не посылал и не принимал сообщений, но интерактивная оболочка дает такую возможность (по крайней мере, для тестирования).

Первое понятие, с которым мы познакомимся, – это *идентификатор процесса* (process identifier, или pid). Проще всего узнать собственный идентификатор процесса, для чего достаточно вызвать функцию `self()`:

```
iex(1)> self()  
#PID<0.26.0>
```

В данном случае `#PID<0.26.0>` – это идентификатор процесса в представлении оболочки. Он состоит из трех целых чисел, уникаль-

но идентифицирующих процесс. Вы можете получить у себя другие три числа. Эта группа чисел гарантированно уникальна в данном сеансе выполнения Elixir, но она не постоянна, и потому бессмысленно запоминать ее для использования в будущем. Идентификаторы процессов используются исключительно для внутренних нужд, и даже притом, что вы можете получать их в оболочке, вы не сможете использовать их непосредственно. Идентификаторы процессов служат лишь абстракцией.



Идентификаторы процессов позволяют идентифицировать даже процессы, выполняющиеся на других компьютерах внутри кластера. Вам потребуется приложить определенные усилия, чтобы настроить кластер, но вам не придется выбрасывать код, использующий идентификаторы процессов и процессы.

Каждый процесс получает собственный идентификатор, и эти идентификаторы действуют подобно почтовым адресам. Ваши программы будут использовать их для отправки сообщений между процессами. Когда в процессе наступит момент проверить свой «почтовый ящик», он сможет извлечь оттуда и обработать все поступившие сообщения.

Однако Elixir *никогда не сообщает*, если попытка отправить сообщение окончилась неудачей, например если процесса с указанным идентификатором `pid` не существует. Он также ничего не сообщает, если какой-то процесс решил проигнорировать сообщение. По этой причине вы должны гарантировать корректную сборку ваших процессов.

Синтаксис отправки сообщений очень прост. Для этого требуется вызвать функцию `send/2` с двумя аргументами: выражение, представляющее `pid`, и сообщение:

```
iex(2)> send(self(), :test1)
:test1
iex(3)> pid = self()
#PID<0.26.0>
iex(4)> send(pid, :test2)
:test2
```

Строка 2 посылает интерактивной оболочке сообщение, содержащее атом `:test1`. Строка 3 сохраняет идентификатор процесса интерактивной оболочки в переменной `pid`, полученный с помощью функции `self()`, и затем строка 4 использует эту переменную для отправки сообщения с атомом `:test2`. (Функция `send/2` всегда возвращает со-

общение, именно поэтому оно было выведено сразу после выполнения строк 2 и 4.)

А куда попали эти сообщения? Что с ними произошло? Прямо сейчас они хранятся в почтовом ящике процесса интерактивной оболочки и с ними ничего не происходит.

В оболочке существует еще одна функция – `flush()`, которую можно использовать для просмотра содержимого почтового ящика, но она при этом удаляет эти сообщения. Первый вызов функции вернет содержимое почтового ящика, но второй ее вызов не вернет ничего, потому что сообщения уже были прочитаны и удалены:

```
iex(5)> flush()
:test1
:test2
:ok
iex(6)> flush()
:ok
```

Более правильный способ чтения почтового ящика, дающий возможность обработать сообщения, – использовать конструкцию `receive...end`. Ее можно проверить в интерактивной оболочке. Следующий пример сначала просто выводит полученное сообщение, а затем принимает число и удваивает его:

```
iex(7)> send(self(), :test1)
:test1
iex(8)> receive do
...(8)>   x -> x
...(8)> end
:test1
iex(9)> send(self(), 23)
23
iex(10)> receive do
...(10)>   y -> 2 * y
...(10)> end
46
```

Пока все просто. Однако если в почтовом ящике не окажется сообщений или в сопоставлении с образцом допустить ошибку, оболочка просто зависнет. В действительности он будет ждать получения требуемого сообщения (или, говоря техническим языком, получения блоков, составляющих сообщения), но вы все равно не сможете продолжить работу. Если вы попадете в такую ситуацию, нажмите **Ctrl+G** и затем введите `q`. В результате оболочка IEx будет перезапущена. (Переменные `x` и `y` в примере выше становятся связанными

переменными, и даже притом, что они не являются неизменяемыми, они, согласно духу функционального программирования, недоступны для повторного использования.)

Порождение процессов из модулей

Несмотря на то что отправка сообщений в интерактивной оболочке выполняется очень просто и дает возможность увидеть происходящее, в этом мало пользы. Процессы по своей сути являются обычными функциями, а вы уже знаете, как конструировать функции в модулях. Инструкция `receive...end` по своей структуре напоминает инструкцию `case...end`.

Пример 9.1 (*ch09/ex1-simple*) демонстрирует простой – очень простой – модуль, содержащий функцию, которая сообщает о полученных сообщениях.

Пример 9.1 ❖ Простейшее определение процесса

```
defmodule Bounce do
  def report do
    receive do
      msg -> IO.puts("Received #{msg}")
    end
  end
end
```

Когда функция `report/0` получает сообщение, она просто выводит его. Этот модуль можно скомпилировать и затем с помощью функции `spawn/3` превратить функцию в самостоятельный процесс. В аргументах функции `spawn/3` нужно передать имя модуля, имя функции (как атом) и список аргументов для функции в модуле. Если функция не имеет аргументов, вы должны включить в вызов пустой список в виде пары квадратных скобок. Функция `spawn/3` вернет `pid`, который следует сохранить в переменной (в следующем примере она названа `pid`):

```
iex(1)> pid = spawn(Bounce, :report, [])
#PID<0.43.0>
```

После порождения (запуска) процесса ему можно посылать сообщения, используя сохраненный `pid`, а процесс будет выводить информацию о принятых сообщениях:

```
iex(2)> send(pid, 23)
Received 23
23
```

Однако здесь есть одна маленькая проблема. После приема первого сообщения процесс `report` завершается – он выполняет инструкцию `receive` только один раз. Если попытаться послать процессу другое сообщение, оболочка просто выведет это сообщение, не генерируя никаких ошибок, но при этом вы не увидите уведомления о том, что процесс принял сообщение, потому что его уже просто нет:

```
iex(3)> send(pid, 23)
23
```

Чтобы заставить процесс продолжать принимать и обрабатывать сообщения, нужно добавить рекурсивный вызов, как показано в примере 9.2 (*ch09/ex2-recursion*).

Пример 9.2 ❖ Функция, создающая процесс, выполняющийся в цикле

```
defmodule Bounce do
  def report do
    receive do
      msg -> IO.puts("Received #{msg}")
      report()
    end
  end
end
```

Дополнительный вызов `report()` означает, что после вывода сообщения функция будет вызвана снова и готова принять следующее сообщение. Если теперь перекомпилировать модуль `Bounce` и вызвать `spawn` (сохранив результат в новой переменной `pid2`), вы сможете послать множество сообщений своему процессу, как показано ниже:

```
iex(4)> r(Bounce)
warning: redefining module Bounce (current version loaded from
  _build/dev/lib/bounce/ebin/Elixir.Bounce.beam)
  /Users/elixir/code/ch09/ex2-recursion/lib/bounce.ex:1

{:reloaded, Bounce, [Bounce]}
iex(5)> pid2 = spawn(Bounce, :report, [])
#PID<0.43.0>
iex(6)> send(pid2, 23)
Received 23
23
iex(7)> send(pid2, :message)
Received message
:message
```



Процессы выполняются асинхронно, поэтому вывод от `send/2` может появиться раньше, чем вывод от `report/0`.

При желании между вызовами можно также передавать аккумулятор, например для подсчета количества сообщений, принятых процессом. В примере 9.3 (*ch09/ex3-counter*) показано, как использовать такой дополнительный аргумент, который в данном случае является простым целым числом, наращиваемым с каждым новым вызовом.

Пример 9.3 ❖ Функция со счетчиком сообщений

```
defmodule Bounce do
  def report(count) do
    receive do
      msg -> IO.puts("Received #{count}: #{msg}")
      report(count + 1)
    end
  end
end
```

Результат предскажем, но не забывайте о необходимости включить начальное значение в список аргументов в вызове `spawn/3`:

```
$ iex -S mix
Erlang/OTP 19 [erts-8.0] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe]
[kernel-poll:false]
```

```
Compiling 1 file (.ex)
Interactive Elixir (1.3.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> pid2 = spawn(Bounce, :report, [1])
#PID<0.43.0>
iex(2)> send(pid2, :test)
:test
Received 1: test
iex(3)> send(pid2, :test2)
:test2
Received 2: test2
iex(4)> send(pid2, :another)
:another
Received 3: another
```

Независимо от операций, выполняемых в рекурсивной функции, старайтесь обеспечить возможность оптимизации до хвостовой рекурсии, потому что такого рода функции могут вызываться в течение жизни процесса много-много раз.



Чтобы создать процесс, останавливающийся спустя некоторое время ожидания сообщения, используйте конструкцию `after` в инструкции `receive`.

Функцию `report` можно записать немного иначе, чтобы происходящее в ней просматривалось более четко. Пример 9.4 (*ch09/ex4-state*) показывает, как можно использовать значение, возвращаемое инструкцией `receive` (здесь – `count` плюс единица), для передачи состояния между итерациями.

Пример 9.4 ❖ Использование значения, возвращаемое инструкцией `receive`, для передачи состояния в следующую итерацию

```
defmodule Bounce do
  def report(count) do
    new_count = receive do
      msg -> IO.puts("Received #{count}: #{msg}")
      count + 1
    end
    report(new_count)
  end
end
```

В этой модели все (хотя в данном случае только одна) инструкции `receive` возвращают значение, которое передается в следующую итерацию. В таком случае возвращаемое значение инструкции `receive` можно рассматривать как состояние, сохраняемое между вызовами функции. Это состояние может иметь намного более сложную организацию, чем простой счетчик, – это может быть кортеж, например, включающий ссылки на ресурсы, необходимые в работе.

Легковесные процессы

Имеющие опыт использования других языков программирования могут испытывать беспокойство по поводу процессов. Процедура запуска потоков выполнения и процессов в других контекстах весьма сложна и потребляет существенные вычислительные ресурсы, а Elixir предполагает, что приложение состоит из группы процессов. Есть ли повод для беспокойства?

Абсолютно нет! В Elixir реализована особенно эффективная модель, и процессы в этом языке намного более легковесные, чем их аналоги у конкурентов. Планировщик Erlang, который используется в Elixir, запускает процессы и распределяет время работы между ними, а также обеспечивает их выполнение на разных процессорах.

Конечно, можно написать процессы, безрассудно потребляющие ресурсы, и организовать приложение так, что оно будет долго ждать,

прежде чем сделать что-то полезное. Однако у вас нет причин волноваться только лишь потому, что в приложении используется несколько процессов.

Регистрация процесса

В большинстве случаев для организации связи с процессами необходимы только их идентификаторы `pid`. Однако нередко возникает необходимость в более универсальных инструментах поиска процессов. В Elixir для этих целей имеется чрезвычайно простая система регистрации процессов: вы передаете ей атом и `pid` процесса, а затем любой процесс, желающий установить связь с зарегистрированным процессом, просто использует атом для его поиска. Это, например, упрощает добавление новых процессов в систему и организацию их взаимодействий с ранее запущенными процессами.

Чтобы зарегистрировать процесс, достаточно вызвать встроенную функцию `Process.register/2`. В первом аргументе ей передается `pid` процесса, а во втором – атом, используемый как имя этого процесса. После регистрации процесса ему можно посылать сообщения, используя атом вместо `pid`:

```
iex(1)> pid1 = spawn(Bounce, :report, [1])
#PID<0.39.0>
iex(2)> Process.register(pid1, :bounce)
true
iex(3)> send(:bounce, :hello)
:hello
Received 1: hello
iex(4)> send(:bounce, "Really?")
Received 2: Really?
"Really?"
```

Если попытаться послать сообщение завершившемуся процессу (возможно, в результате ошибки), вы получите ошибку:

```
iex(5)> send(:zingo, :test)
** (ArgumentError) argument error
:erlang.send(:zingo, :test)
```

Если попытаться зарегистрировать процесс с уже зарегистрированным именем, вы также получите ошибку, но если процесс с таким именем уже завершился (возможно, в результате ошибки), его имя будет считаться незарегистрированным и доступным для повторной регистрации.

С помощью функции `Process.whereis/1` можно также получить `pid` зарегистрированного процесса (или `nil`, если процесс с указанным атомом не зарегистрирован), а с помощью `unregister/1` можно удалить процесс из списка регистрации, не останавливая его. Помните, что в роли имени процесса необходимо использовать атом:

```
iex(5)> get_bounce = Process.whereis(:bounce)
#PID<0.39.0>
iex(6)> Process.unregister(:bounce)
true
iex(7)> test_bounce = Process.whereis(:bounce)
nil
iex(8)> send(get_bounce, "Still there?")
Received 3: Still there?
"Still there?"
```



Чтобы узнать, какие процессы зарегистрированы, используйте функцию `Process.registered/0`.

Имеющие опыт работы с другими языками программирования и наизуок выучившие мантру «никаких глобальных переменных» могут задать вопрос: почему Elixir допускает поддержку общесистемного реестра процессов, подобного описанному выше?

Однако если рассматривать зарегистрированные процессы как службы, а не функции, все становится на свои места. Зарегистрированный процесс фактически является службой, доступной в системе, и может использоваться из нескольких контекстов. При разумном подходе зарегистрированные процессы образуют надежные точки входа в ваши программы, что очень ценное преимущество, которое начинает проявляться с ростом объема и сложности кода.

Когда процесс останавливается

Процесс – хрупкая вещь. Если возникает ошибка, функция останавливается, и процесс завершается. В примере 9.5 (*ch09/ex5-division*) демонстрируется функция `report/0`, которая терпит неудачу, если получит аргумент, не являющийся числом.

Пример 9.5 ❖ Хрупкая функция

```
defmodule Bounce do
  def report do
    receive do
      x -> IO.puts("Divided to #{x / 2}")
      report()
    end
  end
end
```

```

end
end
end

```

Если скомпилировать и запустить этот код, который легко обрушить, можно заметить, что он прекрасно работает, пока ему посылаются числа. Но стоит послать ему что-то другое, как на экране появится сообщение об ошибке, и после этого процесс перестанет откликаться. Он просто завершится:

```

iex(1)> pid3 = spawn(Bounce, :report, [])
#PID<0.50.0>
iex(2)> send(pid3, 38)
38
Divided to 19.0
iex(3)> send(pid3, 27.56)
Divided to 13.78
27.56
iex(4)> send(pid3, :seven)
:seven
iex(5)>
14:18:59.471 [error] Process #PID<0.65.0> raised an exception
** (ArithmeticError) bad argument in arithmetic expression
    bounce.ex:4: Bounce.report/0
iex(5)> send(pid3, 14)
14

```

Углубившись в изучение модели процессов, легко заметить, что философия «позволь потерпеть аварию» не такая уж необычная в Elixir, потому что защита от аварий и обеспечение непрерывной работы требуют дополнительных усилий со стороны программиста. В главе 10 вы узнаете, как обнаруживать и обрабатывать ошибки разных видов.

Взаимодействие между процессами

Послать сообщение процессу просто, но сложнее реализовать возврат ответов, если не сообщить, куда следует послать ответ. Отправка сообщения без идентификатора отправителя сродни звонку с телефона, защищенному антиопределителем номера: вы можете позвонить абоненту и сообщить ему какую-то информацию, но тот не сможет перезвонить вам, чтобы сообщить о результатах.

Чтобы установить связь между двумя процессами без регистрации чрезмерно большого количества процессов, необходимо включать

идентификаторы `pid` в сообщения. Для передачи `pid` требуется добавить аргумент в сообщение. Рассмотрим простой пример процесса, который возвращает ответ оболочке. Пример 9.6 (*ch09/ex6-talking*) основан на модуле `Drop` из примера 3.2. Он добавляет функцию `drop/0`, которая принимает сообщения, и объявляет функцию `fall_velocity/2` приватной.

Пример 9.6 ❖ Процесс, посылающий сообщение вызвавшему процессу

```
defmodule Drop do
  def drop do
    receive do
      {from, planemo, distance} ->
        send(from, {planemo, distance, fall_velocity(planemo, distance)})
        drop()
    end
  end

  defp fall_velocity(:earth, distance) when distance >= 0 do
    :math.sqrt(2 * 9.8 * distance)
  end

  defp fall_velocity(:moon, distance) when distance >= 0 do
    :math.sqrt(2 * 1.6 * distance)
  end

  defp fall_velocity(:mars, distance) when distance >= 0 do
    :math.sqrt(2 * 3.71 * distance)
  end
end
```

Для начала протестируем этот код из оболочки:

```
iex(1)> pid1 = spawn(Drop, :drop, [])
#PID<0.43.0>
iex(2)> send(pid1, {self(), :moon, 20})
{#PID<0.26.0>, :moon, 20}
iex(3)> flush()
{:moon, 20, 8.0}
:ok
```

Пример 9.7 (*ch09/ex7-talkingProcs*) реализует процесс, который вызывает предыдущий процесс, чтобы показать, что такая схема взаимодействий возможна не только в интерактивной оболочке. Мы использовали здесь функцию `IO.write/1`, чтобы уместить листинг по ширине книжной страницы, но данные при этом будут выводиться в одну строку.

Пример 9.7 ❖ Вызов другого процесса и вывод результатов

```
defmodule MphDrop do
  def mph_drop do
    drop_pid = spawn(Drop, :drop, [])
    convert(drop_pid)
  end

  def convert(drop_pid) do
    receive do
      {planemo, distance} ->
        send(drop_pid, {self(), planemo, distance})
        convert(drop_pid)
      {planemo, distance, velocity} ->
        mph_velocity = 2.23693629 * velocity
        IO.write("On #{planemo}, a fall of #{distance} meters ")
        IO.puts("yields a velocity of #{mph_velocity} mph.")
        convert(drop_pid)
    end
  end
end
```

Функция `mph_drop/1` запускает процесс `Drop.drop/0`, используя модуль из примера 9.6, и сохраняет его идентификатор в переменной `drop_pid`. Затем вызывается функция `convert/1`, которая рекурсивно выполняет обработку сообщений.



Если не отделить инициализацию от обработчика, этот процесс будет работать, но он будет запускать новые процессы `Drop.drop/0` при получении сообщения вместо использования уже запущенного.

Инструкция `receive` полагается на вызов из интерактивной оболочки (или другого процесса), включающий только два аргумента, тогда как процесс `Drop.drop/0` посылает обратно результат с тремя аргументами. (По мере увеличения сложности кода вам почти наверняка потребуется использовать какие-то более явные флаги с информацией о содержимом сообщения.) Когда инструкция `receive` получает сообщение с двумя аргументами, она посылает сообщение процессу `drop_pid`, идентифицировав себя как отправителя. Когда процесс `drop_pid` вернет сообщение с результатом, инструкция `receive` преобразует скорость в мили в час и выведет полученные результаты. (Да, высота остается выраженной в метрах, зато скорость становится более понятной американцам.)

Ниже показаны результаты испытания этой пары процессов проше, в интерактивной оболочке после предварительной команды `iex -S mix`:

```
iex(1)> pid1 = spawn(MphDrop, :mph_drop, [])
#PID<0.47.0>
iex(2)> send(pid1, {:earth, 20})
On earth, a fall of 20 meters
yields a velocity of 44.289078952755766 mph.
{:earth,20}
iex(3)> send(pid1, {:mars, 20})
On mars, a fall of 20 meters
yields a velocity of 27.250254686571544 mph.
{:mars,20}
```

Этот простой запрос может показаться усложненной версией вызова функции, но здесь есть одно важное отличие. В интерактивной оболочке, когда система не загружена решением тяжелых задач, результат возвращается максимально быстро – настолько быстро, что появляется раньше, чем оболочка успевает вывести отправленное сообщение, – но в работе участвуют асинхронные процессы.

Оболочка послала сообщение процессу с идентификатором в `pid1`, то есть процессу `MphDrop.convert/1`. Этот процесс послал сообщение процессу с идентификатором в `drop_pid`, то есть процессу `Drop.drop/0`, который вызвал `MphDrop.mph_drop/0` для настройки. Этот процесс вернул другое сообщение процессу `MphDrop.convert/1`, который отправил его на экран. В данном случае сообщения передаются и обрабатываются очень быстро, но в системе с тысячами и миллионами курсирующих сообщений эти запросы и ответы могут разделяться многими и многими другими сообщениями.

Наблюдение за процессами

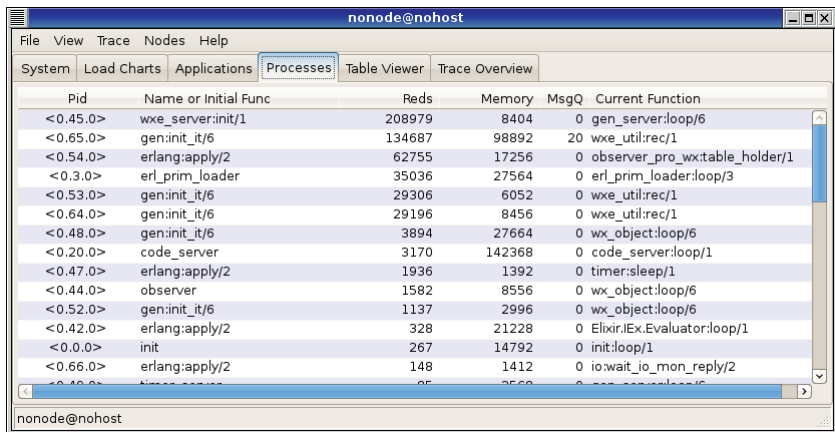
В составе Erlang имеется простой, но мощный инструмент для наблюдения за процессами и происходящим в них. *Observer*, инструмент для наблюдения за процессами и управления ими, имеет графический интерфейс, позволяющий видеть текущее состояние процессов и что в них происходит. В зависимости от того, как устанавливался Erlang во время установки Elixir, может иметься возможность запускать этот инструмент из панели, но его всегда можно запустить из интерактивной оболочки:

```
iex(4)> :observer.start
#PID<0.49.0>
```



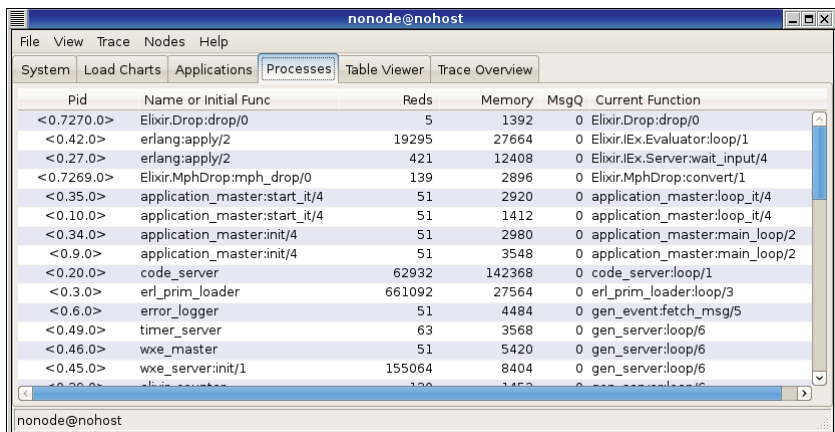
Чтобы пользоваться инструментом Observer, в системе должна быть установлена библиотека `wxwidgets`, а Erlang должен быть скомпилирован с ее поддержкой.

Щелкнув на вкладке **Processes** (Процессы), вы увидите картину, напоминающую рис. 9.1. Это длинный список процессов, намного длиннее, чем вы могли бы ожидать. Если щелкнуть на заголовке столбца **Current Function** (Текущая функция), чтобы отсортировать список в порядке убывания, вы увидите процессы Elixir, как показано на рис. 9.2.



Pid	Name or Initial Func	Reds	Memory	MsgQ	Current Function
<0.45.0>	wxe_server:init/1	208979	8404	0	gen_server:loop/6
<0.65.0>	gen:init_it/6	134687	98892	20	wxe_util:rec/1
<0.54.0>	erlang:apply/2	62755	17256	0	observer_pro_wx:table_holder/1
<0.3.0>	erl_prim_loader	35036	27564	0	erl_prim_loader:loop/3
<0.53.0>	gen:init_it/6	29306	6052	0	wxe_util:rec/1
<0.64.0>	gen:init_it/6	29196	8456	0	wxe_util:rec/1
<0.48.0>	gen:init_it/6	3894	27664	0	wx_object:loop/6
<0.20.0>	code_server	3170	142368	0	code_server:loop/1
<0.47.0>	erlang:apply/2	1936	1392	0	timer:sleep/1
<0.44.0>	observer	1582	8556	0	wx_object:loop/6
<0.52.0>	gen:init_it/6	1137	2996	0	wx_object:loop/6
<0.42.0>	erlang:apply/2	328	21228	0	Elixir.IEx.Evaluator:loop/1
<0.0.0>	init	267	14792	0	init:loop/1
<0.66.0>	erlang:apply/2	148	1412	0	io_wait_io_mon_reply/2

Рис. 9.1 ❖ Список процессов в окне **Observer** сразу после запуска инструмента



Pid	Name or Initial Func	Reds	Memory	MsgQ	Current Function
<0.7270.0>	Elixir.Drop:drop/0	5	1392	0	Elixir.Drop:drop/0
<0.42.0>	erlang:apply/2	19295	27664	0	Elixir.IEx.Evaluator:loop/1
<0.27.0>	erlang:apply/2	421	12408	0	Elixir.IEx.Server:wait_input/4
<0.7269.0>	Elixir.MphDrop:mpm_drop/0	139	2896	0	Elixir.MphDrop:convert/1
<0.35.0>	application_master:start_it/4	51	2920	0	application_master:loop_it/4
<0.10.0>	application_master:start_it/4	51	1412	0	application_master:loop_it/4
<0.34.0>	application_master:init/4	51	2980	0	application_master:main_loop/2
<0.9.0>	application_master:init/4	51	3548	0	application_master:main_loop/2
<0.20.0>	code_server	62932	142368	0	code_server:loop/1
<0.3.0>	erl_prim_loader	661092	27564	0	erl_prim_loader:loop/3
<0.6.0>	error_logger	51	4484	0	gen_event:fetch_msg/5
<0.49.0>	timer_server	63	3568	0	gen_server:loop/6
<0.46.0>	wxe_master	51	5420	0	gen_server:loop/6
<0.45.0>	wxe_server:init/1	155064	8404	0	gen_server:loop/6

Рис. 9.2 ❖ Список процессов в окне **Observer** после сортировки по столбцу **Current Function** (Текущая функция)

Observer обновляет список процессов каждые 10 секунд. Если вы предпочтете обновлять содержимое списка вручную, выберите пункт **Refresh Interval** (Интервал обновления) в меню **View** (Вид) и снимите флажок **Periodical Refresh** (Обновлять периодически).

Наблюдение за движением сообщений между процессами

Список процессов – сам по себе штука хорошая, но Observer также позволяет заглянуть внутрь процессов. Это более сложная процедура, поэтому вдохните глубже!

1. Найдите процесс `Elixir.MphDrop:mph_drop/0` и щелкните на нем правой кнопкой мыши.
2. Выберите **Trace selected processes by name (all nodes)** (Трассировать процессы по имени (все узлы)) и затем установите все флажки в левой половине диалога, как показано на рис. 9.3. Затем щелкните на кнопке **OK**.

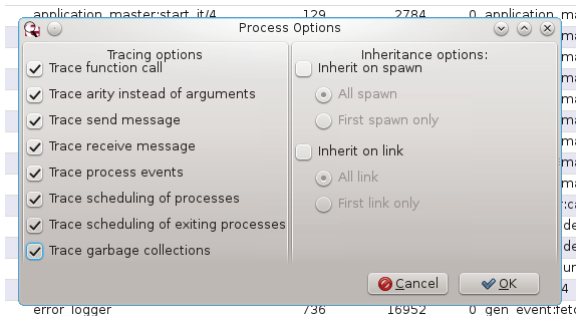


Рис. 9.3 ❖ Параметры трассировки процессов

3. Щелкните на вкладке **Trace Overview** (Обзор трассировки).
4. Щелкните на кнопке **Start Trace** (Начать трассировку), и вы увидите предупреждение, как показано на рис. 9.4. Его можно смело игнорировать.

После этого откроется новое окно с сообщением вида: «Dropped 10 messages» (Сброшено 10 сообщений). Теперь попробуйте заставить процесс выполнить какие-либо действия:

```
iex(5)> send(pid1, {:mars, 20})
On mars, a fall of 20 meters
yields a velocity of 27.25025468657154448238 mph.
{:mars,20}
```

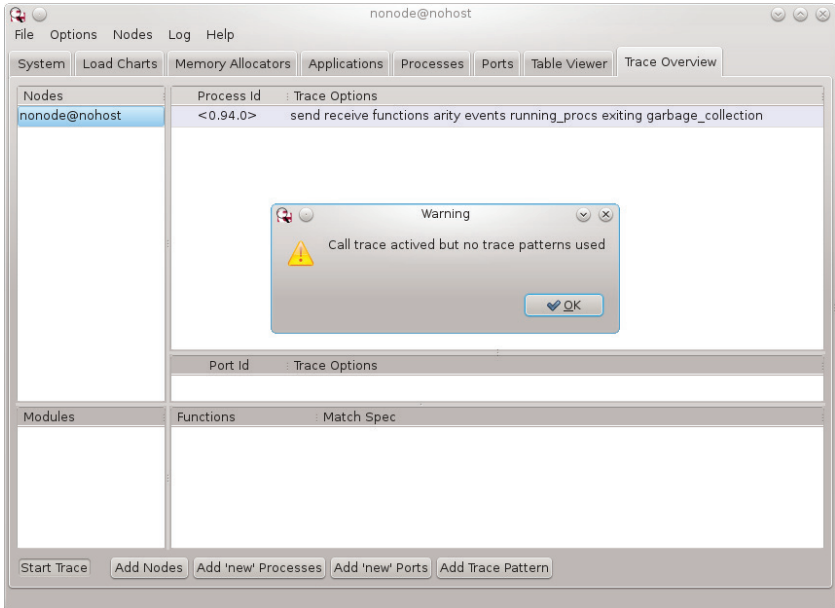



Рис. 9.4 ❖ Запуск трассировки

Окно Observer для этого процесса обновится, и в нем появится информация о сообщениях и вызванных функциях, как показано на рис. 9.5. Две открывающие угловые скобки << означают, что сообщение было получено, а восклицательный знак ! отмечает отправку сообщения.

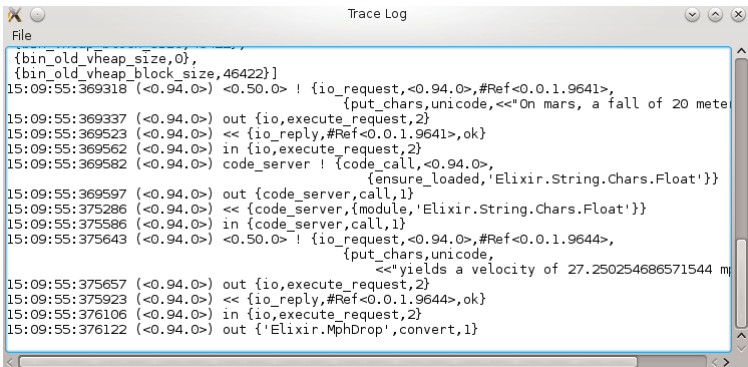


Рис. 9.5 ❖ Трассировка вызовов после отправки сообщения mph_drop

Observer должен быть для вас первым инструментом, к которому следует обратиться, когда требуется выяснить, что происходит в процессах.

Разрыв и установка связей между процессами

Посылая сообщение, вы всегда будете получать его в виде возвращаемого значения. Однако это не означает, что все идет как надо и сообщение было получено и обработано. Если посланное сообщение не соответствует образцу в принимающем процессе, ничего не произойдет (по крайней мере, пока), сообщение просто будет помещено в почтовый ящик, но не вызовет никаких действий. Отправка сообщения, соответствующего образцу, но вызывающего ошибку, приведет к остановке процесса, где возникла ошибка, возможно, даже к остановке нескольких связанных процессов.



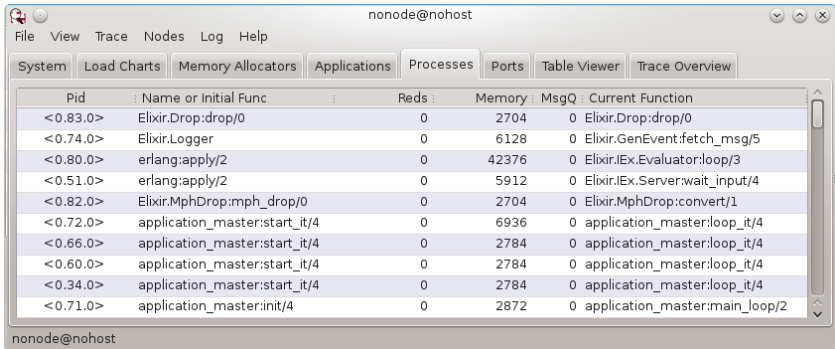
Сообщения, не соответствующие образцу, в инструкции `receive` не удаляются автоматически; они просто лежат мертвым грузом в почтовом ящике. Теоретически можно обновить процесс, добавив в него код для таких сообщений, и они будут обработаны.

Так как процессы могут завершаться аварийно, часто бывает желательно иметь возможность получать уведомления об этом в других процессах. Например, если некорректные данные повлекут сбой в процессе `Drop.drop/0`, нет никакого смысла продолжать выполнять процесс `MphDrop.convert/1`. Давайте понаблюдаем, что происходит в этом случае, в оболочке и в окне Observer. Сначала запустите Observer, откройте список процессов и затем из оболочки запустите процесс `MphDrop.mph_drop/0`:

```
iex(1)> :observer.start()
:ok
iex(2)> pid1 = spawn(MphDrop, :mph_drop, [])
#PID<0.82.0>
```

Вы увидите картину, напоминающую рис. 9.6. Затем передайте процессу некорректные данные, например атом (`:zoids`) вместо числа в аргументе `distance`. После этого окно Observer будет выглядеть, как показано на рис. 9.7:

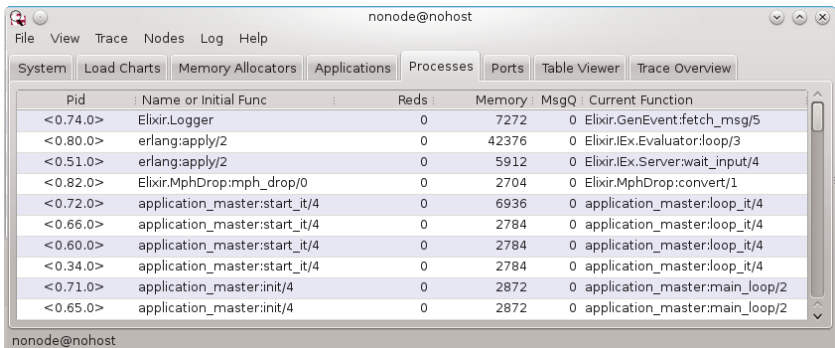
```
iex(3)> send(pid1, {:moon, :zoids})
19:28:27.825 [error] Process #PID<0.83.0> raised an exception
** (ArithmeticError) bad argument in arithmetic expression
    (mph_drop) lib/drop.ex:15: Drop.fall_velocity/2
    (mph_drop) lib/drop.ex:5: Drop.drop/0
```



The screenshot shows the 'Processes' tab in the Erlang shell. The table lists the following processes:

Pid	Name or Initial Func	Reds	Memory	MsgQ	Current Function
<0.83.0>	Elixir.Drop:drop/0	0	2704	0	Elixir.Drop:drop/0
<0.74.0>	Elixir.Logger	0	6128	0	Elixir.GenEvent:fetch_msg/5
<0.80.0>	erlang:apply/2	0	42376	0	Elixir.IEx.Evaluator:loop/3
<0.51.0>	erlang:apply/2	0	5912	0	Elixir.IEx.Server:wait_input/4
<0.82.0>	Elixir.MphDrop:mph_drop/0	0	2704	0	Elixir.MphDrop:convert/1
<0.72.0>	application_master:start_it/4	0	6936	0	application_master:loop_it/4
<0.66.0>	application_master:start_it/4	0	2784	0	application_master:loop_it/4
<0.60.0>	application_master:start_it/4	0	2784	0	application_master:loop_it/4
<0.34.0>	application_master:start_it/4	0	2784	0	application_master:loop_it/4
<0.71.0>	application_master:init/4	0	2872	0	application_master:main_loop/2

Рис. 9.6 ❖ Набор выполняющихся процессов



The screenshot shows the 'Processes' tab after the `Drop.drop/0` process has terminated. The table lists the following processes:

Pid	Name or Initial Func	Reds	Memory	MsgQ	Current Function
<0.74.0>	Elixir.Logger	0	7272	0	Elixir.GenEvent:fetch_msg/5
<0.80.0>	erlang:apply/2	0	42376	0	Elixir.IEx.Evaluator:loop/3
<0.51.0>	erlang:apply/2	0	5912	0	Elixir.IEx.Server:wait_input/4
<0.82.0>	Elixir.MphDrop:mph_drop/0	0	2704	0	Elixir.MphDrop:convert/1
<0.72.0>	application_master:start_it/4	0	6936	0	application_master:loop_it/4
<0.66.0>	application_master:start_it/4	0	2784	0	application_master:loop_it/4
<0.60.0>	application_master:start_it/4	0	2784	0	application_master:loop_it/4
<0.34.0>	application_master:start_it/4	0	2784	0	application_master:loop_it/4
<0.71.0>	application_master:init/4	0	2872	0	application_master:main_loop/2
<0.65.0>	application_master:init/4	0	2872	0	application_master:main_loop/2

Рис. 9.7 ❖ Процесс `drop:drop/0` исчез

Так как оставшийся процесс `MphDrop.convert/1` теперь бесполезен, его желательно было бы завершить сразу после завершения процесса `Drop.drop/0`. Elixir позволяет настроить такую связь между процессами. Самый простой способ добиться этого и избежать неприятностей с возможным состоянием гонки – использовать функцию `spawn_link/3` вместо `spawn/3`. Весь остальной код в модуле остается без изменений, как показано в примере 9.8 (*ch09/ex8-linking*).

Пример 9.8 ❖ Установка связи между процессами

```
defmodule MphDrop do
  def mph_drop do
    drop_pid = spawn_link(Drop, :drop, [])
    convert(drop_pid)
  end
end
```

```

def convert(drop_pid) do
  receive do
    {planemo, distance} ->
      send(drop_pid, {self(), planemo, distance})
      convert(drop_pid)
    {planemo, distance, velocity} ->
      mph_velocity = 2.23693629 * velocity
      IO.write("On #{planemo}, a fall of #{distance} meters ")
      IO.puts("yields a velocity of #{mph_velocity} mph.")
      convert(drop_pid)
  end
end
end
end

```

Если теперь скомпилировать модуль и повторить эксперимент, в окне Observer можно заметить, что остановились оба процесса, когда `drop:drop/0` потерпел аварию, как показано на рис. 9.8:

```

iex(1)> :observer.start()
:ok
iex(2)> pid1 = spawn(MphDrop, :mph_drop, [])
#PID<0.162.0>
iex(3)> send(pid1, {:moon, :zoids})
{:moon, :zoids}
iex(4)>
19:30:26.822 [error] Process #PID<0.163.0> raised an exception
** (ArithmeticError) bad argument in arithmetic expression
    (mph_drop) lib/drop.ex:15: Drop.fall_velocity/2
    (mph_drop) lib/drop.ex:5: Drop.drop/0

```

Pid	Name or Initial Func	Reds	Memory	MsgQ	Current Function
<0.137.0>	Elixir.Logger	0	7272	0	Elixir.GenEvent.fetch_msg/5
<0.143.0>	erlang:apply/2	0	34472	0	Elixir.IEx.Evaluator.loop/3
<0.51.0>	erlang:apply/2	0	5912	0	Elixir.IEx.Server.wait_input/4
<0.135.0>	application_master:start_it/4	0	5792	0	application_master.loop_it/4
<0.84.0>	application_master:start_it/4	0	2784	0	application_master.loop_it/4
<0.66.0>	application_master:start_it/4	0	2784	0	application_master.loop_it/4
<0.60.0>	application_master:start_it/4	0	2784	0	application_master.loop_it/4
<0.34.0>	application_master:start_it/4	0	2784	0	application_master.loop_it/4

Рис. 9.8 ❖ Теперь ошибка вызвала остановку обоих процессов



Связи являются двунаправленными. Если остановить процесс `MphDrop.mph_drop/0`, например, вызовом `Process.exit(pid1, :kill)`, процесс `Drop.drop/0` так же остановится. (`:kill` – самая неукоснительная

причина остановиться для процесса, ее нельзя перехватить, потому что иногда действительно бывает нужно принудительно завершить процесс.)

Такого рода отказы – возможно, не совсем то, что вы подразумевали, когда задумывали связать процессы. Однако именно так по умолчанию действуют связанные процессы Elixir, и это поведение имеет смысл в большинстве ситуаций. Но у вас есть возможность перехватить команду на выход. Когда процесс Elixir терпит неудачу, он посылает описание причины в форме кортежа связанному с ним процессу. Кортеж содержит атом `:EXIT`, идентификатор процесса, потерпевшего неудачу и ошибку в виде сложного кортежа. Если в процессе настроена обработка команды на выход вызовом `Process.flag(:trap_exit, true)`, такие описания будут передаваться ему в виде сообщений, не вызывая остановку процесса.

В примере 9.9 (*ch09/ex9-trapping*) показано, как можно изменить метод `mph_drop/0`, включив в него вызов `Process.flag` для установки флага и добавив еще одно предложение в инструкцию `receive` для обработки сообщения о завершении связанного процесса.

Пример 9.9 ❖ Перехват ошибки, вывод сообщения на экран и завершение

```
defmodule MphDrop do
  def mph_drop do
    Process.flag(:trap_exit, true)
    drop_pid = spawn_link(Drop, :drop, [])
    convert(drop_pid)
  end

  def convert(drop_pid) do
    receive do
      {planemo, distance} ->
        send(drop_pid, {self(), planemo, distance})
        convert(drop_pid)
      {:EXIT, pid, reason} ->
        IO.puts("Failure: #{inspect(pid)} #{inspect(reason)}")
      {planemo, distance, velocity} ->
        mph_velocity = 2.23693629 * velocity
        IO.write("On #{planemo}, a fall of #{distance} meters ")
        IO.puts("yields a velocity of #{mph_velocity} mph.")
        convert(drop_pid)
    end
  end
end
```

Если теперь повторить эксперимент с передачей некорректных данных, метод `convert/1` выведет сообщение об ошибке (практически дублируя сообщение оболочки) перед завершением:

```
iex(1)> pid1 = spawn(MphDrop, :mph_drop, [])
#PID<0.144.0>
iex(2)> send(pid1, {:moon, 20})
On moon, a fall of 20 meters
yields a velocity of 17.89549032 mph.
{:moon, 20}
iex(3)> send(pid1, {:moon, :zoids})
Failure: #PID<0.145.0> {:badarith, [{Drop, :fall_velocity, 2,
  [file: 'lib/drop.ex', line: 15]],
  {Drop, :drop, 0, [file: 'lib/drop.ex', line: 5]}}]
{:moon, :zoids}
iex(4)>
12:04:31.360 [error] Process #PID<0.145.0> raised an exception
** (ArithmeticError) bad argument in arithmetic expression
    (mph_drop) lib/drop.ex:15: Drop.fall_velocity/2
    (mph_drop) lib/drop.ex:5: Drop.drop/0

nil
iex(5)>
```

Более надежная альтернатива могла бы записать в переменную `drop_pid` новое значение, запустив новый процесс. Эта версия показана в примере 9.10 (*ch09/ex10-resilient*). Инstrukция `receive` в ней пропускает любые ошибки и запускает новый процесс (`new_drop_pid`) калькулятора.

Пример 9.10 ❖ Перехват ошибки, вывод сообщения на экран и запуск нового процесса

```
defmodule MphDrop do
  def mph_drop do
    Process.flag(:trap_exit, true)
    drop_pid = spawn_link(Drop, :drop, [])
    convert(drop_pid)
  end

  def convert(drop_pid) do
    receive do
      {planemo, distance} ->
        send(drop_pid, {self(), planemo, distance})
        convert(drop_pid)
      {:EXIT, _pid, _reason} ->
        new_drop_pid = spawn_link(Drop, :drop, [])
        convert(new_drop_pid)
    end
  end
end
```

```

{planemo, distance, velocity} ->
  mph_velocity = 2.23693629 * velocity
  IO.write("On #{planemo}, a fall of #{distance} meters ")
  IO.puts("yields a velocity of #{mph_velocity} mph.")
  convert(drop_pid)
end
end
end

```

Если скомпилировать и запустить пример 9.10, вы увидите окно Observer, как показано на рис. 9.9. Если передать некорректные данные, как показано в строке 6 ниже, на экране все так же появится сообщение об ошибке, но процесс продолжит работу, как можно видеть в окне Observer на рис. 9.10 (будет запущен новый процесс Drop.drop/0) и как демонстрирует строка 7:

```

iex(1)> pid1 = spawn(MphDrop, :mph_drop, [])
#PID<0.145.0>
iex(2)> :observer.start()
:ok
iex(3)> send(pid1, {:moon, 20})
On moon, a fall of 20 meters
yields a velocity of 17.89549032 mph.
{:moon,20}
iex(4)> send(pid1, {:mars, 20})
On mars, a fall of 20 meters
yields a velocity of 27.250254686571544 mph.
{:mars, 20}
iex(5)> send(pid1, {:mars, :zoids})
{:mars, :zoids}
Failure: #PID<0.109.0> {:badarith, [{Drop, :fall_velocity, 2,
  [file: 'lib/drop.ex', line: 19]],
  {Drop, :drop, 0, [file: 'lib/drop.ex', line: 5]}}]
iex(6)>
15:59:49.713 [error] Process #PID<0.146.0> raised an exception
** (ArithmeticError) bad argument in arithmetic expression
    (mph_drop) lib/drop.ex:19: Drop.fall_velocity/2
    (mph_drop) lib/drop.ex:5: Drop.drop/0

nil
iex(7)> send(pid1, {:moon, 20})
On moon, a fall of 20 meters
yields a velocity of 17.89549032 mph.
{:moon,20}

```

Pid	Name or Initial Func	Redts	Memory	MsgQ	Current Function
<0.146.0>	Elixir.Drop:drop/0	0	2744	0	Elixir.Drop:drop/0
<0.137.0>	Elixir.Logger	0	7272	0	Elixir.GenEvent:fetch_msg/5
<0.143.0>	erlang:apply/2	222	26568	0	Elixir.IEx.Evaluator:loop/3
<0.51.0>	erlang:apply/2	4	5912	0	Elixir.IEx.Server:wait_input/4
<0.145.0>	Elixir.MphDrop:mph_drop/0	0	2744	0	Elixir.MphDrop:convert/1
<0.135.0>	application_master:start_it/4	0	5792	0	application_master:loop_it/4
<0.84.0>	application_master:start_it/4	0	2784	0	application_master:loop_it/4
<0.66.0>	application_master:start_it/4	0	2784	0	application_master:loop_it/4

Рис. 9.9 ❖ Процессы до ошибки – обратите внимание на идентификатор процесса в верхней строке

Pid	Name or Initial Func	Redts	Memory	MsgQ	Current Function
<0.5023.0>	Elixir.Drop:drop/0	1	2744	0	Elixir.Drop:drop/0
<0.137.0>	Elixir.Logger	157	7272	0	Elixir.GenEvent:fetch_msg/5
<0.143.0>	erlang:apply/2	1441	55168	0	Elixir.IEx.Evaluator:loop/3
<0.51.0>	erlang:apply/2	8	7056	0	Elixir.IEx.Server:wait_input/4
<0.145.0>	Elixir.MphDrop:mph_drop/0	5	2744	0	Elixir.MphDrop:convert/1
<0.135.0>	application_master:start_it/4	0	5792	0	application_master:loop_it/4
<0.84.0>	application_master:start_it/4	0	2784	0	application_master:loop_it/4
<0.66.0>	application_master:start_it/4	0	2784	0	application_master:loop_it/4

Рис. 9.10 ❖ Процессы после ошибки – обратите внимание на идентификатор процесса в верхней строке

Elixir поддерживает много возможностей управления процессами. Вы можете разрывать связь вызовом `Process.unlink/1` или устанавливать ее вызовом `Process.monitor/1`. Завершить процесс можно вызовом `Process.exit/2`, указав идентификатор процесса и причину завершения. В качестве идентификатора можно передать `pid` другого процесса или результат `self()`.

Создание приложений, способных противостоять ошибкам и восстанавливать свою работоспособность, является основой отказоустойчивого программирования на Elixir. Разработка в таком стиле, вероятно, для большинства программистов станет еще более широким шагом, чем переход к функциональному программированию на Elixir, но именно в этом наиболее очевидно проявляется мощь Elixir.

Глава 10

Исключения, ошибки и отладка

«Позволь потерпеть аварию» – блестящая идея, но не для приложений, которыми вам придется управлять. Можно написать код, который постоянно будет восстанавливать работоспособность после аварий, но иногда проще написать и сопровождать код, который явно обрабатывает отказы. И, выбрав этот, второй путь, связанный с обработкой ошибок, вам определенно потребуется возможность отыскивать их причины в приложении.

Виды ошибок

Как вы уже видели, некоторые виды ошибок возникают на этапе компиляции. Иногда компилятор выводит предупреждения о возможных проблемах, как, например, об объявленных, но неиспользуемых переменных. Другие два распространенных вида ошибок: ошибки времени выполнения, возникающие во время выполнения кода и способные остановить работу функции или процесса, и логические ошибки, которые могут не приводить к остановке программы, но способны вызвать массу неприятностей.

Логические ошибки часто с большим трудом поддаются диагностике, требуют глубокого осмысления кода и, возможно, многих часов в компании с отладчиком, журналами и комплектами тестов. Иногда приходится потратить массу времени, чтобы распутать простую математическую ошибку. Бывает, что проблема связана с синхронизацией, когда операции выполняются не в той последовательности, в какой должны. Состояние гонки в худших случаях может вызывать

клинч и зависание, в менее серьезных ситуациях – приводить к ошибочным результатам и путанице.

Ошибки времени выполнения тоже доставляют много неприятностей, но они проще в диагностике. Иногда обработку ошибок времени выполнения можно сделать частью логики программы, но не стоит увлекаться этим. В Elixir, в отличие от многих других языков, обработка ошибок дает очень небольшие преимущества перед философией «позволь потерпеть аварию», когда вы позволяете ошибке прервать работу процесса и затем восстанавливаете его работоспособность, как было показано в примере 9.10.

Восстановление работоспособности после ошибок времени выполнения

Чтобы поймать ошибку времени выполнения как можно ближе к тому месту, где она произошла, заверните код, вызывающий сомнения, в конструкцию `try...rescue` и добавьте обработку проблем, которые могут возникнуть в этом коде. Она пояснит компилятору, что в этом коде может произойти нечто необычное, а вам позволит разобраться с последствиями.

Для простоты вернемся к примеру 3.1, который вычисляет скорость падения без учета возможности передачи отрицательной высоты. Функция `:math.sqrt/1` в этом случае вернет ошибку `badarith` (арифметическая ошибка). Пример 4.2 решает эту проблему путем введения ограничителя, но если необходимо нечто большее, чем простая блокировка выполнения, можно предпринять более прямой подход на основе инструкции `try/rescue`, как показано в примере 10.1 (*ch10/ex1-tryCatch*).

Пример 10.1 ❖ Использование инструкции `try/catch` для обработки возможных ошибок

```
defmodule Drop do
  def fall_velocity(planemo, distance) do
    gravity = case planemo do
      :earth -> 9.8
      :moon  -> 1.6
      :mars   -> 3.71
    end
    try do
      :math.sqrt(2 * gravity * distance)
    rescue
```

```

    error -> error
  end
end
end

```

Теперь вычисления выполняются внутри инструкции `try`. В случае успешного выполнения инструкции, следующей за `do`, ее результат станет возвращаемым значением функции.

Если в процессе вычислений возникнет ошибка, в данном случае из-за отрицательного аргумента, в игру вступит сопоставление с образом в предложении `rescue`. В этом случае возвращаемым значением станет переменная `error`, содержащая тип исключения и сообщение.

Опробуем это решение в интерактивной оболочке:

```

iex(1)> Drop.fall_velocity(:earth, 20)
19.79898987322333
iex(2)> Drop.fall_velocity(:earth, -20)
%ArithmeticError{message: "bad argument in arithmetic expression"}

```

В случае успеха мы просто получили результат. Но в случае ошибки на экране появилось сообщение об исключении. Решение пока неполное, но оно может служить основой для дальнейшего развития.

Внутри `try` может находиться несколько инструкций (практически так же, как в `case`). Но старайтесь включать в `try` как можно меньше кода, по крайней мере на начальном этапе, чтобы проще было определить, где происходит ошибка. Однако если вы хотите понаблюдать за запросами с атомами, не соответствующими известным планетам в инструкции `case`, заключите всю эту инструкцию в `try`:

```

defmodule Drop do
  def fall_velocity(planemo, distance) do
    try do
      gravity = case planemo do
        :earth -> 9.8
        :moon  -> 1.6
        :mars   -> 3.71
      end
      :math.sqrt(2 * gravity * distance)
    rescue
      error -> error
    end
  end
end

```

Если теперь попытаться указать неизвестную планету, вы увидите, что код перехватил ошибку (не забудьте скомпилировать код, чтобы задействовать новую версию):

```
iex(3)> Drop.fall_velocity(:jupiter, 20)
** (CaseClauseError) no case clause matching: :jupiter
   drop.ex:3: Drop.fall_velocity/2
iex(3)> r(Drop)
warning: redefining module Drop (current version defined in memory)
  lib/drop.ex:1

{:reloaded, Drop, [Drop]}
iex(4)> Drop.fall_velocity(:jupiter, 20)
%CaseClauseError{term: :jupiter}
```

Сообщение `CaseClauseError` указывает, что ошибка произошла из-за отсутствия совпадения в инструкции `case`, и сообщает найденный атом.

В `rescue` также можно включить несколько сопоставлений с образцом. Если фактическая ошибка не совпадет ни с одним из них, она будет действовать подобно обычной ошибке времени выполнения, как если бы инструкция `try` вообще отсутствовала. Следующий пример генерирует разные сообщения для каждого типа ошибок. Эта версия не сохраняет исключения в переменной, потому что не использует эту информацию:

```
defmodule Drop do
  def fall_velocity(planemo, distance) do
    try do
      gravity = case planemo do
        :earth -> 9.8
        :moon  -> 1.6
        :mars  -> 3.71
      end
      :math.sqrt(2 * gravity * distance)
    rescue
      ArithmeticError -> {error, "Distance must be non-negative"}
      CaseClauseError -> {error, "Unknown planemo #{planemo}"}
    end
  end
end
```

И вот как действует эта версия:

```
iex(5)> r(Drop)
warning: redefining module Drop (current version defined in memory)
  lib/drop.ex:1
```

```
{:reloaded, Drop, [Drop]}
iex(6)> Drop.fall_velocity(:earth, -20)
{:error, "Distance must be non-negative"}
iex(7)> Drop.fall_velocity(:jupiter, 20)
{:error, "Unknown planemo jupiter"}
```



Если для разных исключений потребуется выполнять одни и те же действия, это можно реализовать так:

```
[ArithmeticError, CaseClauseError] -> "Generic Error"
err in [ErlangError, RuntimeError] -> {:error, err}
```

Если код, способный вызвать исключение, может создать путаницу, попробуйте добавить предложение `after` после `rescue` и перед закрывающим ключевым словом `end`. Код в предложении `after` гарантированно будет выполняться в обоих случаях – успеха или отказа, и это отличное место для создания любых побочных эффектов. Кроме того, код в `after` не влияет на возвращаемое значение инструкции `try`.

Журналирование результатов выполнения и ошибок

Функцию `IO.puts` удобно использовать для вывода простых сообщений в окне командной оболочки, но с ростом программ (особенно с появлением распределенных процессов) простой вывод текста в стандартный вывод едва ли сможет обеспечить вас необходимой информацией. Для решения этой проблемы Elixir предлагает набор функций журналирования. Они способны подключаться к более сложным системам журналирования, но для начала попробуем использовать их как средство структурирования сообщений.

Функции в модуле `Logger` языка Elixir поддерживают четыре уровня журналирования:

- `:info` – для вывода информации любого вида;
- `:debug` – для вывода отладочных сообщений;
- `:warn` – для вывода предупреждений, в ответ на которые кто-то в конечном итоге должен предпринять какие-то действия;
- `:error` – для вывода сообщений об ошибках, требующих обязательного решения.

Как показано ниже, эти вызовы производят сообщения, отличающиеся визуально. Если запустить оболочку `IE` и ввести следующие инструкции, вы увидите также, что сообщения отображаются разными цветами:

```

iex(1)> require Logger
Logger
iex(2)> counter=255
255
iex(3)> Logger.info("About to begin test")
18:57:36.846 [info] About to begin test
:ok
iex(4)> Logger.debug("Current value of counter is #{counter}")
18:58:06.526 [debug] Current value of counter is 255
:ok
iex(5)> Logger.warn("Connection lost; will retry.")
18:58:21.759 [warn] Connection lost; will retry.
:ok
iex(6)> Logger.error("Unable to read database.")
18:58:37.008 [error] Unable to read database.
:ok

```

Эти функции лишь немного совершенствуют вывод в сравнении с `IO.puts`, тогда зачем их использовать? Дело в том, что эти функции дают намного больше, чем кажется. По умолчанию в момент запуска Elixir настраивает модуль `Logger` на вывод в окно интерактивной оболочки. Однако есть возможность подключить свой механизм хранения журналируемой информации.

Сама возможность журналировать информацию очень полезна, но в связи с этим часто возникает вопрос: что журналировать и где. Вы можете разбросать инструкции журналирования по всему своему коду или прибегнуть к помощи отладчика Erlang, который также доступен в Elixir.

Трассировка сообщений

Elixir предлагает широкий выбор инструментов трассировки кода, и в виде дополнительного кода, добавляемого в программу (встроенные функции Erlang `trace` и `trace_pattern`), и в виде текстового отладчика. Наиболее простым в комплекте является модуль `dbg`. Он дает возможность указать, что требуется отследить, и выводит результаты трассировки в окне интерактивной оболочки.



Модуль `:dbg` – это модуль языка Erlang, но если вы установите вводить двоеточие каждый раз, когда обращаетесь к функции Erlang, можете реализовать синтаксис, ближе напоминающий синтаксис Elixir, выполнив следующую команду:

```
iex(1)> alias :dbg, as: Dbg
:dbg
```

А пока мы продолжим использовать :dbg.

Для начала попробуем проследить сообщения, пересылаемые между процессами. Используем :dbg.p для трассировки сообщений, передаваемых между процессом mph_drop из примера 9.8 и процессом drop из примера 9.6. После компиляции модулей вызовем :dbg.tracer(), чтобы запустить вывод трассировочной информации в окно оболочки. Затем запустим процесс mph_drop, как обычно, и передадим его идентификатор pid процессу :dbg.p/2. Вторым аргументом здесь будет атом :m, означающий, что трассировке должны подвергаться сообщения (messages). Код из примера 9.8 повторяется в *ch10/ex2-debug*, и перед экспериментом должна быть выполнена команда `iex -S mix`:

```
iex(1)> :dbg.tracer()
{:ok, #PID<0.71.0>}
iex(2)> pid1 = spawn(MphDrop, :mph_drop, [])
#PID<0.148.0>
iex(3)> :dbg.p(pid1, :m)
{:ok, [{:matched, :nonode@nohost, 1}]}
```

Теперь, после отправки сообщения процессу mph_drop, в окне оболочки появится поток записей, описывающих порядок передачи сообщений (<0.148.0> — это процесс mph_drop, а <0.149.0> — процесс drop):

```
iex(4)> send(pid1, {:moon, 20})
On moon, a fall of 20 meters {:moon, 20}
(<0.148.0>) << {moon,20}
yields a velocity of 17.89549032 mph.
(<0.148.0>) <0.149.0> ! {<0.148.0>,moon,20}
(<0.148.0>) << {moon,20,8.0}
(<0.148.0>) <0.50.0> ! {io_request,<0.148.0>,#Ref<0.0.2.159>,
                        {put_chars,unicode,
                          <<"On moon, a fall of 20 meters ">>}}
(<0.148.0>) << {io_reply,#Ref<0.0.2.159>,ok}
(<0.148.0>) <0.50.0> ! {io_request,<0.148.0>,#Ref<0.0.3.350>,
                        {put_chars,unicode,
                          <<"yields a velocity of 17.89549032 mph.\n">>}}
(<0.148.0>)<< {io_reply,#Ref<0.0.3.350>,ok}
```

Группа символов << указывает на идентификатор процесса, получившего сообщение. Идентификатор процесса-отправителя отмечается восклицательным знаком, за которым следует сообщение. Очень похоже на то, что мы наблюдали, когда использовали инструмент Ob-

server для наблюдения за обработкой сообщений в разделе «Наблюдение за процессами» в главе 9. Так как `:dbg` – это модуль Erlang, значения в сообщениях (посылаемых и принимаемых) отображаются с использованием синтаксиса языка Erlang, а не Elixir. Ниже приводятся некоторые пояснения к полученным результатам:

- в этом эксперименте процесс `mph_drop` сообщает, что `On moon, a fall of 20 meters yields a velocity of 17.89549032 mph` (на Луне объект, падающий с высоты 20 метров, разовьет скорость 17,89549032 миль/ч), а результат `{:moon, 20}` находится в середине трассировочной записи. Остальная часть трассировки описывает порядок передачи сообщений;
- `mph_drop (<0.148.0>)` принял сообщение-кортеж `{moon, 20}`;
- затем послал сообщение-кортеж `{<0.148.0>, moon, 20}` процессу `drop` с идентификатором `<0.149.0>`;
- `mph_drop` принял кортеж `{moon, 20, 8.0}` (от процесса `drop`);
- затем вызвал функцию `io:request/2`, которая породила группу других сообщений для формирования вывода.

Трассировочная информация перемежается с фактическим выполнением кода, но она позволяет четко определить, как выглядит поток сообщений. С течением времени вы научитесь использовать многие другие возможности `:dbg` для трассировки своего кода и, возможно даже, с помощью сопоставления с образцом и функции `trace` создадите собственную систему трассировки конкретного кода.

Трассировка вызовов функций

Если вам требуется лишь понаблюдать, как передаются аргументы между функциями, для этого можно использовать простую функцию `IO.puts`, как было показано в главе 4, где мы следили за ходом выполнения рекурсивных функций. Тем не менее ниже мы рассмотрим еще один способ, и снова с применением модуля `:dbg`.

Код в примере 4.11, поиска факториала с использованием логики прямого отсчета, запускается вызовом функции `Fact.factorial/1`, которая затем вызывает рекурсивную функцию `Fact.factorial/3`. Модуль `:dbg` позволяет проследить фактические вызовы функций и их аргументы, перемежающиеся отладочными вызовами `IO.puts`. (Исходный код примера находится в папке `ch10/ex3-debug`.)

Трассировка функций немного сложнее трассировки сообщений, потому что функции `:dbg.p/2` нельзя просто передать `pid` процесса. Как показано в строке 2, в примере ниже, ей нужно сообщить, что вы

собираетесь получить информацию по всем (:all) процессам и вызовам (:c, от англ. *call* – *вызов*). После этого нужно указать, какие именно вызовы вас интересуют, используя :dbg.tpl, как показано в строке 3. Эта функция принимает имя модуля (Fact), имя функции в виде атома (:factorial) и необязательную спецификацию сопоставления, которая позволяет точнее определить аргументы. Разновидность этой функции также позволяет указать аргументы.

Итак, запустим трассировщика, сообщим ему, что желаем понаблюдать за вызовами функций, и укажем функцию (или функции, несколькими вызовами :dbg.tpl) для наблюдения. Затем вызовем функцию и посмотрим, что из этого получилось:

```
iex(1)> :dbg.tracer()
{:ok, #PID<0.43.0>}
iex(2)> :dbg.p(:all, :c)
{:ok, [{:matched, :nonode@nohost, 51}]}
iex(3)> :dbg.tpl(Fact, :factorial, [])
{:ok, [{:matched, :nonode@nohost, 2}]}

iex(4)> Fact.factorial(4)
1 yields 1.
(<0.26.0>) call 'Elixir-Fact':factorial(4)
(<0.26.0>) call 'Elixir-Fact':factorial(1,4,1)
2 yields 2.
(<0.26.0>) call 'Elixir-Fact':factorial(2,4,1)
3 yields 6.
(<0.26.0>) call 'Elixir-Fact':factorial(3,4,2)
4 yields 24.
(<0.26.0>) call 'Elixir-Fact':factorial(4,4,6)
finished!
(<0.26.0>) call 'Elixir-Fact':factorial(5,4,24)
24
```

Как видите, вывод трассировщика перемешался с выводом от функции IO.puts. Из-за того, что трассировщик выполняется в своем процессе (<0.43.0>), отдельно от процесса с трассируемой функцией (<0.26.0>), его вывод может не точно совпадать с отладочными сообщениями.

Закончив трассировку, не забудьте вызвать :dbg.stop/0 (если вы предполагаете повторно запустить трассировку с теми же параметрами) или :dbg.stop_clear/0 (если предполагается, что следующий запуск трассировщика будет производиться с другими настройками).

Модуль :dbg и функции trace, на которых он основан, – необычайно мощные инструменты.

Глава 11

Статический анализ, спецификации типов и тестирование

В программировании известны три основных класса ошибок: *синтаксические ошибки*, *ошибки времени выполнения* и *семантические ошибки*. Компилятор Elixir обнаруживает и сообщает о синтаксических ошибках, а в главе 10 вы узнали, как обрабатывать ошибки времени выполнения. Остаются *логические ошибки*, когда вы требуете от Elixir выполнить что-то, хотя имели в виду совсем другое. Журналирование и трассировка могут помочь в поиске логических ошибок, но все же лучше постараться предотвратить их появление с самого начала. И в этом вам помогут статический анализ, спецификации типов и модульное тестирование.

Статический анализ

Под статическим анализом понимается отладка путем анализа исходного кода без запуска программы. Dialyzer (DIscrepancy AnalYZer for Erlang – анализатор несоответствий для программ на Erlang, <http://erlang.org/doc/man/dialyzer.html>) – это инструмент статического анализа программного кода на Erlang и файлов *.beam*, позволяющий выявлять такие проблемы, как неиспользуемые функции, никогда не выполняемый код, неправильные списки, неиспользуемые или не имеющие совпадений образцы и т. д. Чтобы упростить использование Dialyzer для анализа кода на языке Elixir, был создан инструмент

Dialyxir (<https://github.com/jeremyjh/dialyxir>). Мы выполнили его установку с помощью следующих команд:

```
$ git clone https://github.com/jeremyjh/dialyxir
$ cd dialyxir
$ mix archive.build
$ mix archive.install
$ mix dialyzer.plt
```

Последняя команда создает *хранимую таблицу поиска* (Persistent Lookup Table, PLT), где хранятся результаты анализа с помощью Dialyzer часто используемых библиотек Erlang и Elixir, чтобы их не приходилось анализировать снова и снова. Этот этап занимает несколько минут, поэтому наберитесь терпения и сделайте короткий перерыв. Согласно инструкциям на сайте GitHub, эту команду следует запускать повторно всякий раз, после установки новых версий Elixir или Erlang.

Теперь, после установки Dialyxir, посмотрим, как он работает. Взгляните на код в примере 11.1 (*ch11/ex1-guards*), добавляющий ошибочную функцию в пример *ch03/ex2-guards*.

Пример 11.1 ❖ Ошибочные вызовы предложений

```
defmodule Drop do
```

```
  def fall_velocity(:earth, distance) when distance >= 0 do
    :math.sqrt(2 * 9.8 * distance)
  end

  def fall_velocity(:moon, distance) when distance >= 0 do
    :math.sqrt(2 * 1.6 * distance)
  end

  def fall_velocity(:mars, distance) when distance >= 0 do
    :math.sqrt(2 * 3.71 * distance)
  end

  def wrongness() do
    total_distance = fall_velocity(:earth, 20) +
      fall_velocity(:moon, 20) +
      fall_velocity(:jupiter, 20) +
      fall_velocity(:earth, "20")
    total_distance
  end
end
```

Компилятор не обнаружит никаких проблем в этом модуле. Они проявятся только при попытке вызвать функцию *wrongness/0*:

```
$ iex -S mix
Erlang/OTP 19 [erts-8.0] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe]
[kernel-poll:false]

Compiling 1 file (.ex)
Generated drop app
Interactive Elixir (1.3.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> Drop.wrongness()
** (FunctionClauseError) no function clause matching in Drop.fall_velocity/2
    (drop) lib/drop.ex:3: Drop.fall_velocity(:jupiter, 20)
    (drop) lib/drop.ex:18: Drop.wrongness/0
iex(1)>
```

Однако если передать этот модуль инструменту Dialyzer, он предупредит об имеющейся проблеме. Сначала выполним команду `mix clean`, чтобы удалить все скомпилированные файлы и дать Dialyzer возможность «начать с нуля»:

```
$ mix clean
$ mix dialyzer
Compiling 1 file (.ex)
Generated drop app
Starting Dialyzer
dialyzer --no_check_plt --plt /home/david/.dialyir_core_19_1.3.1.plt
-Wunmatched_returns -Werror_handling -Wrace_conditions -Wunderspecs
/Users/elixir/code/ch11/ex1-guards/_build/dev/lib/drop/ebin
Proceeding with analysis...
drop.ex:15: Function wrongness/0 has no local return
drop.ex:18: The call 'Elixir.Drop':fall_velocity('jupiter',20) will never
return
    since it differs in the 1st argument from the success typing
    arguments: ('earth' | 'mars' | 'moon',number())
drop.ex:19: The call 'Elixir.Drop':fall_velocity('earth',<<_:16>>) will
never return since it differs in the 2nd argument from the success
    typing arguments: ('earth' | 'mars' | 'moon',number())
done in 0m3.27s
done (warnings were emitted)
```

Dialyzer скомпилировал файл и проверил его. Все элементы в командной строке `dialyzer`, начинающиеся с `-W`, определяют, какие предупреждения должен вывести анализатор, в дополнение к стандартным.

Первая ошибка, `Function wrongness/0 has no local return` (Функция `wrongness/0` не имеет локального возвращаемого значения), означает, что функция ничего не возвращает, потому что в ней имеются другие ошибки. (Если бы в функции `wrongness/0` отсутствовали ошибки, но

она вызывалась бы из функций, имеющих ошибки, Dialyzer вывел бы то же самое сообщение.)

Вторая ошибка сообщает, что вызов функции `fall_velocity(:jupiter, 20)` (которую Dialyzer обозначил иначе, так как принадлежит вселенной Erlang) не сработает, потому что в ней отсутствует образец `:jupiter`, соответствующий первому аргументу.

Последняя ошибка демонстрирует всю мощь Dialyzer. Даже при том, что в определении `drop/1` отсутствует спецификатор типа `@spec`, Dialyzer, как по волшебству, догадался, что второй аргумент должен быть числом и, соответственно, вызов `fall_velocity(:earth, "20")` оформлен неправильно. (Ладно-ладно, не по волшебству, но алгоритм определения типов действительно крут!)

Спецификации типов

Dialyzer – превосходный инструмент и способен многое сделать самостоятельно. Однако вы тоже можете помочь ему еще лучше справиться со своей работой (а заодно и всем тем, кто будет читать ваш код), явно обозначив типы параметров и возвращаемых значений в ваших функциях. Взгляните на модуль в примере 11.2 (*ch11/ex2-specs*), реализующий некоторые уравнения, связанные с ускорением свободного падения.

Пример 11.2 ❖ Явное определение типов

```
defmodule Specs do
```

```
  @spec fall_velocity({atom(), number()}, number()) :: float()
  def fall_velocity({_planemo, gravity}, distance) when distance > 0 do
    :math.sqrt(2 * gravity * distance)
  end

  @spec average_velocity_by_distance({atom(), number()}, number()) :: float()
  def average_velocity_by_distance({planemo, gravity}, distance) when distance
> 0 do
    fall_velocity({planemo, gravity}, distance) / 2.0
  end

  @spec fall_velocity({atom(), number()}, number()) :: float()
  def fall_velocity({_planemo, gravity}, time) when time > 0 do
    gravity * time * time / 2.0
  end

  def calculate() do
    earth_v = average_velocity_by_distance({:earth, 9.8}, 10)
    moon_v = average_velocity_by_distance({:moon, 1.6}, 10)
```

```

mars_v = average_velocity_by_distance({3.71, :mars}, 10)
IO.puts("After 10 seconds, average velocity is:")
IO.puts("Earth: #{earth_v} m.")
IO.puts("Moon: #{moon_v} m.")
IO.puts("Mars: #{mars_v} m.")
end
end

```

Каждая из этих функций принимает в первом параметре кортеж с названием планеты и ускорением свободного падения на ней и высоту или время во втором параметре. Возможно, вы заметили, что функция `calculate/0` содержит ошибку в инструкции, вычисляющей скорость объекта на Марсе (`mars_v`); элементы в первом кортеже перепутаны местами. Компилятор не обнаружит эту ошибку, поэтому в IEх получится следующий результат:

```

iex(1)> Specs.calculate()
** (ArithmeticError) bad argument in arithmetic expression
    (specs) lib/specs.ex:5: Specs.fall_velocity/2
    (specs) lib/specs.ex:10: Specs.average_velocity_by_distance/2
    (specs) lib/specs.ex:21: Specs.calculate/0
iex(1)>

```

Однако Dialyzer благодаря наличию спецификаций `@spec` сможет обнаружить эту проблему:

```
$ mix dialyzer
```

```
Starting Dialyzer
```

```

dialyzer --no_check_plt --plt /home/david/.dialyxi_core19_1.3.1.plt
-Wunmatched_returns -Werror_handling -Wrace_conditions -Wunderspecs
/Users/elixir/code/ch11/ex2-specs/_build/dev/lib/specs/ebin
Proceeding with analysis...

```

```
specs.ex:23: Function calculate/0 has no local return
```

```
specs.ex:26:
```

```

The call 'Elixir.Specs':average_velocity_by_distance(
  {3.7099999999999996447, 'mars'},10)
will never return since the success typing is
({atom(),number()},number()) -> float()
and the contract is ({atom(),number()},number()) -> float()
done in 0m3.04s

```

```
done (warnings were emitted)
```



Это один из случаев, когда использование спецификатора `@spec` ухудшает ясность сообщений, выводимых инструментом Dialyzer, — без `@spec` он вывел бы следующее сообщение:

```

specs.ex:26: The call 'Elixir.Specs':average_velocity_by_
distance({3.7099999999999996447, 'mars'},10)

```

```
will never return since it differs in the 1st argument
from the success typing arguments:
({atom()},number()),number())
```

Но в любом случае Dialyzer предупредит вас о проблеме.

В спецификаторах `@spec` присутствует много повторений. От них можно избавиться, определив собственную спецификацию типа. В примере 11.3 (*ch11/ex3-type*) определяется тип `planetuple`, который затем указывается во всех директивах `@spec`.

Пример 11.3 ❖ Определение спецификации типа

```
defmodule NewType do
  @type planetuple :: {atom(), number()}

  @spec fall_velocity(planetuple, number()) :: float()
  def fall_velocity({_planemo, gravity}, distance) when distance > 0 do
    :math.sqrt(2 * gravity * distance)
  end

  @spec average_velocity_by_distance(planetuple, number()) :: float()
  def average_velocity_by_distance({_planemo, gravity}, distance) when distance
> 0 do
    fall_velocity({_planemo, gravity}, distance) / 2.0
  end

  @spec fall_distance(planetuple, number()) :: float()
  def fall_distance({_planemo, gravity}, time) when time > 0 do
    gravity * time * time / 2.0
  end

  def calculate() do
    earth_v = average_velocity_by_distance({:earth, 9.8}, 10)
    moon_v = average_velocity_by_distance({:moon, 1.6}, 10)
    mars_v = average_velocity_by_distance({3.71, :mars}, 10)
    IO.puts("After 10 seconds, average velocity is:")
    IO.puts("Earth: #{earth_v} m.")
    IO.puts("Moon: #{moon_v} m.")
    IO.puts("Mars: #{mars_v} m.")
  end
end
```

Если требуется, чтобы определение типа было приватным, используйте `@typed` вместо `@type`; если необходимо сделать его общедоступным, но с сокрытием внутренней структуры, используйте `@opaque`. Полный список всех встроенных спецификаций типов, а также спецификаций, определяемых Elixir, можно найти в документации (<https://hexdocs.pm/elixir/typespecs.html>).

Модульные тесты

В дополнение к статическому анализу и спецификаторам типов для функций `@spec` лишних затрат времени на отладку можно также избежать, организовав предварительное тестирование своего кода. Этой цели в языке Elixir служит модуль с именем `ExUnit`, упрощающий реализацию модульных тестов.

Для демонстрации `ExUnit` мы с помощью `Mix` создали новый проект с именем `drop`. В файл `lib/drop.ex` мы поместили модуль `Drop`, содержащий ошибку: в определении константы ускорения свободного падения для Марса допущена опечатка, и теперь она имеет значение 3.41 вместо 3.71 (у набиравшего число на цифровой клавиатуре просто соскользнул палец):

```
defmodule Drop do
  def fall_velocity(planemo, distance) do
    gravity = case planemo do
      :earth -> 9.8
      :moon  -> 1.6
      :mars  -> 3.41
    end
    :math.sqrt(2 * gravity * distance)
  end
end
```

Кроме каталога `lib`, диспетчер проектов `Mix` уже создал каталог `test`. Заглянув в него, можно найти два файла с расширением `.exs`: `test_helper.exs` и `drop_test.exs`. Расширение `.exs` указывает, что это файлы сценариев, скомпилированные версии которых не записываются на диск. Файл `test_helper.exs` подготавливает `ExUnit` к работе. В файле `drop_test.exs` вы можете определить свои тесты, используя макрос `test`. Ниже приводятся два примера таких тестов. Первый проверяет возврат нулевой скорости, когда высота равна нулю, а второй проверяет правильность вычисления скорости объекта, упавшего на Марсе с высоты 10 метров. Сохраните этот код в файле `drop_test.exs`:

```
defmodule DropTest do
  use ExUnit.Case, async: true

  test "Zero distance gives zero velocity" do
    assert Drop.fall_velocity(:earth, 0) == 0
  end

  test "Mars calculation correct" do
    assert Drop.fall_velocity(:mars, 10) == :math.sqrt(2 * 3.71 * 10)
  end
end
```


Конструкция `async: true` в строке `use` разрешает Elixir выполнять наборы тестов параллельно. В данном случае тестируется единственный модуль, поэтому имеется только один набор тестов.

Тест начинается с макроса `test`, за которым следует строка с описанием. Тело теста составляет некоторый код, проверяющий определенное условие. Тест считается успешным, если все проверки увенчались успехом. И напротив, тест считается проваленным, если хотя бы одна проверка не увенчалась успехом. В частности, проверка `assert` считается пройденной, когда выполняемый в ней код возвращает истинное значение, и проваленной, когда код возвращает ложное значение. Чтобы запустить тестирование, введите команду `mix test`:

```
$ mix test
Compiling 1 file (.ex)
Generated drop app
.

1) test Mars calculation correct (DropTest)
   test/drop_test.exs:8
   Assertion with == failed
   code: Drop.fall_velocity(:mars, 10) == :math.sqrt(2 * 3.71 * 10)
   lhs:  8.258329128825032
   rhs:  8.613942186943213
   stacktrace:
     test/drop_test.exs:9: (test)

Finished in 0.06 seconds
2 tests, 1 failure

Randomized with seed 585665
```

Начальная строка с точкой (.) показывает состояние каждого теста; одна точка (.) означает, что только один тест выполнен успешно.

Исправьте ошибку в модуле `Drop`, заменив константу с ускорением свободного падения для Марса правильным значением 3.71. Затем запустите тестирование еще раз, и вы должны увидеть, что оба теста пройдены успешно:

```
$ mix test
Compiling 1 file (.ex)
..

Finished in 0.05 seconds
2 tests, 0 failures

Randomized with seed 811304
```

Помимо `assert/1`, можно также использовать макрос `refute/1`, который ожидает получить ложное значение в случае успеха. Оба макроса – `assert/1` и `refute/1` – автоматически генерируют соответствующие сообщения. Кроме того, каждый из них имеет версию с двумя аргументами, которая дает возможность определить сообщение для вывода в случае неудачи.

При проверке операций над числами с плавающей точкой часто бессмысленно сравнивать результат с точным значением. В таких случаях можно использовать макрос `assert_in_delta/4`. В четырех аргументах ему передаются: ожидаемое значение, фактически вычисленное значение, значение погрешности и текст сообщения. Если ожидаемое и полученное значения отличаются не больше, чем на величину *погрешности*, тест считается успешным. Иначе `ExUnit` выведет ваше сообщение. Следующий тест проверяет, достаточно ли близка скорость падения объекта с высоты 1 метр на Земле к 4,4 метра в секунду. Вы можете добавить этот тест в файл `drop_test.exs` или (как это сделали мы) создать новый файл с именем `drop2_test.exs` в каталоге `test`:

```
defmodule Drop2Test do
  use ExUnit.Case, async: true
  test "Earth calculation correct" do
    calculated = Drop.fall_velocity(:earth, 1)
    assert_in_delta calculated, 4.4, 0.05,
      "Result of #{calculated} is not within 0.05 of 4.4"
  end
end
```

Если вам интересно увидеть свое сообщение об ошибке, уменьшите в новом тесте величину погрешности и сохраните файл (эта версия находится в файле `ch11/ex4-testing/test/drop3_test.exs`):

```
defmodule Drop2Test do
  use ExUnit.Case, async: true
  test "Earth calculation correct" do
    calculated = Drop.fall_velocity(:earth, 1)
    assert_in_delta calculated, 4.4, 0.0001,
      "Result of #{calculated} is not within 0.05 of 4.4"
  end
end
```

Вот что получается в результате:

```
$ mix test
..
```

```
1) test Earth calculation correct (Drop3Test)
```

```
test/drop3_test.exs:4
Result of 4.427188724235731 is not within 0.0001 of 4.4
stacktrace:
  test/drop3_test.exs:6: (test)
```

```
.
Finished in 0.08 seconds
4 tests, 1 failure
```

```
Randomized with seed 477713
```

Кроме всего прочего, есть возможность проверить возбуждение исключений в вашем коде. Следующие два теста проверяют возбуждение исключений при передаче неправильной планеты и отрицательной высоты. В каждом тесте проверяемый код завернут в анонимную функцию. Эти дополнительные тесты находятся в файле *ch11/ex4-testing/test/drop4_test.exs*:

```
defmodule Drop4Test do
  use ExUnit.Case, async: true
  test "Unknown planet causes error" do
    assert_raise CaseClauseError, fn ->
      Drop.fall_velocity(:planetX, 10)
    end
  end

  test "Negative distance causes error" do
    assert_raise ArithmeticError, fn ->
      Drop.fall_velocity(:earth, -10)
    end
  end
end
```

Настройка тестов

Можно написать код, который должен выполняться до и после каждого теста, а также перед началом сеанса тестирования и по его завершении. Например, перед выполнением любого теста можно устанавливать соединение с сервером, а по завершении теста – разрывать его.

Код, выполняемый перед началом тестирования, определяется в функции обратного вызова `setup_all`. Эта функция должна возвращать атом `:ok` и, необязательно, список ключей, добавленных в *контекст тестирования*, доступный в тестах под именем `Map`. Взгляните на следующий код (*ch11/ex5-setup*):

```

setup_all do
  IO.puts "Beginning all tests"

  on_exit fn ->
    IO.puts "Exit from all tests"
  end

  {:ok, [connection: :fake_PID]}
end

```

Этот код добавляет в контекст тестирования ключ `:connection`; макрос `on_exit` определяет код для выполнения после завершения всех тестов.

Код для запуска до и после каждого теста определяется посредством `setup`. Он имеет доступ к контексту тестирования:

```

setup context do
  IO.puts "About to start a test. Connection is #{Map.get(context,
    :connection)}"

  on_exit fn ->
    IO.puts "Individual test complete."
  end

  :ok
end

```

Наконец, внутри каждого теста можно обратиться к контексту, как показано ниже:

```

test "Zero distance gives zero velocity", context do
  IO.puts "In zero distance test. Connection is #{Map.get(context,
    :connection)}"
  assert Drop.fall_velocity(:earth,0) == 0
end

```

Вот как выглядят результаты тестирования:

```

$ mix test
Beginning all tests
About to start a test. Connection is fake_PID
In zero distance test, connection is fake_PID
Individual test complete.
.About to start a test. Connection is fake_PID
Test two
Individual test complete.
.Exit from all tests

Finished in 0.08 seconds
2 tests, 0 failures

Randomized with seed 519579

```

Встраивание тестов в документацию

Существует еще один способ тестирования: встраивание тестов в документацию с описанием функций и модулей. Такие тесты называют *встроенными* (doctest). В этом случае содержимое сценария тестирования имеет следующий вид:

```
defmodule DropTest do
  use ExUnit.Case, async: true
  doctest Drop
end
```

Вслед за макросом `doctest` указывается имя тестируемого модуля. `doctest` просмотрит документацию с описанием модуля, отыщет строки, напоминающие команды в оболочке IEx и их вывод. Такие строки начинаются с `iex>` или `iex(n)>`, где n – это число; а следующая строка интерпретируется как вывод. Пустые строки интерпретируются как начало нового теста. Ниже демонстрируется пример встроенного теста (*ch11/ex6-doctest*):

```
defmodule Drop do

  @doc """
  вычисляет скорость падающего объекта на указанном объекте
  плането (объекте с планетарной массой)

  iex(1)> Drop.fall_velocity(:earth, 10)
  14.0

  iex(2)> Drop.fall_velocity(:mars, 20)
  12.181953866272849

  iex> Drop.fall_velocity(:jupiter, 10)
  ** (CaseClauseError) no case clause matching: :jupiter
  """

  def fall_velocity(planemo, distance) do
    gravity = case planemo do
      :earth -> 9.8
      :moon  -> 1.6
      :mars  -> 3.71
    end
    :math.sqrt(2 * gravity * distance)
  end
end
```

Средства тестирования в Elixir позволяют также проверять прием сообщений, писать функции, общие для нескольких тестов, и многое другое. Более подробное описание можно найти в документации к Elixir.

Глава 12

Хранение структурированных данных

Несмотря на общую тенденцию стараться избегать побочных эффектов в программах на Elixir, хранение и совместное использование данных является фундаментальным побочным эффектом, необходимым в самых разных проектах.

Так как Elixir основан на Erlang и прекрасно взаимодействует с кодом на Erlang, для хранения данных и управления ими можно использовать ETS (Erlang Term Storage – хранилище термов Erlang) или базу данных Mnesia, предоставляющую возможность распределенного хранения.

Записи: структурирование данных до появления структур

Как было показано в разделе «От отображений к структурам» в главе 7, структуры в языке Elixir позволяют использовать имена для связывания данных вместо порядковых номеров (как в кортежах). Однако структуры основаны на отображениях, новом типе данных для Erlang и Elixir. До появления отображений в Erlang использовалось другое решение для хранения структурированных данных – записи. По аналогии со структурами данные в записях можно читать, изменять и сопоставлять с образцом, не заботясь о том, где в кортеже находится поле, или о том, что кто-то мог добавить новое поле.

Записи не пользуются особой любовью программистов на Erlang. Они поддерживаются в Elixir, но не рекомендуются к использованию. Требования к определению записей довольно сложны. Однако записи широко используются в библиотеках на Erlang и работают весьма эффективно, поэтому вам определенно стоит познакомиться с ними. По крайней мере, вы лучше будете понимать, о чем идет речь в обсуждениях структур данных в Elixir и Erlang.



Записи основаны на кортежах, и иногда Elixir открывает доступ к ним. Не пытайтесь использовать эти кортежи непосредственно, иначе к непростому синтаксису записей вы добавите все проблемы, сопутствующие кортежам.

Определение записей

Чтобы использовать запись, нужно сообщить языку Elixir ее строение в специальном объявлении `defrecord`:

```
defrecord Planemo, name: :nil, gravity: 0, diameter: 0, distance_from_sun: 0
```

Это объявление определяет запись типа `Planemo`, содержащую поля `name`, `gravity`, `diameter` и `distance_from_sun` с соответствующими значениями по умолчанию. Следующее объявление определяет записи для разных башен для сбрасывания объектов:

```
defrecord Tower, location: "", height: 20, planemo: :earth, name: ""
```

В отличие от объявлений `defstruct`, объявления записей часто приходится использовать сразу в нескольких модулях и даже (например, в этой главе) в интерактивной оболочке. Чтобы обеспечить возможность совместного использования записей в разных модулях, поместите их объявления в отдельный файл с расширением `.ex`. В зависимости от своих потребностей вы можете поместить каждое объявление в свой, отдельный файл или все объявления в один файл.

Для начала давайте попробуем поместить оба объявления в один файл, `records.ex`, как показано в примере 12.1 (*ch12/ex1-records*).

Пример 12.1 ❖ Файл `records.ex` с двумя объявлениями записей

```
defmodule Planemo do
  require Record
  Record.defrecord :planemo, [name: :nil, gravity: 0, diameter: 0,
    distance_from_sun: 0]
end

defmodule Tower do
  require Record
```

```
Record.defrecord :tower, Tower,
  [location: "", height: 20, planemo: :earth, name: ""]
end
```

Конструкция `Record.defrecord` определяет макрос для создания записей и доступа к ним. Первый элемент, следующий за `Record.defrecord`, – это имя записи. Второй элемент можно опустить; это *тег*. Если тег не указан, Elixir будет использовать имя записи. В данном случае мы указали тег для записи `Tower`, но опустили для записей `Planemo`. За именем и необязательным тегом следует список, определяющий пары ключей и значений по умолчанию. Elixir автоматически встраивает в модули функции для создания новых записей, чтения и изменения значений полей. Так как записи определяются в собственных модулях, чтобы сделать их доступными в вашей программе, необходимо обеспечить их компиляцию в том же каталоге, где находятся другие ваши модули. Для компиляции объявлений `defrecord` из командной строки можно использовать программу `elixirc` или доверить эту работу диспетчеру Mix, выполняя команду `iex -S mix`.

После этого интерактивная оболочка будет знать о существовании записей с именами `Planemo` и `Tower`, но чтобы получить возможность использовать их в программе или в интерактивной оболочке, эти записи необходимо загрузить с помощью `require`.



Записи можно также объявить непосредственно в оболочке, введя определение модуля с объявлением `defrecord`, но если записи нужны не только для экспериментов в оболочке, их лучше определить во внешнем файле.

Создание и чтение записей

Теперь вы можете создавать переменные с новыми записями, используя функцию с именем записи:

```
iex(1)> require Tower
Tower
iex(2)> tower1 = Tower.tower()
{Tower, "", 20, :earth, ""}
iex(3)> tower2 = Tower.tower(location: "Grand Canyon")
{Tower, "Grand Canyon", 20, :earth, ""}
iex(4)> tower3 = Tower.tower(location: "NYC", height: 241,
... (4)> name: "Woolworth Building")
{Tower, "NYC", 241, :earth, "Woolworth Building"}
iex(5)> tower4 = Tower.tower location: "Rupes Altat 241", height: 500,
... (5)> planemo: :moon, name: "Piccolini View"
{Tower, "Rupes Altat 241", 500, :moon, "Piccolini View"}
```



```
iex(6)> tower5 = Tower.tower planemo: :mars, height: 500,
...(6)> name: "Daga Vallis", location: "Valles Marineris"
{Tower, "Valles Marineris", 500, :mars, "Daga Vallis"}
```

Эти башни (или, точнее, точки, откуда можно сбрасывать предметы) демонстрируют различные способы использования синтаксиса записей для создания переменных, а также применения значений по умолчанию:

- строка 2 создает `tower1` со значениями полей по умолчанию. Фактические значения можно присвоить позже;
- строка 3 создает башню с определенным значением в поле `location` и значениями по умолчанию в остальных полях;
- строка 4 переопределяет значения по умолчанию в полях `location`, `height` и `name`, но оставляет нетронутым поле `planemo`;
- строка 5 изменяет значения по умолчанию во всех полях. Обратите также внимание, что в Elixir обычно не принято помещать аргументы в новые круглые скобки;
- строка 6 изменяет значения по умолчанию во всех полях, а также демонстрирует, что *порядок следования пар имя/значение в списке не играет никакой роли*. Elixir сам расположит их как надо.

Существуют два разных способа чтения элементов записей. Можно использовать «полное квалифицированное имя», с именем записи перед точкой (`.`); такая форма напоминает обращение к полям структур в других языках. Например, вот как можно узнать, на какой планете находится башня `tower5`:

```
iex(7)> Tower.tower(tower5, :planemo)
:mars
```

Или можно пойти более простым путем и импортировать функцию, чтобы избавиться от необходимости применять полное квалифицированное имя:

```
iex(8)> import Tower
nil
iex(9)> tower(tower5, :height)
500
```

Если понадобится изменить значение поля в записи, это можно сделать, как показано в следующем примере. Правая сторона в действительности возвращает совершенно новую запись и связывает ее с именем `tower5`:

```
iex(10)> tower5
{Tower, "Valles Marineris", 500, :mars, "Daga Vallis"}
iex(11)> tower5 = tower(tower5, height: 512)
{Tower, "Valles Marineris", 512, :mars, "Daga Vallis"}
```

Использование записей в функциях

К записям, передаваемым в аргументе, можно применять сопоставление с образцом. Самый простой способ – сопоставление с типом записи, как показано в примере 12.2 (*ch12/ex2-records*).

Пример 12.2 ❖ Метод сопоставления с полной записью

```
defmodule RecordDrop do
  require Planemo
  require Tower

  def fall_velocity(t = Tower.tower()) do
    fall_velocity(Tower.tower(t, :planemo), Tower.tower(t, :height))
  end

  def fall_velocity(:earth, distance) when distance >= 0 do
    :math.sqrt(2 * 9.8 * distance)
  end

  def fall_velocity(:moon, distance) when distance >= 0 do
    :math.sqrt(2 * 1.6 * distance)
  end

  def fall_velocity(:mars, distance) when distance >= 0 do
    :math.sqrt(2 * 3.71 * distance)
  end
end
```

Здесь сопоставление с образцом обнаружит совпадение только с записью `Tower` и поместит ее в переменную `t`. Затем, так же как в предшествующем примере 3.8, передаст отдельные аргументы в вызов `fall_velocity/2` для вычислений, но на этот раз используя синтаксис записей:

```
iex(12)> r(RecordDrop)
warning: redefining module RecordDrop (current version loaded from
  _build/dev/lib/record_drop/ebin/Elixir.RecordDrop.beam)
  lib/record_drop.ex:1

{:reloaded, RecordDrop, [RecordDrop]}
iex(13)> RecordDrop.fall_velocity(tower5)
60.909769331364245
iex(14)> RecordDrop.fall_velocity(tower1)
19.79898987322333
```

Функция `RecordDrop.fall_velocity/1`, представленная в примере 12.3, извлекает значения полей `planemo` и `height` и связывает их с переменными `planemo` и `distance`. Затем она возвращает скорость объекта, сброшенного с этой высоты, в точности как в предыдущих примерах в этой книге.

Путем сопоставления с образцом можно извлекать из записей конкретные поля, как показано в примере 12.3 (*ch12/ex3-records*).

Пример 12.3 ❖ Метод извлечения полей записи сопоставлением с образцом

```
defmodule RecordDrop do
  require Tower
  def fall_velocity(Tower.tower(planemo: planemo, height: distance)) do
    fall_velocity(planemo, distance)
  end

  def fall_velocity(:earth, distance) when distance >= 0 do
    :math.sqrt(2 * 9.8 * distance)
  end

  def fall_velocity(:moon, distance) when distance >= 0 do
    :math.sqrt(2 * 1.6 * distance)
  end

  def fall_velocity(:mars, distance) when distance >= 0 do
    :math.sqrt(2 * 3.71 * distance)
  end
end
```

Можно извлечь отдельные компоненты записи и передать их в функцию, которая вернет скорость приземления объекта, сброшенного с вершины башни.

Наконец, можно сопоставить и запись целиком, и ее отдельные поля одновременно. Пример 12.4 (*ch12/ex4-records*) демонстрирует использование этого смешанного подхода для создания более детального ответа, содержащего не только скорость падения.

Пример 12.4 ❖ Метод одновременного сопоставления записи целиком и отдельных ее компонентов

```
defmodule RecordDrop do
  require Tower
  import Tower
  def fall_velocity(t = tower(planemo: planemo, height: distance)) do
    IO.puts("From #{tower(t, :name)}'s elevation" <>
      "of #{distance} meters on #{planemo},")
    IO.puts("the object will reach #{fall_velocity(planemo, distance)} m/s")
  end
end
```

```

IO.puts("before crashing in #{tower(t, :location)}")
end

def fall_velocity(:earth, distance) when distance >= 0 do
  :math.sqrt(2 * 9.8 * distance)
end

def fall_velocity(:moon, distance) when distance >= 0 do
  :math.sqrt(2 * 1.6 * distance)
end

def fall_velocity(:mars, distance) when distance >= 0 do
  :math.sqrt(2 * 3.71 * distance)
end
end
end

```



Как отмечалось выше, допускается передавать переменные, имена которых совпадают с именами полей (как `planet` в данном случае).

Если передать функции `RecordDrop.fall_velocity/1` запись `Tower`, она извлечет отдельные поля, необходимые для вычислений, и сохранит структуру целиком в переменной `t`, чтобы потом вывести более интересное сообщение, пусть и не совсем грамматически корректное:

```

iex(15)> RecordDrop.fall_velocity(tower5)
From Daga Vallis's elevation of 500 meters on mars,
the object will reach 60.90976933136424520399 m/s
before crashing in Valles Marineris
:ok
iex(16)> RecordDrop.fall_velocity(tower3)
From Woolworth Building's elevation of 241 meters on earth,
the object will reach 68.72845116834803036454 m/s
150 | Chapter 12: Storing Structured Data
before crashing in NYC
:ok

```

Сохранение данных в долговременном хранилище Erlang

Хранилище термов Erlang (Erlang Term Storage, ETS) – простое, но мощное хранилище коллекций в памяти. Оно хранит кортежи, а поскольку записи основаны на кортежах, они также могут храниться в этом хранилище. Хранилище ETS и его дисковая версия DETS обеспечивают простое (возможно, даже слишком) решение для многих задач, связанных с управлением данными. ETS не является полноценной базой данных, но действует очень похоже и может

использоваться как самостоятельное хранилище или как отражение базы данных Mnesia, с которой вы познакомитесь в следующем разделе.

Все записи в таблицах ETS – это кортежи (или соответствующие записи), и один из элементов кортежа играет роль ключа. ETS предлагает на выбор несколько разных структурных решений, в зависимости от особенностей обработки этого ключа. Хранилища ETS способны хранить четыре вида коллекций:

- *множества* (:set) – могут содержать только один элемент данных для каждого конкретного ключа. Этот вид используется по умолчанию;
- *упорядоченные множества* (:ordered_set) – то же, что и множества, но также поддерживает определенный порядок обхода элементов, основанный на ключах. Отлично подходит для всего, что должно храниться в алфавитном или числовом порядке;
- *мультимножества* (:bag) – позволяет сохранять несколько элементов с заданным ключом. Однако если попытаться сохранить в таком хранилище несколько элементов с полностью идентичными значениями, они будут объединены в один элемент;
- *повторяющиеся мультимножества* (:duplicate_bag) – не только позволяет сохранить несколько элементов с заданным ключом, но и дает возможность сохранить несколько элементов с полностью идентичными значениями.

По умолчанию таблицы ETS организованы как множества, но вы можете выбрать любой другой из доступных вариантов при создании таблицы. Далее в примерах мы будем рассматривать только множества, потому что они проще, но все описываемые приемы в равной степени применимы ко всем четырем разновидностям таблиц.



Механизм ETS не требует, чтобы все элементы данных в таблице имели одинаковую структуру. Однако на начальном этапе знакомства намного проще ограничиться одним видом записей или кортежами с одинаковой структурой. Кроме того, в роли ключей можно использовать значения любых видов, включая сложные кортежи и списки, но, опять же, на этапе знакомства старайтесь избегать лишних сложностей.

Все примеры в этом разделе используют тип записи `Planemo`, объявленный в разделе «Записи: структурирование данных до появления структур» выше, и данные, представленные в табл. 12.1.

Таблица 12.1. Планеты для исследования гравитации

Планета	Ускорение свободного падения (м/с ²)	Диаметр (км)	Расстояние до Солнца (10 ⁶ км)
mercury	3,7	4878	57,9
venus	8,9	12 104	108,2
earth	9,8	12 756	149,6
moon	1,6	3475	149,6
mars	3,7	6787	227,9
ceres	0,27	950	413,7
jupiter	23,1	142 796	778,3
saturn	9,0	120 660	1427,0
uranus	8,7	51 118	2871,0
neptune	11,0	30 200	4497,1
pluto	0,6	2300	5913,0
haumea	0,44	1150	6484,0
makemake	0,5	1500	6850,0
eris	0,8	2400	10 210,0

Несмотря на название – *Erlang Term Storage*, это хранилище с успехом можно использовать в программах на Elixir. Подобно тому, как мы использовали модуль `math` языка Erlang для вычисления квадратных корней, например `:math.sqrt(3)`, можно использовать функции ETS, предваряя их префиксом `:ets`.

Создание и заполнение таблицы

Функция `:ets.new/2` создает новую таблицу. В первом аргументе она принимает имя таблицы, а во втором – список параметров настройки. Таких параметров много, очень много, включая идентификаторы типов таблиц, перечисленные в предыдущем разделе, но наиболее важными являются два: `:named_table` и кортеж, начинающийся с `:keypos`.

Все таблицы имеют имена, но не ко всем можно обратиться по имени. Если вы не определили параметр `:named_table`, имя будет существовать только внутри хранилища. Для ссылки на таблицу вам придется использовать значение, возвращаемое функцией `:ets.new/2`. Если вы определили параметр `:named_table`, таблица будет доступна любым процессам по этому имени, и им не придется использовать упомянутое возвращаемое значение.



Даже в случае с именованными таблицами имеется возможность ограничивать права доступа к таблице для процессов посредством параметров `:private`, `:protected` и `:public`.

Еще один важный параметр, особенно для таблиц ETS, хранящих записи, – кортеж `:keypos`. По умолчанию ETS интерпретирует первое значение кортежа как ключ. Кортежи, лежащие в основе записей (и с которыми никогда не следует работать непосредственно), в первом значении всегда хранят идентификатор типа записи, поэтому подход по умолчанию к организации ключей не подходит для записей. Используя кортеж `:keypos`, можно определить, какое значение в записи будет играть роль ключа.

Давайте вспомним, как выглядит определение записи `Planemo`:

```
defmodule Planemo do
  require Record
  Record.defrecord :planemo, [name: :nil, gravity: 0, diameter: 0,
    distance_from_sun: 0]
end
```

Так как эта таблица в основном используется для вычислений на основе `planemo`, имеет смысл использовать в роли ключа поле `:name`. Соответствующее объявление таблицы ETS могло бы выглядеть, как показано ниже:

```
planemo_table = :ets.new(:planemos, [ :named_table, { :keypos,
  Planemo.planemo(:name) + 1 } ])
```

Согласно этому объявлению, таблица получает имя `:planemos` и благодаря использованию параметра `:named_table` становится доступной процессам, которым известно ее имя. Так как по умолчанию используется уровень доступа `:protected`, процесс-создатель может писать в таблицу, а другие – только читать из нее. Это объявление также сообщает механизму ETS, что в качестве ключа должно использоваться поле `:name`.



ETS предполагает получить числовое значение в параметре `:keypos`, определяющее индекс ключевого поля в кортеже, лежащем в основе записи, при условии что нумерация полей начинается с единицы. Вызов функции `planemo` возвращает индекс поля, нумеруя их с нуля. Именно поэтому в предыдущем коде к результату функции прибавляется единица.

Так как не указано иное, таблица интерпретируется как `:set`, где каждому ключу соответствует только один экземпляр записи и ETS не поддерживает сортировку по ключу.

После создания таблицы, как показано в примере 12.5 (*ch12/ex5-ets*), с помощью функции `:ets.info/1` можно узнать некоторую информацию о ней.

Пример 12.5 ❖ Создание простой таблицы ETS и получение информации о ней

```
defmodule PlanemoStorage do
  require Planemo

  def setup do
    planemo_table = :ets.new(:planemos, [:named_table,
      {:keypos, Planemo.planemo(:name) + 1}])
    :ets.info planemo_table
  end
end
```

Если скомпилировать и выполнить этот код, вы получите информацию о пустой таблице ETS со множеством свойств, знание которых может пригодиться в будущем:

```
$ iex -S mix
Erlang/OTP 19 [erts-8.0] [source] [64-bit] [smp:4:4] [async-threads:10]
  [hipe] [kernel-poll:false]

Compiling 2 files (.ex)
Generated planemo_storage app
Interactive Elixir (1.3.1) - press Ctrl+C to exit (type h()) ENTER for help
iex(1)> PlanemoStorage.setup
[read_concurrency: false, write_concurrency: false, compressed: false,
memory: 299, owner: #PID<0.57.0>, heir: :none, name: :planemos, size: 0,
node: :nonode@nohost, named_table: true, type: :set, keypos: 2,
protection: :protected]
```

Большая часть этой информации не представляет особого интереса или не вызывает удивления. Наиболее примечательными сведениями в данный момент являются: имя (`:planemos`), размер (0 – пустая таблица!) и индекс ключевого поля (`keypos` имеет не значение 1 по умолчанию, а 2 – индекс поля с именем в кортеже, который лежит в основе записи). Кроме того, для параметров `type` и `protection` установлены значения по умолчанию: `:protected` и `:set`.

Можно создать только одну таблицу ETS с конкретным именем. Повторный вызов функции `PlanemoStorage.setup/0` приведет к ошибке:

```
iex(2)> PlanemoStorage.setup
** (ArgumentError) argument error
  (stdlib) :ets.new(:planemos, [:named_table, {:keypos, 2}])
  planemo_storage.ex:5: PlanemoStorage.setup/0
```

Чтобы избежать этого, по крайней мере на этапе начальных экспериментов, используйте команду `:ets.delete/1` для удаления таблиц, передавая ей имя таблицы (в данном случае `:planemos`) в виде аргу-

мента. Если вы предполагаете многократно вызывать код инициализации освоения основ, можете также добавить в него вызов `:ets.info/1` и проверять результат, сравнивая его с `:undefined`, чтобы убедиться в отсутствии таблицы, или заключить вызов `:ets.new/2` в конструкцию `try...catch`.

Конечно, намного интереснее экспериментировать с таблицей ETS, когда она наполнена данными. Поэтому перейдем к следующему шагу и используем функцию `:ets.insert/2` для наполнения таблицы. В первом аргументе она принимает ссылку на таблицу либо в виде ее имени (если при создании таблицы был установлен параметр `named_table`), либо в виде значения, полученного в результате вызова `:ets.new/2`. В примере 12.6 (*ch12/ex6-ets*) в первом вызове мы использовали имя, чтобы показать, что такой прием работает, но в остальной части примера используется переменная со значением, которое вернула функция `:ets.new/2`. Второй аргумент – запись, представляющая одну строку из табл. 12.1.

Пример 12.6 ❖ Заполнение простой таблицы ETS и вывод информации о ней

```
defmodule PlanemoStorage do
  require Planemo

  def setup do
    planemos_table = :ets.new(:planemos, [:named_table,
      {keypos, Planemo.planemo(:name) + 1}])
    :ets.insert :planemos, Planemo.planemo(name: :mercury, gravity: 3.7,
      diameter: 4878, distance_from_sun: 57.9)
    :ets.insert :planemos, Planemo.planemo(name: :venus, gravity: 8.9,
      diameter: 12104, distance_from_sun: 108.2)
    :ets.insert :planemos, Planemo.planemo(name: :earth, gravity: 9.8,
      diameter: 12756, distance_from_sun: 149.6)
    :ets.insert :planemos, Planemo.planemo(name: :moon, gravity: 1.6,
      diameter: 3475, distance_from_sun: 149.6)
    :ets.insert :planemos, Planemo.planemo(name: :mars, gravity: 3.7,
      diameter: 6787, distance_from_sun: 227.9)
    :ets.insert :planemos, Planemo.planemo(name: :ceres, gravity: 0.27,
      diameter: 950, distance_from_sun: 413.7)
    :ets.insert :planemos, Planemo.planemo(name: :jupiter, gravity: 23.1,
      diameter: 142796, distance_from_sun: 778.3)
    :ets.insert :planemos, Planemo.planemo(name: :saturn, gravity: 9.0,
      diameter: 120660, distance_from_sun: 1427.0)
    :ets.insert :planemos, Planemo.planemo(name: :uranus, gravity: 8.7,
      diameter: 51118, distance_from_sun: 2871.0)
    :ets.insert :planemos, Planemo.planemo(name: :neptune, gravity: 11.0,
```

```

    diameter: 30200, distance_from_sun: 4497.1)
:ets.insert :planemos, Planemo.planemo(name: :pluto, gravity: 0.6,
    diameter: 2300, distance_from_sun: 5913.0)
:ets.insert :planemos, Planemo.planemo(name: :haumea, gravity: 0.44,
    diameter: 1150, distance_from_sun: 6484.0)
:ets.insert :planemos, Planemo.planemo(name: :makemake, gravity: 0.5,
    diameter: 1500, distance_from_sun: 6850.0)
:ets.insert :planemos, Planemo.planemo(name: :eris, gravity: 0.8,
    diameter: 2400, distance_from_sun: 10210.0)

:ets.info planemo_table
end
end

```

Последний вызов `:ets.info/1` теперь сообщает, что в таблице имеются 14 элементов:

```

iex(2)> r(PlanemoStorage)
warning: redefining module PlanemoStorage (current version loaded from
  Elixir.PlanemoStorage.beam)
  lib/planemo_storage.ex:1

{:reloaded, PlanemoStorage, [PlanemoStorage]}
iex(3)> :ets.delete(:planemos)
true
iex(4)> PlanemoStorage.setup
[read_concurrency: false, write_concurrency: false, compressed: false,
memory: 495, owner: #PID<0.57.0>, heir: :none, name: :planemos, size: 14,
node: :nonode@nohost, named_table: true, type: :set, keypos: 2,
protection: :protected]

```

Чтобы увидеть содержимое таблицы в интерактивной оболочке, можно воспользоваться функцией `:ets.tab2list/1`, которая возвращает список записей, разбитый на отдельные строки для простоты чтения:

```

iex(5)> :ets.tab2list :planemos
[{:planemo, :neptune, 11.0, 30200, 4497.1},
{:planemo, :jupiter, 23.1, 142796, 778.3},
{:planemo, :haumea, 0.44, 1150, 6484.0}, {:planemo, :pluto, 0.6, 2300, 5913.0},
{:planemo, :mercury, 3.7, 4878, 57.9}, {:planemo, :earth, 9.8, 12756, 149.6},
{:planemo, :makemake, 0.5, 1500, 6850.0}, {:planemo, :moon, 1.6, 3475, 149.6},
{:planemo, :mars, 3.7, 6787, 227.9}, {:planemo, :saturn, 9.0, 120660, 1427.0},
{:planemo, :uranus, 8.7, 51118, 2871.0}, {:planemo, :ceres, 0.27, 950, 413.7},
{:planemo, :venus, 8.9, 12104, 108.2}, {:planemo, :eris, 0.8, 2400, 10210.0}]

```

Если воспользоваться инструментом Observer, ту же информацию можно увидеть в еще более удобочитаемой форме. Для этого запус-

тите программу из оболочки командой `:observer.start()` и щелкните на вкладке **Table Viewer** (Обозреватель таблиц). Вы увидите список таблиц, как показано на рис. 12.1. Дважды щелкните на таблице `planemos`, и вы увидите более подробное описание ее содержимого, как показано на рис. 12.2.

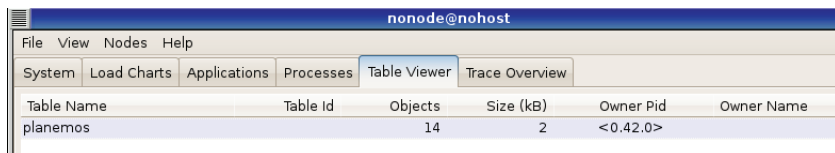


Рис. 12.1 ❖ Обзорщик таблиц

1	2	3	4	5
planemo	ceres	0.270	950	4.14e+2
planemo	earth	9.80	12756	1.50e+2
planemo	eris	0.800	2400	1.02e+4
planemo	haumea	0.440	1150	6.48e+3
planemo	jupiter	23.1	142796	7.78e+2
planemo	makemake	0.500	1500	6.85e+3
planemo	mars	3.70	6787	2.28e+2
planemo	mercury	39	4878	57.9
planemo	moon	1.60	3475	1.50e+2
planemo	neptune	11.0	30200	4.50e+3
planemo	pluto	0.600	2300	5.91e+3
planemo	saturn	9.00	120660	1.43e+3
planemo	uranus	8.70	51118	2.87e+3
planemo	venus	8.90	12104	1.08e+2

Objects: 14

Рис. 12.2 ❖ Обзор таблицы `planemos` в обозревателе

Обозреватель ничего не знает о ваших объявлениях записей; он знает только количество полей. Меню **Edit** (Правка) позволяет опросить таблицу, чтобы убедиться, что в окне обозревателя отображается самое последнее ее содержимое, и настроить интервал опроса, если вам больше нравится, чтобы обновление выполнялось автоматически. Если таблица объявлена общедоступной, вы сможете даже править ее содержимое в обозревателе.



Если вы пожелаете получить список всех таблиц ETS, имеющихся в данный момент, попробуйте вызвать `:ets.i()` в оболочке. Вы увидите свои таблицы (если создавали их) ближе к концу списка.

Простые запросы

Самый простой способ увидеть содержимое записей в вашей таблице ETS – воспользоваться функцией `:ets.lookup/2`. Работу этой функции легко проверить в интерактивной оболочке:

```
iex(6)> :ets.lookup(:planemos, :eris)
[{:planemo, :eris, 0.8, 2400, 10210.0}]
```

Она всегда возвращает список. Это верно даже для таблиц ETS с типом `:set`, то есть когда с ключом может быть связана только одна запись, и даже если запись состоит из единственного значения. В ситуациях, подобных этой, когда заранее известно, что будет возвращено единственное значение, воспользуйтесь функцией `hd/1`, чтобы быстро извлечь голову списка. Так как может быть только один элемент, он как раз окажется головой списка:

```
iex(7)> hd(:ets.lookup(:planemos, :eris))
{:planemo, :eris, 0.8, 2400, 10210.0}
```

Квадратные скобки исчезли, а это означает, что теперь без труда можно извлечь, например, ускорение свободного падения для планеты:

```
iex(8)> result = hd(:ets.lookup(:planemos, :eris))
{:planemo, :eris, 0.8, 2400, 10210.0}
iex(9)> require Planemo
Planemo
iex(10)> Planemo.planemo(result, :gravity)
0.8
```

Изменение значений

Несмотря на возможность повторного присваивания значений переменным в языке Elixir, лучше воздержаться от изменения значений переменных или элементов списка. Следование стратегии «однократного присваивания значений» способствует созданию более надежных программ, использующих прием взаимодействий нескольких процессов (как было показано в главе 9 и как будет показано в главе 13). Однако хранилище ETS предназначено для хранения значений, которые может потребоваться изменить. Если, к примеру, вам понадобится изменить значение ускорения свободного падения на Меркурии (`:mercury`), вы сможете это сделать:

```
iex(11)> :ets.insert(:planemos, Planemo.planemo(name: :mercury,
...(11)> gravity: 3.9, diameter: 4878, distance_from_sun: 57.9))
```

```
true  
iex(12)> :ets.lookup(:planemos, :mercury)  
[{:planemo, :mercury, 3.9, 4878, 57.9}]
```

Однако допустимость изменения значений в таблице ETS не означает, что вы должны переписать весь свой код и заменить все переменные таблицами ETS. Кроме того, не следует стремиться сделать все свои таблицы общедоступными, чтобы различные процессы могли читать и изменять любые значения в них (превратив тем самым таблицы в разновидность общей памяти).

Подумайте, когда изменение данных может оказаться полезным и когда такие изменения могут способствовать появлению трудноуловимых ошибок. Очевидно, что данной программе не нужна возможность изменения ускорения свободного падения на Меркурии, но другой программе, реализующей интернет-магазин, определенно может понадобиться изменить адрес доставки. Если у вас возникают сомнения, действуйте очень осторожно.

Таблицы ETS и процессы

Теперь, имея возможность извлекать ускорения свободного падения на разных планетах, можно добавить в модуль `Drop` возможность вычисления скорости падения в самых разных местах. Пример 12.7 (*ch12/ex7-ets-calculator*) объединяет модуль `Drop` из примера 9.6 с таблицей ETS из примера 12.6 и реализует более мощный калькулятор.

Пример 12.7 ❖ Вычисление скорости падения с использованием таблицы ETS, хранящей сведения о планетах

```
defmodule Drop do  
  require Planemo  
  def drop do  
    setup  
    handle_drops  
  end  
  
  def handle_drops do  
    receive do  
      {from, planemo, distance} ->  
        send(from, {planemo, distance, fall_velocity(planemo, distance)})  
        handle_drops  
    end  
  end  
  
  def fall_velocity(planemo, distance) when distance >= 0 do  
    p = hd(:ets.lookup(:planemos, planemo))
```

```

:math.sqrt(2 * Planemo.planemo(p, :gravity) * distance)
end

def setup do
  :ets.new(:planemos, [:named_table,
    {:keypos, Planemo.planemo(:name) + 1}])
  info = [
    {: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}]
  insert_into_table(info)
end

def insert_into_table([]) do # остановить рекурсию
  :undefined
end

def insert_into_table([{:name, gravity, diameter, distance} | tail]) do
  :ets.insert(:planemos, Planemo.new(name: name, gravity: gravity,
    diameter: diameter, distance_from_sun: distance))
  insert_into_table(tail)
end
end

```

Функция `drop/0` немного изменилась, теперь инициализация выполняется в отдельной функции, чтобы избежать создания таблицы в каждом вызове. Обработка сообщений также перенесена в отдельную функцию, `handle_drops/0`. Функция `fall_velocity/2` тоже изменилась, теперь она отыскивает планеты в таблице ETS и получает ускорение свободного падения оттуда, а не из констант, определяемых в теле функции. (Можно было бы передавать переменную `planemo_table` из предыдущего примера рекурсивному обработчику сообщений в виде аргумента, но намного проще использовать ее как именованную таблицу.)

Функция `setup` также претерпела существенные изменения. Вместо нескольких вызовов `:ets.insert` она сначала создает список кор-

тежей с информацией о планетах, а затем вызывает функцию `insert_into_table/1`, которая рекурсивно вставляет все записи.



Если этот процесс завершится аварийно и его потребуется перезапустить, в момент запуска он вновь вызовет функцию `setup/0`, которая в настоящий момент не проверяет существование таблицы ETS. Это может вызвать ошибку, если таблицы ETS не удаляются по завершении процесса, создавшего их. Механизм ETS поддерживает параметр `heir` и функцию `:ets.give_away/3` на случай, если вам потребуется избежать подобной проблемы, но пока нам этого не требуется.

Если объединить этот модуль с модулем `MphDrop` из примера 9.7, вы сможете вычислить скорость падения на всех этих планетах. Поскольку сообщения передаются асинхронно, кортеж может появиться в середине вывода:

```
iex(1)> c("drop.ex")
[Drop]
iex(2)> c("mph_drop.ex")
[MphDrop]
iex(3)> pid1 = spawn(MphDrop, :mph_drop, [])
#PID<0.47.0>
iex(4)> send(pid1, {:earth, 20})
On earth, a fall of 20 meters yields a velocity of 44.289078952755766 mph.
{:earth,20}
iex(5)> send(pid1, {:eris, 20})
On eris, a fall of 20 meters yields a velocity of 12.65402255793022 mph.
{:eris,20}
iex(6)> send(pid1, {:makemake, 20})
On makemake, a fall of 20 meters yields a velocity of 10.003883211552367 mph.
{:makemake,20}
```

Теперь мы имеем намного более богатый выбор планет, чем прежде, когда имелись только `:earth`, `:moon` и `:mars`!

Следующие шаги

Многим приложениям необходимо лишь быстродействующее хранилище пар ключ/значение, тогда как таблицы ETS обладают намного более широкими возможностями, чем было продемонстрировано в примерах выше. Имеется возможность использовать спецификации сопоставления на языке Erlang и функцию `:ets.fun2ms` для создания более сложных запросов с применением `:ets.match` и `:ets.select`. А с помощью `:ets.delete` можно удалять записи и даже целые таблицы.

Функции `:ets.first`, `:ets.next` и `:ets.last` позволяют реализовать рекурсивный обход таблиц.

Но самой важной, пожалуй, является поддержка механизма DETS (Disk-Based Erlang Term Storage – дисковое хранилище термов Erlang), которое обладает аналогичными возможностями, но хранит таблицы на диске. Этот механизм действует медленнее, ограничивает размеры таблиц 2 гигабайтами, но данные не исчезают при остановке управляющего процесса.

Вы можете самостоятельно продолжить исследование ETS и DETS, но если вам требуется что-то более мощное и надежное, поддерживающее возможность распределенного хранения данных на нескольких узлах, обратите внимание на базу данных Mnesia.

Хранение записей в Mnesia

Mnesia – это система управления базами данных (Database Management System, DBMS), распространяемая вместе с Erlang, которую с успехом можно использовать в программах на Elixir. Она основана на механизмах ETS и DETS, но предлагает больше возможностей, чем составляющие ее компоненты.

Вам обязательно следует подумать о возможности миграции с таблиц ETS (и DETS) на базу данных Mnesia, если:

- требуется обеспечить хранение и доступность данных на нескольких узлах;
- не важно, где хранятся данные, в памяти или на диске;
- требуется поддержка транзакций, чтобы иметь возможность откатить операцию, если что-то пошло не так;
- желательно иметь более удобный синтаксис поиска данных и соединения таблиц;
- руководству больше нравится, как звучат слова «база данных», чем «таблицы».

Вы можете даже обнаружить, что для одних разделов проекта лучше подходят таблицы ETS, а для других – база данных Mnesia.



В переводе с греческого «амнезия» (ἀμνησία) означает «потеря памяти», а «мнезия» – «помнить», «память».

Настройка базы данных Mnesia

Если у вас есть желание сохранить данные на диске, вы должны выполнить настройку базы данных Mnesia. Прежде чем включить Mnesia в работу, нужно создать базу данных вызовом функции `mnesia.create_schema/1`. В данный момент мы используем только один узел, локальный, поэтому вызов имеет следующий вид:


```
iex(1)> :mnesia.create_schema([node()])
:ok
```

По умолчанию, когда вызывается функция `:mnesia.create_schema/1`, Mnesia сохраняет схему базы данных на диске, в текущем рабочем каталоге. Если после вызова функции заглянуть в текущий рабочий каталог, вы увидите новую папку с именем *Mnesia.nonode@nohost*. Изначально в ней хранится единственный файл *FALLBACK.BUP*. Функция `node/0` просто вернет идентификатор узла, на котором вы сейчас находитесь, чего вполне достаточно для начала.



Если запустить Mnesia без предварительного вызова `:mnesia.create_schema/1`, Mnesia сохранит схему в памяти и уничтожит ее в момент остановки.

В отличие от механизмов ETS и DETS, которые доступны всегда, базу данных Mnesia требуется запустить:

```
iex(2)> :mnesia.start()
:ok
```

Этот вызов создаст файл *schema.DAT* в каталоге *Mnesia.nonode@nohost*. Имеется также парная ей функция `:mnesia.stop/0`, которую можно использовать для остановки базы данных.



Если компьютер, на котором выполняется база данных Mnesia, перейдет в режим «сна», после восстановления нормального режима работы вы можете получить малопонятное сообщение, например: `«Mnesia(nonode@nohost): ** WARNING ** Mnesia is overloaded: {dump_log, time_threshold}»` («Mnesia(nonode@nohost): ** ВНИМАНИЕ ** Mnesia перегружена: {dump_log, time_threshold}»). Не волнуйтесь, это всего лишь побочный эффект смены состояния компьютера — ваши данные находятся в полной безопасности. Разумеется, система, находящаяся в промышленной эксплуатации, не должна работать на устройствах, которые могут переходить в режим «сна».

Создание таблиц

Подобно ETS, основой базы данных Mnesia являются таблицы — коллекции записей. Она также предлагает типы таблиц `:set`, `:ordered_set` и `:bag`, но не поддерживает таблиц `:duplicate_bag`.

База данных Mnesia требует больше информации о ваших данных, чем ETS. Механизм ETS просто принимает кортежи с данными любого вида, и ему необходимо только знать, какое поле будет играть роль ключа. Базе данных Mnesia требуется знать больше о том, что вы собираетесь хранить, в том числе и список имен полей. Эта проблема

проще всего решается путем определения записей и согласованного использования имен полей. Предусмотрен даже простой способ передачи имен записей в Mnesia в виде функции `record_info/2`.

Таблицу с описанием планет легко можно перенести в Mnesia и пользоваться ею так же, как с применением механизма ETS, а некоторые операции с таблицей будут выглядеть даже проще. Пример 12.8 (*ch12/ex8-mnesia*) демонстрирует, как создать таблицу в базе данных Mnesia. Метод `setup/0` создает схему и запускает базу данных Mnesia, а затем создает в ней таблицу, состоящую из записей типа `Planemo`. После создания таблицы в нее записываются значения из табл. 12.1.

Пример 12.8 ❖ Создание таблицы в базе данных Mnesia с информацией о планетах

```
defmodule Drop do
  require Planemo

  def drop do
    setup
    handle_drops
  end

  def handle_drops do
    receive do
      {from, planemo, distance} ->
        send(from, {planemo, distance, fall_velocity(planemo, distance)})
        handle_drops
    end
  end

  def fall_velocity(planemo, distance) when distance >= 0 do
    {:atomic, [p | _]} = :mnesia.transaction(fn() ->
      :mnesia.read(PlanemoTable, planemo) end)
    :math.sqrt(2 * Planemo.planemo(p, :gravity) * distance)
  end

  def setup do
    :mnesia.create_schema([node()])
    :mnesia.start()
    :mnesia.create_table(PlanemoTable, [{:attributes,
      [:name, :gravity, :diameter, :distance_from_sun]},
      {:record_name, :planemo}])

    f = fn ->
      :mnesia.write(PlanemoTable, Planemo.planemo(name: :mercury, gravity: 3.7,
        diameter: 4878, distance_from_sun: 57.9), :write)
      :mnesia.write(PlanemoTable, Planemo.planemo(name: :venus, gravity: 8.9,
        diameter: 12104, distance_from_sun: 108.2), :write)
    end
```

```

:mnesia.write(PlanemoTable, Planemo.planemo(name: :earth, gravity: 9.8,
diameter: 12756, distance_from_sun: 149.6), :write)
:mnesia.write(PlanemoTable, Planemo.planemo(name: :moon, gravity: 1.6,
diameter: 3475, distance_from_sun: 149.6), :write)
:mnesia.write(PlanemoTable, Planemo.planemo(name: :mars, gravity: 3.7,
diameter: 6787, distance_from_sun: 227.9), :write)
:mnesia.write(PlanemoTable, Planemo.planemo(name: :ceres, gravity: 0.27,
diameter: 950, distance_from_sun: 413.7), :write)
:mnesia.write(PlanemoTable, Planemo.planemo(name: :jupiter, gravity: 23.1,
diameter: 142796, distance_from_sun: 778.3), :write)
:mnesia.write(PlanemoTable, Planemo.planemo(name: :saturn, gravity: 9.0,
diameter: 120660, distance_from_sun: 1427.0), :write)
:mnesia.write(PlanemoTable, Planemo.planemo(name: :uranus, gravity: 8.7,
diameter: 51118, distance_from_sun: 2871.0), :write)
:mnesia.write(PlanemoTable, Planemo.planemo(name: :neptune, gravity: 11.0,
diameter: 30200, distance_from_sun: 4497.1), :write)
:mnesia.write(PlanemoTable, Planemo.planemo(name: :pluto, gravity: 0.6,
diameter: 2300, distance_from_sun: 5913.0), :write)
:mnesia.write(PlanemoTable, Planemo.planemo(name: :haumea, gravity: 0.44,
diameter: 1150, distance_from_sun: 6484.0), :write)
:mnesia.write(PlanemoTable, Planemo.planemo(name: :makemake, gravity: 0.5,
diameter: 1500, distance_from_sun: 6850.0), :write)
:mnesia.write(PlanemoTable, Planemo.planemo(name: :eris, gravity: 0.8,
diameter: 2400, distance_from_sun: 10210.0), :write)
end
:mnesia.transaction(f)
end
end

```

В вызов `:mnesia.create_table` явно передаются атрибуты таблицы, потому что на настоящий момент не существует простого способа извлечения имен полей из записей. Обычно Mnesia предполагает, что имя таблицы совпадает с именем первого поля в записи, но в данном случае таблице дается имя `PlanemoTable`, а первое поле в записи имеет имя `:planemo`. Имя записи определяется явно, следующим кодом:

```
{:record_name, :planemo}.
```

Функция `:mnesia.write` принимает три параметра: имя таблицы, запись и тип используемой блокировки (в данном случае `:write`).

Кроме первоначальной настройки, все операции записи выполняются в функции `fn`, которая передается в вызов `:mnesia.transaction` для выполнения в рамках транзакции. Mnesia автоматически перезапустит транзакцию при наличии какой-то активности, блокирующей ее выполнение, поэтому код может выполняться многократно,

пока транзакция не преуспеет. По этой причине не следует выполнять в транзакциях никаких операций с побочными эффектами или перехватывать и обрабатывать исключения. Если ваша функция, переданная в `:mnesia.transaction`, вызовет `:mnesia.abort/1` (например, из-за несоблюдения некоторых условий), произойдет откат транзакции, а в вызывающий код будет возвращен кортеж, начинающийся с `:aborted` вместо `:atomic`.



Если вам потребуется смешивать разные задачи в транзакциях, обратите внимание на более гибкую функцию `:mnesia.activity/2`.

Все операции в базе данных Mnesia должны выполняться в рамках транзакций, особенно если база данных хранится на нескольких узлах. Основные методы – `:mnesia.write`, `:mnesia.read` и `:mnesia.delete` – работают только внутри транзакций, точка. Имеются также методы с префиксом `dirty_`, но всякий раз, когда вы используете их, особенно для записи информации в базу данных, вы сильно рискуете.



Так же как при использовании таблиц ETS, вы можете изменять значения полей в записях, используя соответствующий ключ.

Если вам любопытно увидеть результаты работы этой функции, попробуйте вызвать функцию `:mnesia.table_info`, которая сообщит исчерпывающую информацию, как показано в следующем листинге, где приводятся наиболее важные сведения:

```
iex(1)> c("drop.ex")
[Drop]
iex(2)> Drop.setup
{:atomic, :ok}
iex(3)> :mnesia.table_info(PlanemoTable, :all)
[access_mode: :read_write,
 active_replicas: [:"nonode@nohost"],
 all_nodes: [:"nonode@nohost"],
 arity: 5,
 attributes: [:name, :gravity, :diameter, :distance_from_sun],
 ...
 ram_copies: [:"nonode@nohost"],
 record_name: :planemo,
 record_validation: {:planemo, 5, :set},
 type: :set,
 size: 14,
 ...]
```

Здесь можно видеть, какие узлы вовлечены в хранение таблицы (`nonode@nohost` – это текущий узел). Параметр `arity` в данном случае

отражает количество полей в записи, а параметр `attributes` сообщает их имена. Имя текущего узла в параметре `ram_copies` говорит о том, что эта таблица находится в локальной памяти. Здесь также видно, что таблица имеет тип `:set` и хранит 14 записей.



По умолчанию база данных Mnesia хранит таблицы в ОЗУ (`ram_copies`) только на текущем узле. Это увеличивает скорость доступа к данным, но также означает, что они уничтожаются в случае аварийного сбоя узла. Если определить параметр `disc_copies`, Mnesia будет поддерживать копию данных на диске, но все еще использовать ОЗУ для ускорения работы. Можно также определить параметр `disc_only_copies`, но это сильно ухудшит производительность. В отличие от ETS, таблица в базе данных Mnesia сохранится, даже если процесс, создавший ее, завершится аварийно, и почти наверняка переживет крах узла, при условии, что она будет храниться не только в ОЗУ на единственном узле. Комбинируя эти параметры и используя несколько узлов, вы сможете конструировать быстрые и надежные системы.

Теперь таблица готова к использованию. Содержимое базы данных Mnesia, так же как таблиц ETS, можно просматривать с помощью Observer. Запустите этот инструмент и перейдите на вкладку **Table Viewer** (Обозреватель таблиц), а затем в меню **View** (Вид) выберите пункт **Mnesia Tables** (Таблицы Mnesia). Интерфейс обзора таблиц Mnesia напоминает интерфейс обзора таблиц ETS.

Чтение данных

По аналогии с операциями записи, вызовы `:mnesia.read` следует заворачивать в определение функции `fn` и передавать ее в вызов `:mnesia.transaction`. Для знакомства это можно проделать в интерактивной оболочке:

```

iex(4)> :mnesia.transaction(fn()->:mnesia.read(PlanemoTable, :neptune) end)
{:atomic, [{:planemo, :neptune, 11.0, 30200, 4497.1}]}
```

Результат возвращается в виде кортежа, который, в случае успеха, содержит в первом элементе `:atomic`, а во втором — список с данными из таблицы. Данные упакованы в записи, и вы легко сможете извлекать их по именам полей.

Функцию `fall_velocity/2` из примера 12.8 легко переписать так, чтобы она использовала базу данных Mnesia вместо таблицы ETS. Версия на основе таблицы ETS имеет следующую реализацию:

```

def fall_velocity(planemo, distance) when distance >= 0 do
  p = hd(:ets.lookup(:planemos, planemo))
```

```
:math.sqrt(2 * Planemo.planemo(p, :gravity) * distance)
end
```

Реализация для Mnesia отличается только одной второй строкой:

```
def fall_velocity(planemo, distance) when distance >= 0 do
  {:atomic, [p | _]} = :mnesia.transaction(fn() ->
    :mnesia.read(PlanemoTable, planemo) end)
  :math.sqrt(2 * Planemo.planemo(p, :gravity) * distance)
end
```

Так как Mnesia возвращает кортеж, а не список, потребовалось использовать сопоставление с образцом для извлечения первого элемента списка, содержащегося во втором элементе кортежа (и отбросить хвост списка путем присваивания переменной `_`). Эта таблица имеет тип множества, поэтому всегда будет хранить единственный элемент, то есть точно так же можно было бы использовать образец `{:atomic, [p]}`. Данные в переменной `p` можно использовать для вычислений, как и прежде.

Этот код возвращает те же результаты, что и пример 12.8:

```
iex(5)> r(Drop)
warning: redefining module Drop (current version loaded from
  _build/dev/lib/mph_drop/ebin/Elixir.Drop.beam)
lib/drop.ex:1

{:reloaded, Drop, [Drop]}
iex(6)> Drop.fall_velocity(:earth, 20)
19.79898987322333
iex(7)> pid1 = spawn(MphDrop, :mph_drop, [])
#PID<0.115.0>
iex(8)> send(pid1, {:earth, 20})
On earth, a fall of 20 meters yields a velocity of 44.289078952755766 mph.
{:earth, 20}
```

Для целей учебного проекта простой функции `:mnesia.read` вполне достаточно. Вы можете также потребовать от Mnesia построить индексы по другим полям, отличным от ключа, и выполнять запросы с помощью `:mnesia.index_read`.



Для удаления записей можно использовать функцию `:mnesia.delete/2`, которую также следует вызывать внутри транзакции.

Глава 13

ОСНОВЫ ОТП

На данный момент вы обладаете всеми знаниями, необходимыми для создания проектов на Elixir, ориентированных на процессы. Вы знаете, как создавать функции, в том числе и рекурсивные, знаете, какие структуры данных поддерживаются в языке Elixir, и, что особенно важно, знаете, как создавать и управлять процессами. Что еще может потребоваться?

Программирование на уровне процессов – отличная методология, но, как известно, дьявол скрывается в деталях. Основные инструменты Elixir обладают широкими возможностями, но также могут заводить в лабиринт отладки состояний гонки, что случается время от времени. Смешивание разных стилей программирования может вызывать поведение, не соответствующее ожиданиям, и код, прекрасно работающий в одном окружении, может тяжело интегрироваться в другое.

В лаборатории Ericsson, где создавался язык Erlang (не забывайте, что код на Elixir выполняется в виртуальной машине Erlang), был также создан ряд библиотек. Одна из них, ОТП (Open Telecom Platform – открытая телекоммуникационная платформа), была разработана для поддержки крупномасштабных проектов на Elixir и Erlang. Эта библиотека распространяется в составе Erlang, и хотя она не является частью языка, она определенно является неотъемлемой частью культуры Erlang. Границы, где заканчиваются Elixir и Erlang и начинается ОТП, не всегда очевидны, но точками входа определенно являются модули, реализующие поведение. Процессы объединяются с модулями поведения в ОТП-приложения, которые управляются супервизором.

Процессы, которые мы видели до сих пор, имели очень простой жизненный цикл. При необходимости они настраивали дополни-

тельные ресурсы и запускали другие процессы. После запуска они переходили в цикл ожидания и обработки сообщений и завершались в случае сбоя. Некоторые из них при необходимости могли восстанавливать свою работоспособность после сбоя.

OTP формализует все эти операции и некоторые виды поведения. Из числа наиболее типичных поведения можно назвать `GenServer` (generic server – обобщенный сервер) и `Supervisor`. В программах на Erlang можно использовать поведения `gen_fsm` (конечный автомат, или машина с конечным числом состояний) и `gen_event`. Инструмент сборки Mix в Elixir позволяет создавать приложения, упаковывая OTP-код в единую выполняемую (и обновляемую) систему.

Поведения предопределяют механизмы, используемые для создания процессов и взаимодействий с ними, а компилятор предупредит вас, если вы что-то пропустите. Вам придется определить обратные вызовы, обрабатывающие конкретные виды событий, а также принять решение о структуре приложения.



Желающие могут посмотреть видеоролик Стива Виноски (Steve Vinoski) с введением в OTP: «Erlang's Open Telecom Platform (OTP) Framework» (<https://www.infoq.com/presentations/Erlang-OTP-Behaviors>). Первые полчаса покажутся вам знакомыми, но в целом обзор превосходен. Если вам интересно узнать, зачем изучать OTP и разработку на основе процессов в целом, посмотрите презентацию Франческо Чезарини (Francesco Cesarini), которая понятна даже без пояснений (<https://www.erlang-factory.com/upload/presentations/719/francesco-otp.pdf>).

Создание служб с помощью GenServer

Программист, разрабатывая ядро программы, обычно думает о вычислениях, хранении информации и подготовке ответов. Этот подход прекрасно укладывается в идею поведения `GenServer`. Этот модуль предоставляет комплект методов, позволяющих запускать процессы, отвечать на запросы, корректно завершать процессы и даже передавать состояние в новые процессы, если это потребуется.

В табл. 13.1 перечислены функции, которые потребуются вам для реализации службы на основе `GenServer`. Для простых служб самыми важными являются первые две-три функции, а для остальных можете использовать обычные заглушки.

Таблица 13.1. Какие функции вызываются модулем *GenServer*

Функция	Вызывается из	Что делает
init/1	GenServer.start_link	Запускает процесс
handle_call/3	GenServer.call	Обрабатывает синхронные вызовы
handle_cast/2	GenServer.cast	Обрабатывает асинхронные вызовы
handle_info/2	Случайные сообщения	Обрабатывает сообщения, не относящиеся к OTP
terminate/2	Отказ или сигнал завершения от супервизора	Останавливает процесс
code_change/3	Системные библиотеки для обновления кода	Позволяет переключиться на новую версию кода без потери состояния

Пример 13.1 (*ch13/ex1-drop*) демонстрирует, с чего начать. Он смешивает простые вычисления из примера 2.1 со счетчиком из примера 9.4.

Пример 13.1 ❖ Простой пример использования `gen_server`

```
defmodule DropServer do
  use GenServer

  defmodule State do
    defstruct count: 0
  end

  # Вспомогательный метод для запуска
  def start_link do
    GenServer.start_link(__MODULE__, [], [{:name, __MODULE__}])
  end

  # Далее следуют функции обратного вызова,
  # которые используются модулем GenServer
  def init([]) do
    {:ok, %State{}}
  end

  def handle_call(request, _from, state) do
    distance = request
    reply = {:ok, fall_velocity(distance)}
    new_state = %State{count: state.count + 1}
    {:reply, reply, new_state}
  end

  def handle_cast(_msg, state) do
    IO.puts("So far, calculated #{state.count} velocities.")
    {:noreply, state}
  end

  def handle_info(_info, state) do
```

```

{:noreply, state}
end

def terminate(_reason, _state) do
  {:ok}
end

def code_change(_old_version, state, _extra) do
  {:ok, state}
end

# Функция для внутреннего использования
def fall_velocity(distance) do
  :math.sqrt(2 * 9.8 * distance)
end
end

```

Имя модуля (`DropServer`) должно быть знакомо по предыдущим примерам. Вторая строка указывает, что этот модуль используется модулем `GenServer`.

Вложенное объявление `defmodule` также должно быть знакомо; оно определяет структуру с единственным полем для хранения счетчика вызовов. Многие службы хранят в таких структурах много полей, в том числе подключения к базе данных, ссылки на другие процессы, информацию о сети и метаданные, характерные для данной конкретной службы. Точно так же можно создать службу, не имеющую состояния, определив здесь пустой кортеж. Как вы увидите далее, каждая отдельная функция `GenServer` ссылается на это состояние.



Объявление структуры `State` – хороший пример объявлений, которые должны находиться внутри модуля, а не в отдельном файле. Может так получиться, что вам потребуется использовать модели состояния в разных процессах, использующих `GenServer`, но намного удобнее, когда содержимое структуры `State` находится перед глазами.

Первая функция в примере, `start_link/0`, не используется функциями из модуля `GenServer`. Она вызывает функцию `GenServer.start_link`, чтобы запустить процесс. Это удобно для тестирования на начальных этапах освоения ОТП. При переходе к разработке промышленного кода вы можете обнаружить, что лучше отказаться от `start_link/0` и использовать другие механизмы.

Функция `start_link/0` использует встроенное объявление `__MODULE__`, которое возвращает имя текущего модуля:

```

# Вспомогательный метод для запуска
def start_link do

```

```
GenServer.start_link(__MODULE__, [], [{:name, __MODULE__}])
end
```

Первый аргумент – атом (`__MODULE__`) – разворачивается в имени текущего модуля, и это имя будет использоваться как имя данного процесса. Далее следуют список аргументов для передачи в процедуру инициализации модуля и список параметров настройки. Параметры могут определять, например, режим отладки, тайм-ауты и параметры для порождаемых процессов. По умолчанию имя процесса регистрируется в локальном реестре Elixir. Так как нам требуется зарегистрировать со всеми связанными узлами, мы указали в списке параметров кортеж `{:name, __MODULE__}`.



Иногда можно встретить вызов `GenServer.start_link` с атомом `:via` в кортеже с параметрами. Он позволяет задействовать собственные реестры процессов, самым известным из которых является `gproc` (<https://github.com/uwiger/gproc>).

Все остальные функции являются частью поведения `GenServer`. Функция `init/1` создает новый экземпляр структуры состояния со значением 0 в поле `count` – ни одного вычисления скорости еще не производилось. Основную работу выполняют функции `handle_call/3` и `handle_cast/2`. В данном примере `handle_call/3` ожидает получить высоту в метрах и возвращает скорость с данной высоты на Земле, а `handle_cast/2` вызывается, чтобы изменить счетчик вычислений.

Функция `handle_call/3` превращает асинхронные взаимодействия между процессами Erlang в простую задачу:

```
def handle_call(request, _from, state) do
  distance = request
  reply = {:ok, fall_velocity(distance)}
  new_state = %State{count: state.count + 1}
  {:reply, reply, new_state}
end
```

Она извлекает высоту (`distance`) из запроса (`request`), что, впрочем, необязательно – мы лишь хотели сохранить имена переменных, как в шаблоне (с таким же успехом можно было объявить эту функцию как `handle_call(distance, _from, state)`). На практике аргумент `request` чаще оказывается кортежем или списком, а не простым значением, но в простых случаях можно использовать и такой подход.

Затем функция создает ответ, посылая высоту (`distance`) простой функции `fall_velocity/1` в конце модуля. Далее она создает новое состояние `new_state`, содержащее увеличенный счетчик вызовов. Затем

добавляет атом `:reply` в кортеж с ответом, содержащий вычисленную скорость, и новым состоянием `new_state`, содержащим изменившийся счетчик, и возвращает его.

Так как вычисления действительно очень простые, выполнение их в виде синхронного вызова вполне оправданно. В более сложных ситуациях, когда трудно предсказать, как долго будет вычисляться ответ, можно вернуть ответ `:noreply` и использовать аргумент `_from` для отправки действительного ответа позднее. (Существует также вариант ответа `:stop`, который приводит к вызову метода `:terminate/2` и остановке процесса.)



По умолчанию фреймворк OTP ограничивает время выполнения синхронных запросов пятью секундами. Этот порог можно изменить явно, вызовом `GenServer.call/3` с новым значением тайм-аута (в миллисекундах), или использовав атом `:infinity`.

Функция `handle_cast/2` реализует поддержку асинхронных взаимодействий. Она, как предполагается, не возвращает ответ непосредственно, но должна вернуть `:noreply` (или `:stop`) и обновленное состояние. В данном случае она использует весьма слабый подход (но вполне пригодный для демонстрации), вызывая `IO.puts/1` для вывода числа вызовов:

```
def handle_cast(_msg, state) do
  IO.puts("So far, calculated #{state.count} velocities.")
  {:noreply, state}
end
```

Само состояние не изменяется, потому что запрос количества обращений за получением скорости падения – это еще не сами вычисления.

Пока нет серьезных оснований изменять его, вы можете оставить в покое функции `handle_info/2`, `terminate/2` и `code_change/3`.

Запуск и вызов процесса с помощью `GenServer` выглядит немного иначе, чем запуск процессов, продемонстрированный в главе 9:

```
iex(1)> DropServer.start_link()
{:ok, #PID<0.46.0>}
iex(2)> GenServer.call(DropServer, 20)
{:ok, 19.79898987322333}
iex(3)> GenServer.call(DropServer, 40)
{:ok, 28.0}
iex(4)> GenServer.call(DropServer, 60)
{:ok, 34.292856398964496}
iex(5)> GenServer.cast(DropServer, {})
```

```
So far, calculated 3 velocities.  
:ok
```

Вызов `DropServer.start_link()` запускает процесс и подготавливает его к приему запросов. Затем вы можете использовать `GenServer.call` или `GenServer.cast` для отправки сообщений и получения ответов.



Несмотря на возможность получить идентификатор процесса, нет необходимости хранить его, чтобы использовать процесс. Тем не менее функция `start_link` возвращает кортеж, и если вам понадобится сохранить идентификатор, вы можете выполнить вызов этой функции, например так: `{:ok, pid} = DropServer.start_link()`.

Так как работа OTP основана на вызовах функций `GenServer`, вы получаете дополнительное преимущество – или недостаток – возможность обновления кода прямо в процессе выполнения. Например, представьте, что мы изменили ускорение свободного падения на Земле, уменьшив его с 9,8 до 9,1 м/с². Скомпилировав код и запросив скорость, вы получите другой ответ:

```
iex(6)> r(DropServer)  
warning: redefining module DropServer (current version loaded from  
  _build/dev/lib/drop_server/ebin/Elixir.DropServer.beam)  
  lib/drop_server.ex:1  
  
warning: redefining module DropServer.State (current version loaded from  
  _build/dev/lib/drop_server/ebin/Elixir.DropServer.State.beam)  
  lib/drop_server.ex:4  
  
{:reloaded, DropServer, [DropServer.State, DropServer]}  
iex(7)> GenServer.call(DropServer, 60)  
{:ok, 33.04542328371661}
```

Это очень удобно на этапе разработки, но требует осторожного отношения на сервере, находящемся в промышленной эксплуатации. Фреймворк OTP имеет в своем арсенале другие механизмы обновления кода во время выполнения. Кроме того, этот подход имеет внутреннее ограничение: `init` вызывается, только когда `start_link` запускает службу. То есть она не вызывается после перекомпиляции кода. Если новый код предполагает какие-либо изменения в структуре состояния, код может потерпеть аварию при следующем обращении к нему.

Простой супервизор

Запуская модуль `DropServer` из интерактивной оболочки, вы фактически превращаете оболочку в супервизор модуля, которая, впрочем, не

выполняет никаких функций, свойственных супервизору. Вы легко можете остановить модуль:

```
iex(8)> GenServer.call(DropServer, -60)
** (EXIT from #PID<0.141.0>) an exception was raised:
    ** (ArithmeticError) bad argument in arithmetic expression
        (stdlib) :math.sqrt(-1176.0)
        (drop_server) lib/drop_server.ex:44: DropServer.fall_velocity/1
        (drop_server) lib/drop_server.ex:20: DropServer.handle_call/3
        (stdlib) gen_server.erl:615: :gen_server.try_handle_call/4
        (stdlib) gen_server.erl:647: :gen_server.handle_msg/5
        (stdlib) proc_lib.erl:247: :proc_lib.init_p_do_apply/3
```

Interactive Elixir (1.3.1) - press Ctrl+C to exit (type h() ENTER for help)

```
iex(1)>
10:50:58.899 [error] GenServer DropServer terminating
** (ArithmeticError) bad argument in arithmetic expression
    (stdlib) :math.sqrt(-1176.0)
    (drop_server) lib/drop_server.ex:44: DropServer.fall_velocity/1
    (drop_server) lib/drop_server.ex:20: DropServer.handle_call/3
    (stdlib) gen_server.erl:615: :gen_server.try_handle_call/4
    (stdlib) gen_server.erl:647: :gen_server.handle_msg/5
    (stdlib) proc_lib.erl:247: :proc_lib.init_p_do_apply/3
Last message: -60
State: %DropServer.State{count: 5}
```

Сообщение об ошибке достаточно подробное и даже содержит последнее сообщение и состояние, но если теперь попробовать обратиться к службе, у вас ничего не получится, потому что оболочка IEx перезапустилась. Можно, конечно, вручную вызвать `DropServer.start_link/0` еще раз, но нельзя же постоянно сидеть перед экраном и лично наблюдать за работой процессов.

Вместо этого было бы желательно иметь еще что-то, что наблюдало бы за вашими процессами и перезапускало их при необходимости. Фреймворк ОТР формализует управление процессом, который был показан в примере 9.10, в виде поведения *Supervisor*.

Простейший супервизор должен реализовать единственную функцию обратного вызова, `init/1`, и может также иметь функцию `start_link` для его запуска. Функция `init/1` должна сообщать фреймворку ОТР, какими дочерними процессами управляет ваш супервизор и как должны обрабатываться их отказы. В примере 13.2 (*ch13/ex2-drop-sup*) показано, как мог бы выглядеть супервизор для модуля `Drop`.

Пример 13.2 ❖ Простой супервизор

```
defmodule DropSup do
  use Supervisor
```

```
# вспомогательный метод для запуска
def start_link do
  Supervisor.start_link(__MODULE__, [], [{:name, __MODULE__}])
end

# обратный вызов супервизора

def init([]) do
  child = [worker(DropServer, [], [])]
  supervise(child, [{:strategy, :one_for_one}, {:max_restarts, 1},
                   {:max_seconds, 5}])
end

# внутренние функции (здесь их нет)
end
```

Задача функции `init/1` – определить процесс или процессы, за которыми должен следить супервизор, и как должны обрабатываться отказы.

Функция `worker/3` определяет модуль, который должен запускать супервизор, список его аргументов и любые параметры для передачи функции `start_link` этого модуля. В данном примере под наблюдением находится только один дочерний процесс, а параметры передаются как список кортежей с парами ключ/значение.



Параметры можно также передавать в виде списка пар ключей и значений, который можно записать так:

```
supervise(child, [strategy: :one_for_one, max_restarts: 1,
                  max_seconds: 5])
```

Функция `supervise/2` принимает список дочерних процессов в первом аргументе и список параметров во втором.

Значение `:one_for_one` в параметре `:strategy` сообщает фреймворку OTP, что он должен создавать новый дочерний процесс, если процесс с параметром `:permanent` (по умолчанию) неожиданно завершился. Также можно использовать значение `:one_for_all`, чтобы завершать и повторно запускать все процессы, находящиеся под наблюдением супервизора, если хотя бы один из них завершился, или `:rest_for_one`, чтобы обеспечить повторный запуск остановившегося процесса и любых других процессов, запущенных после него.



Когда вы будете готовы перейти к более непосредственному управлению процессами, обратите внимание на динамические функции: `Supervisor.start/2`, `Supervisor.terminate_child/2`, `Supervisor.restart_child/2` и `Supervisor.delete_child/2`, а также на стратегию повторного запуска (параметр `:strategy`) `:simple_one_for_one`.

Следующие два параметра (`:max_restarts` и `:max_seconds`) определяют, как часто наблюдаемый процесс может завершаться аварийно, прежде чем завершится сам супервизор. В данном случае перезапуск допускается не чаще, чем один раз в пять секунд. Подстройка этого параметра позволит вам адаптировать приложение под конкретные условия, но на этапе освоения ОТР это будет требоваться нечасто. (Значение 0 в параметре `:max_restarts` означает, что супервизор будет просто завершаться, если дочерний процесс потерпит аварию.)

Функция `supervise` принимает описанные аргументы и создает структуру данных для использования фреймворком ОТР. По умолчанию служба получает параметр `:permanent`, то есть супервизор всегда должен повторно запускать дочерний процесс в случае его сбоя. Определяя дочерний процесс, можно указать параметр `:restart`, чтобы изменить это поведение. По умолчанию супервизор ждет пять секунд, прежде чем полностью остановить дочерний процесс; этот период можно изменить с помощью параметра `:shutdown` в определении дочернего процесса. Более сложные ОТР-приложения могут содержать целое дерево супервизоров, управляющих другими супервизорами, которые, в свою очередь, управляют третьими супервизорами или рабочими процессами. Чтобы запустить дочерний процесс-супервизор, используйте функцию `supervisor/3`, которой должны передаваться те же аргументы, что и функции `worker/3`.



Фреймворк ОТР должен знать о зависимостях между процессами, чтобы иметь возможность динамически обновлять действующий код. Это необходимо для обновления системы без полной ее остановки.

Теперь, имея процесс-супервизор, вы легко сможете запустить DropServer простым вызовом супервизора. Однако запуск супервизора из интерактивной оболочки вызовом функции `start_link/0` создает свое множество проблем; оболочка сама является супервизором и завершает процессы, сообщившие об ошибке. Под длинным отчетом об ошибке вы можете увидеть строки, сообщающие, что ваш рабочий процесс и супервизор были остановлены.

На практике это означает, что необходимо разорвать связь между ОТР-процессами и оболочкой. Метод, демонстрируемый ниже, явно разрывает связь между оболочкой и процессом-супервизором, путем предварительного определения идентификатора процесса супервизора (строка 1) и последующего вызова `Process.unlink/1` (строка 2). После этого процесс можно вызывать как обычно, с помощью `GenServer.call/2`, и получать ответы. Ошибка (как показано в строке 5),

не приведет к остановке супервизора. Супервизор повторно запустит рабочий процесс, и вы сможете продолжить благополучно обращаться к нему. Вызовы `Process.whereis(DropServer)` в строках 3 и 6 демонстрируют, что супервизор действительно перезапускает `DropServer` с новым идентификатором:

```
iex(1)> {:ok, pid} = DropSup.start_link()
{:ok,#PID<0.44.0>}
iex(2)> Process.unlink(pid)
true
iex(3)> Process.whereis(DropServer)
#PID<0.45.0>
iex(4)> GenServer.call(DropServer, 60)
{:ok,34.292856398964496}
iex(5)> GenServer.call(DropServer, -60)
** (exit) exited in: GenServer.call(DropServer, -60, 5000)
    ** (EXIT) an exception was raised:
        ** (ArithmeticError) bad argument in arithmetic expression
           (stdlib) :math.sqrt(-1176.0)
           (drop_sup) lib/drop_server.ex:44: DropServer.fall_velocity/1
           (drop_sup) lib/drop_server.ex:20: DropServer.handle_call/3
           (stdlib) gen_server.erl:615: :gen_server.try_handle_call/4
           (stdlib) gen_server.erl:647: :gen_server.handle_msg/5
           (stdlib) proc_lib.erl:247: :proc_lib.init_p_do_apply/3

11:05:00.438 [error] GenServer DropServer terminating
** (ArithmeticError) bad argument in arithmetic expression
   (stdlib) :math.sqrt(-1176.0)
   (drop_sup) lib/drop_server.ex:44: DropServer.fall_velocity/1
   (drop_sup) lib/drop_server.ex:20: DropServer.handle_call/3
   (stdlib) gen_server.erl:615: :gen_server.try_handle_call/4
   (stdlib) gen_server.erl:647: :gen_server.handle_msg/5
   (stdlib) proc_lib.erl:247: :proc_lib.init_p_do_apply/3
Last message: -60
State: %DropServer.State{count: 1}
(elixir) lib/gen_server.ex:604: GenServer.call/3
iex(5)> GenServer.call(DropServer, 60)
{:ok,34.292856398964496}
iex(6)> Process.whereis(DropServer)
#PID<0.46.0>
```



Также можно открыть диалог **Process Manager** (Диспетчер процессов) в окне **Observer** и, остановив рабочий процесс выбором пункта **Kill** (Остановить) в меню **Trace** (Слежение), наблюдать за его повторным появлением.

Все это выглядит очень интересно, но это лишь малая часть того, на что способны супервизоры. Они могут создавать дочерние процессы динамически и намного точнее управлять их жизненным циклом.

Упаковка приложения с помощью Mix

В этом разделе мы попробуем использовать Mix для создания приложения из только что написанных супервизора и сервера.

Повторим действия, которые мы уже выполняли в разделе «Запуск» в главе 1. Создайте каталог для приложения командой `mix new name`, как показано ниже:

```
$ mix new drop_app
* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/drop_app.ex
* creating test
* creating test/test_helper.exs
* creating test/drop_app_test.exs
```

Your Mix project was created successfully.

You can use "mix" to compile it, test it, and more:

```
cd drop_app
mix test
```

Run "mix help" for more commands.

Диспетчер проектов Mix автоматически создал необходимые файлы и каталоги. Перейдите в каталог `drop_app`, созданный диспетчером Mix. Затем откройте файл `mix.exs` в текстовом редакторе. Прежде мы ничего не говорили об этих файлах, потому что нам не требовалось изменять их содержимое, но теперь пришло время сделать это. Итак, взгляните:

```
defmodule DropApp.Mixfile do
  use Mix.Project

  def project do
    [app: :drop_app,
     version: "0.0.1",
     elixir: "~> 1.3",
     build_embedded: Mix.env == :prod,
     start_permanent: Mix.env == :prod,
     deps: deps]
  end

  # Конфигурация OTP-приложения
  #
```

```

# Введите команду "mix help compile.app", чтобы получить больше информации
def application do
  [applications: [:logger]]
end

# Зависимости могут быть пакетами Hex:
#
# {:mydep, "~> 0.3.0"}
#
# Или репозиториум git/path:
#
# {:mydep, git: "https://github.com/elixir-lang/mydep.git", tag: "0.1.0"}
#
# Введите команду "mix help deps", чтобы получить
# дополнительные примеры и описание параметров

defp deps do
  []
end
end

```

Функция `project/0` присваивает имя вашему приложению, номер версии и определяет зависимости для сборки проекта.

Зависимости возвращаются функцией `deps/0`. Пример в комментариях демонстрирует, что у вас должен иметься проект `mydep` версии 0.3.0 или выше, доступный в `git` по указанному адресу URL. Помимо `git`, можно указать локальный каталог с зависимостями (`path:`).

В данном примере приложение не имеет зависимостей, поэтому тело функции можно оставить в текущем его виде.

После ввода команды `mix compile` диспетчер Mix скомпилирует пустой проект. Если после этого заглянуть в каталог проекта, можно увидеть, что Mix создал подкаталог `_build` для скомпилированного кода:

```

$ mix compile
Compiling 1 file (.ex)
Generated drop_app app
$ ls
_build config lib mix.exs README.md test

```

От пустого приложения мало проку, поэтому скопируем файлы `drop_server.ex` и `drop_sup.ex` в папку `lib`. Затем выполним команду `iex -S mix`. Mix скомпилирует новые файлы, и вы сможете приступить к использованию сервера:

```

$ iex -S mix
Erlang/OTP 19 [erts-8.0] [source] [64-bit] [smp:4:4] [async-threads:10]
[hipe] [kernel-poll:false]

```

```

Compiling 2 files (.ex)
Generated drop_app app
Interactive Elixir (1.3.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> {:ok, pid} = DropServer.start_link()
{:ok, #PID<0.60.0>}

```

Последние шаги, которые осталось сделать: написать код самого приложения и затем сообщить диспетчеру Mix, где он находится.

Внутри *mix.exs* измените функцию `application/0`, как показано ниже:

```

def application do
  [ applications: [:logger],
    registered: [:drop_app],
    mod: {DropApp, []} ]
end

```

Ключу `:registered` соответствует список всех имен, регистрируемых приложением (в данном случае это единственное имя `:drop_app`), а ключу `:mod` соответствует кортеж с именем модуля, который должен запускаться в момент запуска приложения, и списком аргументов, передаваемых этому модулю. Ключу `:applications` соответствует список приложений, от которых зависит данное приложение во время выполнения.

Ниже приводится код, который мы добавили в модуль `DropApp` (*ch13/ex3-drop-app/drop_app/lib/drop_app.ex*):

```

defmodule DropApp do
  use Application

  def start(_type, _args) do
    IO.puts("Starting the app...") # сообщит, что приложение запущено
    DropSup.start_link()
  end
end

```

Функция `start/2` является обязательной. Первый аргумент сообщает, как виртуальная машина, на которой выполняется Elixir, должна обрабатывать аварийные сбои приложения. Во втором аргументе передается список аргументов, соответствующий ключу `:mod` в приложении `application/0`. Функция `start/2` должна вернуть кортеж вида `{:ok, pid}`, то есть именно то, что возвращает `DropSup.start_link/0`.

Если теперь ввести команду `mix compile`, диспетчер Mix сгенерирует файл `_build/dev/lib/drop_app/ebin/drop_app.app`. (Заглянув в этот файл, вы увидите кортеж на языке Erlang, содержащий массу инфор-

мации, извлеченной из уже созданных файлов.) После этого приложение можно запустить из командной строки:

```
$ elixir -pa _build/dev/lib/drop_app/ebin --app drop_app  
Starting the app...
```

В этой главе мы лишь коснулись самых основ. Вообще, тема программирования с применением ОТР заслуживает отдельной книги (или даже нескольких книг). Надеемся, что здесь нам удалось дать вам достаточно информации, чтобы вы могли поэкспериментировать и уже со знанием дела читать такие книги. Однако считаем своим долгом сказать, что разрыв между объемом информации, который мы смогли уместить в эту главу, и тем, который нужен для разработки надежных ОТР-приложений, огромен.

Глава 14

Расширение языка Elixir с помощью макросов

Теперь вы обладаете достаточным объемом знаний, чтобы самостоятельно писать на Elixir интересные и очень мощные программы. Однако иногда возникает потребность расширить сам язык, чтобы упростить код и тем самым облегчить его чтение или реализовать какие-то новые возможности. Макросы в Elixir позволяют сделать это.

Функции и макросы

Макросы очень похожи на функции, отличаясь только тем, что начинаются с инструкции `defmacro` вместо `def`. Однако макросы действуют совершенно иначе, чем функции. Чтобы лучше понять суть различий, рассмотрим пример 14.1 (*ch14/ex1-difference*).

Пример 14.1 ❖ Демонстрация различий между вызовами функций и макросов

```
defmodule Difference do

  defmacro m_test(x) do
    IO.puts("#{inspect(x)}")
    x
  end

  def f_test(x) do
    IO.puts("#{inspect(x)}")
    x
  end

end
```

Чтобы получить возможность пользоваться макросом, нужно подключить модуль с его определением при помощи инструкции `require`. Введите следующие команды в интерактивной оболочке:

```
iex(1)> require Difference
Difference
iex(2)> Difference.f_test(1 + 3)
4
4
iex(3)> Difference.m_test(1 + 3)
{:+, [line: 3], [1, 3]}
4
```

Строка 2 возвращает ожидаемый результат – Elixir вычисляет выражение `1 + 3` и передает его значение функции `f_test`, которая выводит число 4 и возвращает его.

Строка 3 может немного удивить. Вместо того чтобы вычислить выражение, она интерпретирует аргумент как кортеж. Макрос возвращает кортеж, и затем этот кортеж передается интерпретатору Elixir для оценки.



Первый элемент кортежа – это оператор, второй элемент – список метаданных об операции, и третий элемент – список операндов.

Простой макрос

Так как `defmacro` получает код до того, как Elixir получит шанс вычислить его, макрос имеет все возможности трансформировать код перед передачей его Elixir для вычисления. В примере 14.2 демонстрируется макрос, создающий код, который удваивает аргумент. (Ту же задачу проще было бы решить при помощи функции, но наша цель – разобраться с макросами.) Он создает кортеж, который Elixir распознает как операцию умножения. Этот пример можно найти в папке *ch14/ex2-double*.

Пример 14.2 ❖ Макрос, удваивающий аргумент

```
defmodule Double do
  defmacro double x do
    {:+, [], [2, x]}
  end
end
```

После компиляции его можно опробовать в интерактивной оболочке:

```
iex(1)> require Double
Double
iex(2)> Double.double(3)
6
iex(3)> Double.double(3 * 7)
42
```

Он работает, но должен же быть более простой путь? Было бы неплохо иметь возможность сказать: «Преврати этот код на Elixir во внутренний формат», – чтобы создать кортеж при помощи самого Elixir. И действительно, такая возможность *имеется*, реализованная в виде функции `quote`, которая принимает произвольное выражение на языке Elixir и преобразует его во внутреннее представление:

```
iex(4)> quote do: 1 + 3
{:+, [context: Elixir, import: Kernel], [1, 3]}
iex(5)> x = 20
20
iex(6)> quote do: 3 * x + 20
{:+, [context: Elixir, import: Kernel],
 [{:*, [context: Elixir, import: Kernel], [3, {:x, [], Elixir}]], 20}]}
```

Как видите, `quote` принимает обычный код на Elixir и преобразует его во внутренний формат. У кого-то из вас наверняка тут же появились соблазны переписать макрос из предыдущего примера, как показано ниже:

```
defmodule Double do
  defmacro double(x) do
    quote do: 2 * x
  end
end
```

Но этот код не будет работать. Причина в том, что он говорит: «Преврати `2 * x` в кортеж», но `x` уже имеет форму кортежа, поэтому нужен какой-то способ сообщить Elixir, что аргумент следует оставить в покое. Как раз для этого пример 14.3 (*ch14/ex3-double*) использует функцию `unquote/1`.

Пример 14.3 ❖ Использование `quote` для создания макроса

```
defmodule Double do
  defmacro double(x) do
    quote do
```



```

    2 * unquote(x)
  end
end
end

```

Он говорит: «Преврати `2 * x` во внутреннее представление, но оставь аргумент `x` в исходном виде – он не требует преобразования»:

```

iex(7)> r(Double)
warning: redefining module Double (current version loaded from
  _build/dev/lib/double/ebin/Elixir.Double.beam)
  /Users/elixir/code/ch14/ex3-double/lib/double.ex:1

{:reloaded, Double, [Double]}
double.ex:1: redefining module Double
[Double]
iex(8)> require Double
Double
iex(9)> Double.double(3 * 5)
30

```



Многие, кто пишет макросы, допускают типичную ошибку, забывая передать аргументы функции `unquote`. Помните, что все аргументы передаются макросам во внутреннем формате.

Подведем итог: `quote` означает: «Преврати все в блоке `do` в формат внутреннего кортежа»; `unquote` означает: «Не превращай это во внутренний формат». (Имена `quote` и `unquote` были заимствованы из языка программирования Lisp.)



Если функции `quote` передать атом, число, список, строку или кортеж с двумя элементами, она просто вернет свой аргумент, а не кортеж во внутреннем формате.

Создание новой логики

Макросы позволяют добавлять в язык новые команды. Например, если бы в Elixir отсутствовала конструкция `unless` (которая является противоположностью для конструкции `if`), ее можно было бы добавить в язык, написав макрос, представленный в примере 14.4 (*ch14/ex4-unless*).

Пример 14.4 ❖ Макрос, реализующий конструкцию `unless`

```

defmodule Logic do
  defmacro unless(condition, options) do
    quote do

```

```

        if(!unquote(condition), unquote(options))
    end
end
end

```

Этот макрос принимает условие (`condition`) и параметры (`options`) во внутреннем представлении и преобразует их в код для эквивалентной инструкции `if` с обратным условием. Как и в предыдущих примерах, условие и параметры должны оставаться в неприкосновенности, так как они уже находятся во внутреннем формате. Проверим макрос в интерактивной оболочке, предварительно скомпилировав модуль командой `iex -S mix`:

```

iex(1)> require(Logic)
Logic
iex(2)> Logic.unless (4 == 5) do
... (2)> IO.puts("arithmetic still works")
... (2)> end
arithmetic still works
:ok

```

Программное создание функций

В Elixir все конструкции имеют внутреннее представление, даже функции. Это означает, что макрос может принять данные на входе и вернуть настроенную функцию.

В примере 14.5 (*ch14/ex5-programmatic*) демонстрируется простой макрос `create_multiplier`, который принимает атом и множитель и возвращает функцию, именем которой является атом, умножающую входное число на заданный множитель.

Пример 14.5 ❖ Использование макроса для программного создания функции

```

defmodule FunctionMaker do
  defmacro create_multiplier(function_name, factor) do
    quote do
      def unquote(function_name)(value) do
        unquote(factor) * value
      end
    end
  end
end
end

```

Чтобы вызвать этот макрос, необходимо определить другой модуль:

```
defmodule Multiply do
  require FunctionMaker

  FunctionMaker.create_multiplier(:double, 2)
  FunctionMaker.create_multiplier(:triple, 3)

  def example do
    x = triple(12)
    IO.puts("Twelve times 3 is #{x}")
  end
end
```

Скомпилировав оба модуля, вы сможете использовать новые функции:

```
iex(1)> Multiply.double(21)
42
iex(2)> Multiply.triple(54)
162
iex(3)> Multiply.example()
Twelve times 3 is 36
:ok
```



Вы не сможете определить функцию программным способом за пределами модуля или внутри функции.

Можно даже написать единственный макрос, создающий несколько разных функций. Если, к примеру, понадобится определить отдельную функцию `drop/1` для каждой планеты, для этого можно написать макрос, принимающий список планет с ускорениями свободного падения и создающий эти функции. Пример 14.6 (*ch14/ex6-multidrop*) создаст функции `mercury_drop/1`, `venus_drop/1` и т. д. из списка ключей.

Пример 14.6 ❖ Создание нескольких функций при помощи макроса

```
defmodule FunctionMaker do

  defmacro create_functions(planemo_list) do
    Enum.map planemo_list, fn {name, gravity} ->
      quote do
        def unquote(:"#{name}_drop")(distance) do
          :math.sqrt(2 * unquote(gravity) * distance)
        end
      end
    end
  end
end
```

В выражении `:"#{name}_drop"` используется механизм интерполяции для добавления строки `"drop"` в конец имени планеты и начального двоеточия (`:`) для преобразования результата в атом. Все выражение передается функции `unquote`, так как атомы передаются во внутреннем формате.

И снова, чтобы вызвать макрос, требуется определить другой модуль:

```
defmodule Drop do
  require FunctionMaker

  FunctionMaker.create_functions([{:mercury, 3.7}, {:venus, 8.9},
    {:earth, 9.8}, {:moon, 1.6}, {:mars, 3.7},
    {:jupiter, 23.1}, {:saturn, 9.0}, {:uranus, 8.7},
    {:neptune, 11.0}, {:pluto, 0.6}])
end
```

После компиляции вам будут доступны 10 новых функций:

```
iex(1)> Drop.earth_drop(20)
19.79898987322333
iex(2)> Drop.moon_drop(20)
8.0
```

Когда (не) следует использовать макросы

К настоящему моменту вы увидели несколько хороших примеров простых программ. Однако все, что демонстрировалось в этой главе, легко можно реализовать при помощи функций. Пока вы изучаете Elixir, не бойтесь экспериментировать с макросами. Но, разрабатывая программы общего пользования и испытывая соблазн написать макрос, спросите себя: «Можно ли то же самое реализовать в виде функции?» Если вы ответите на этот вопрос утвердительно (а часто так и есть), используйте функции. Макросы следует использовать, только когда они действительно смогут облегчить жизнь пользователям вашего кода.

Тогда зачем было поднимать волну, описывая макросы, если от их применения следует воздерживаться? Во-первых, сам Elixir очень широко использует макросы. Например, когда вы определяете запись, Elixir программно генерирует функции для доступа к полям этой записи. Даже `def` и `defmodule` — это макросы!

Что более важно, информация из этой главы поможет вам понять, как действуют макросы, которые могут вам встретиться в чужом коде. (Это сродни изучению иностранного языка; существуют фразы, которых вы сами никогда не произносите, но которые хотелось бы понимать, если их произносит кто-то другой.)

Глава 15

Phoenix

Elixir прекрасно подходит для создания приложений командной строки, но иногда требуется написать веб-приложение. Фреймворк Phoenix предлагает комплект инструментов на основе Elixir, напоминающий Ruby on Rails, для создания веб-приложений. Он гарантирует высокую надежность и масштабируемость и основан на макросах, ОТР и сервере Erlang Cowboy. Фреймворк скрывает эти мощные возможности настолько, что вы можете начать писать простые приложения, совершенно не владея ими.

Установка базовых компонентов фреймворка

После установки Elixir установка Phoenix выполняется просто. Полный перечень компонентов, которые могут понадобиться фреймворку Phoenix, включает PostgreSQL и Node.js. Описание их установки выходит за рамки данного раздела, но, даже имея только базовые компоненты Phoenix, вы сможете писать довольно полезные приложения.

Для начала установим Phoenix при помощи Mix (команды в примере ниже мы отформатировали, чтобы уместить их по ширине книжной страницы, но у себя вы должны вводить команды в одну строку):

```
$ mix archive.install
https://github.com/phoenixframework/archives/raw/master/phoenix_new.ez
Are you sure you want to install archive "https://github.com/phoenixframework/
archives/raw/master/phoenix_new.ez"? [Yn] y
* creating .mix/archives/phoenix_new
```

Установив Phoenix, вы сможете создать минимально возможное приложение. Директива `--no-brunch` выключает поддержку управляемых ресурсов в Phoenix, которая необходима для установки Node. Ди-

ректива `--no-ecto` выключает механизм реляционного отображения объектов (Object Relational Mapping, ORM), который предполагает наличие установленной базы данных PostgreSQL:

```
$ mix phoenix.new fall --no-brunch --no-ecto
```

```
* creating web/config/config.exs
```

```
* creating web/config/dev.exs
```

```
...
```

```
* creating web/web/views/layout_view.ex
```

```
* creating web/web/views/page_view.ex
```

```
Fetch and install dependencies? [Yn] y
```

```
* running mix deps.get
```

We are all set! Run your Phoenix application:

```
$ cd fall
```

```
$ mix phoenix.server
```

You can also run your app inside IEx as:

```
$ iex -S mix phoenix.server
```

Фреймворк создаст внутри проекта несколько каталогов, как показано на рис. 15.1.

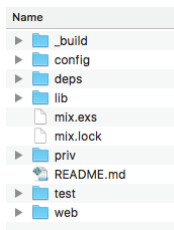


Рис. 15.1 ❖ *Каталоги, созданные фреймворком Phoenix*

В оставшейся части главы мы будем работать исключительно в каталоге *web*.

Воспользуемся советом Phoenix и перейдем в каталог *fall*, затем запустим фреймворк:

```
$ cd fall
```

```
$ iex -S mix phoenix.server
```



При первом запуске Phoenix может предложить установить локальные копии дополнительного программного обеспечения. В нашем случае потребовалась установка Rebar, которая прошла без всяких осложнений.

Первый запуск Phoenix потребует некоторого времени на сборку перед запуском. После множества примечаний о файлах, скомпилированных фреймворком, вы увидите:

Generated web app

[info] Running Web.Endpoint with Cowboy using http://localhost:4000

После этого на вашем компьютере запустится простой веб-сайт, принимающий соединения на порту 4000. Если в браузере перейти по адресу *http://localhost:4000*, вы увидите страницу с приветствием, как показано на рис. 15.2.

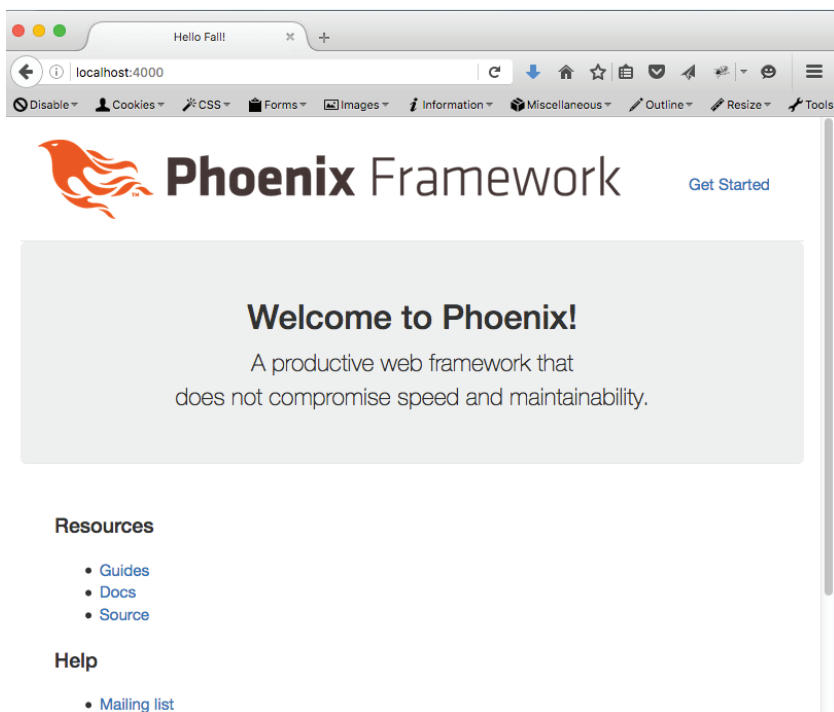


Рис. 15.2 ❖ Страница фреймворка Phoenix с приветствием

Принадлежность этой страницы к вашему приложению можно определить по заголовку «Hello Fall!» на вкладке. В командной строке, где вы запускали фреймворк, должны появиться примерно такие строки:


```
[info] GET /  
[debug] Processing by Web.PageController.index/2  
Parameters: %{}  
Pipelines: [:browser]  
[info] Sent 200 in 2ms
```

Структура простого Phoenix-приложения

Минимальное приложение на основе фреймворка Phoenix, которое ничего не делает (кроме отображения страницы приветствия!), требует вашего вмешательства в реализацию четырех разделов:

- *маршрутизатор* – логика маршрутизации определяет, что должен делать фреймворк в ответ на прием того или иного адреса URL;
- *контроллер* – это своеобразный распределительный щит для входящей и исходящей информации, соединяющий запросы на вычисления с вычислениями;
- *представление* получает результаты вычислений от контроллера и форматирует их при помощи шаблона;
- *шаблон* объединяет разметку HTML с переменными, поступающими из контроллера, используя логику представления.

В этом простом приложении мы пропустили букву «М» в аббревиатуре «MVC» (Model-View-Controller – модель-представление-контроллер) – Модель, – которая управляет взаимодействиями с данными. Обычно для управления данными Phoenix использует библиотеку Ecto, которая, в свою очередь, использует PostgreSQL, а для этого необходимо пройти более сложную процедуру установки, чем было описано выше.



Если вам понравилась база данных Mnesia, ее тоже можно использовать с фреймворком Phoenix, однако это довольно тернистый путь и, чтобы пройти его, сначала необходимо изучить Phoenix. Дополнительную информацию можно найти в статье «Using mnesia database from Elixir» на сайте AmberBit (<https://www.amberbit.com/elixir-cocktails/elixir/using-mnesia-database-from-elixir/>).

Представление страницы

Чтобы получить HTML-страницу, отличную от страницы по умолчанию, необходимо реализовать все четыре компонента. Реализация HTML-страницы может показаться более сложной, чем вычисление скоростей падающих объектов, тем не менее, овладев этим умением,

вы получите прочный фундамент, на котором сможете создать что-то более сложное. Кроме того, преодолевая ошибки, вызванные неполной установкой, вы получите знания, которые определенно пригодятся вам в будущем. Когда что-то идет не так, как надо, для начала следует обратить внимание на уровень, в котором возникла проблема.

Попробуйте запустить Phoenix и открыть страницу <http://localhost:4000/welcome>. В результате вы увидите страницу, как показано на рис. 15.3.

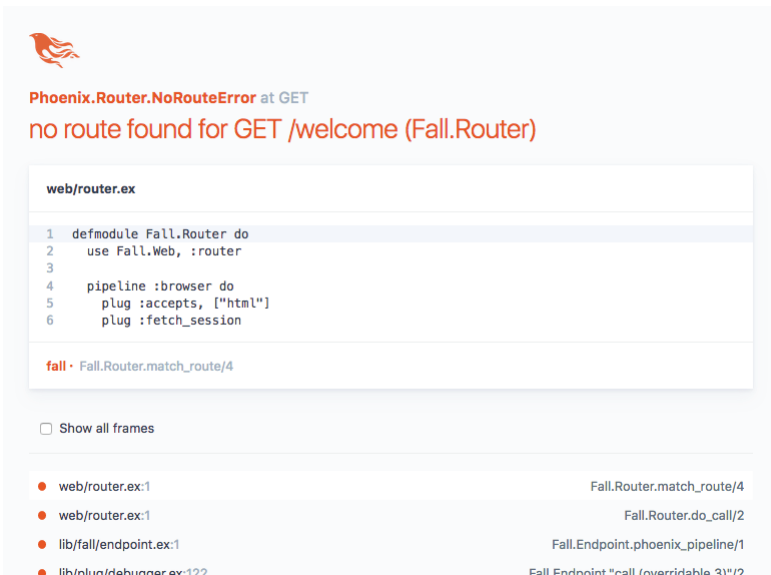


Рис. 15.3 ❖ Фреймворку требуется маршрут

Маршрутизация

Фреймворку Phoenix требуются явные инструкции, указывающие, куда направлять запросы по конкретному адресу URL. Он знает, как отобразить страницу с приветствием, потому что в файле *web/router.ex* присутствует следующая информация:

```
defmodule Fall.Router do
  use Fall.Web, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
  end
end
```

```
plug :fetch_flash
plug :protect_from_forgery
plug :put_secure_browser_headers
end

pipeline :api do
  plug :accepts, ["json"]
end

scope "/", Fall do
  pipe_through :browser # Использовать стек браузера по умолчанию
  get "/", PageController, :index
end

# Другие области видимости могут использовать другие стеки.
# scope "/api", Fall do
#   pipe_through :api
# end
end
```

В этом файле много интересного. Во-первых, он явно указывает, что приложение называется `Fall`. Ссылки на `Fall.Web` здесь будут встречаться очень часто, поэтому, скопировав код из другого приложения, не забудьте исправить все эти ссылки. (`Fall.Web` определяется в файле `web/web.ex`, но сейчас мы не будем трогать его.)

Модуль `Router` определяет также два конвейера. Первый служит для отображения содержимого в браузере, обычно в формате HTML, а второй — для передачи содержимого в формате JSON. Пока это все, что вам нужно знать о них.

Поддержка нового URL для HTML-интерфейса определяется в функции `scope`. Phoenix знает, как показать страницу по умолчанию, потому что инструкция `get "/", PageController, :index` запускает целый ряд необходимых событий. `get` сообщает Phoenix, что данный маршрут используется для обработки HTTP-запросов GET. Чтобы реализовать поддержку своего адреса URL, добавьте чуть ниже строку:

```
get "/welcome", FallController, :welcome
```

Когда маршрутизатор получит запрос к адресу `/welcome`, он передаст информацию из запроса функции `welcome` в `FallController`. Это означает, что теперь вы получите другую ошибку, как показано на рис. 15.4.



Рис. 15.4 ❖ Фреймворку требуется контроллер

Простой контроллер

Phoenix ожидает найти контроллер в каталоге *web/controllers*. Изначально в этом каталоге имеется только один файл, *page_controller.ex*, который является контроллером страницы с приветствием. Заглянув в него, можно увидеть, как выглядит типичный контроллер:

```
defmodule Fall.PageController do
  use Fall.Web, :controller

  def index(conn, _params) do
    render conn, "index.html"
  end
end
```

Файл контроллера определяет модуль, опирающийся на код в модуле *Web* приложения. Маршрутизатор также опирается на него, но маршрутизатор обозначается как *:router*, а контроллер (что неудивительно) — как *:controller*.

Все функции в этом модуле должны принимать соединение (*conn*) с параметрами и (возможно, после некоторой обработки) вызывать *render* с аргументом, представляющим соединение, и именем файла, отсылаемого обратно.

Чтобы определить модуль *FallController*, ссылка на который добавлена в файл маршрутизатора, создайте файл *fall_controller.ex* в каталоге *web/controllers* и добавьте в него функцию *welcome*:

```
defmodule Fall.FallController do
  use Fall.Web, :controller

  def welcome(conn, params) do
    render conn, "welcome.html"
  end
end
```

Затем перезагрузите страницу *http://localhost:4000/welcome*. Следующая ошибка сообщает об отсутствии представления, как показано на рис. 15.5.



Рис. 15.5 ❖ Когда контроллер ссылается на несуществующее представление

Простое представление

Новые представления создаются в два этапа. Во-первых, нужно добавить реализацию представления в *web/views/fall.ex*, которая в данном случае выглядит элементарно:

```
defmodule Fall.FallView do
  use Fall.Web, :view
end
```

Реализация добавлена, но новое сообщение об ошибке говорит, что это еще не все, как показано на рис. 15.6.



Phoenix.Template.UndefinedError at GET

Could not render "welcome.html" for Fall.FallView, please define a matching clause for render/2 or define a template at "web/templates/fall". No templates were compiled for this module. Assigns:

```
%{conn: %Plug.Conn{adapter: {Plug.Adapters.Cowboy.Conn, ...}, assigns: %{layout: {Fall.LayoutView, "ap
web/templates/layout/app.html.eex
```

Рис. 15.6 ❖ Когда контроллер
ссылается на несуществующий шаблон

На втором этапе необходимо создать шаблон. Контроллер требует отобразить страницу *welcome.html*, поэтому вы должны создать шаблон этой страницы в файле *web/templates/fall/welcome.html.eex*. Расширение *.eex* расшифровывается как Embedded Elixir (встроенный Elixir), часть Elixir, которая позволяет создавать строковые шаблоны, обрабатываемые Elixir. В данном случае Phoenix использует EEx для создания HTML-страниц, которые затем передаются браузеру для отображения. Шаблоны не обязаны создавать страниц целиком: Phoenix уже содержит шаблоны для оберты вашего содержимого, обеспечивая единство внешнего вида приложений. То есть ваш шаблон может состоять всего из одной строки:

```
<h1>Falling starts now!</h1>
```

Такой шаблон преобразуется в страницу, изображенную на рис. 15.7. Вам может даже не потребоваться перезагружать страницу, потому что в режиме разработки Phoenix автоматически пытается перезагрузить страницу, если обнаружил ошибку.



Рис. 15.7 ❖ После создания всех необходимых компонентов
получается ожидаемый результат

Возможно, вы захотите убрать логотип Phoenix Framework из всех своих проектов. Для этого откройте файл *web/templates/layout/app.html.eex* и удалите элемент *header*.



Есть возможность при помощи Phoenix сгенерировать простой код контроллера и представления командой `mix phoenix.gen.html`, но она также сгенерирует модель, которая не будет работать в данном простом примере.

Вычисления

Теперь, закончив создание страницы, пришло время реализовать обработку данных, посылаемых серверу. То есть нам предстоит узнать, как создать простую форму и как реализовать выборку параметров из HTTP-запросов, что немного сложнее, чем прием параметров в функциях.

Использование Phoenix предполагает создание длинной цепочки файлов, помогающих сохранить код организованным с увеличением сложности приложений, но большинство изменений, которые вам придется внести в процессе конструирования простого приложения, такого как это, коснутся только контроллера и шаблона, да еще иногда придется заходить в файл *router.ex*. Пока вы не приступите к созданию сложного приложения, требующего повторного использования кода между несколькими интерфейсами, вам не придется вносить много изменений в другие файлы.

Для начала зададим посетителю несколько вопросов на странице с приветствием. Данный пример использует те же вычисления скорости, что и многие другие примеры выше, поэтому сосредоточим основное внимание на форме и контроллере, а не на вычислениях. Phoenix поддерживает базовый набор функций для создания безопасной разметки HTML из предоставляемых данных.

Поместите следующие строки в файл *web/templates/fall/welcome.html.eex*:

```
<h1>Falling starts now!</h1>

<%= form_for @conn, fall_path(@conn, :faller), [as: :calculation],
fn f -> %>
  <%= select f, :planemo, @choices %>
  <%= text_input f, :distance %>
  <%= submit "Calculate" %>
<% end %>
```

Тег `h1` остался прежним – это самый обычный заголовок, но появился новый тег `form_for`. Код на Elixir, заключенный в `<%=` и `%>`, производит результаты, которые автоматически преобразуются в разметку HTML. (Если код заключить в `<%` и `%>`, без `=`, код на Elixir выполнится точно так же, но его результаты не будут преобразованы в HTML.) В данном случае `form_for` – это функция, производящая разметку HTML с элементом формы `form`, и каждая строка в этой функции также производит некоторый код HTML (как часть формы).

Для нормальной работы формам HTML необходимо несколько составляющих. Они должны включать элементы управления для получения ввода пользователя. В данном случае нам понадобятся раскрывающийся список и текстовое поле ввода. Кнопка отправки формы дает пользователю возможность сообщить форме, что данные введены и готовы к отправке.

Но куда? Куда должны отправляться данные? Функция `form_for` в Phoenix сама позаботится об этом. Первые аргументы как раз определяют, откуда пришел запрос и куда должна отправляться форма с данными. Аргумент `@conn` содержит информацию о соединении, через которое была передана эта форма. Далее эта форма с данными посылается другой странице, которой пока не существует, подключенной к методу `:faller` контроллера. Функция `fall_path` возвращает информацию, необходимую для передачи данных из формы в браузер обратно на сервер Phoenix.

Аргумент `[as: :calculation]` совершенно необходим, потому что данная форма не связана с моделью данных. Элемент `as:` сообщает функции `form_for`, как представлена информация, полученная вместе с формой (мы еще вернемся к этому аргументу, когда будем рассматривать его обработку в контроллере). `fn f` создает точку привязки, которую используют другие элементы формы для доступа к информации, относящейся к форме целиком.

Функции `select` и `text_input`, обе, получают эту точку привязки `f` в первом аргументе. Во втором аргументе им передается имя поля, которое будет использовано контроллером для извлечения данных. Функция `select` принимает также третий аргумент, `@choices`. (Символ `@` обозначает *ссылку* на переменную, определяемую контроллером.) В нем находятся элементы для поля с раскрывающимся списком, но сама переменная определяется контроллером, которого пока нет.

Если запустить приложение прямо сейчас, шаблон потерпит неудачу еще до того, как заметит отсутствие переменной `@choices` (рис. 15.8).



ArgumentError at GET

No helper clause for `Fall.Router.Helpers.fall_path/2` defined for action `:faller`. The following `fall_path` actions are defined under your router:

* `:welcome`

Рис. 15.8 ❖ И снова ошибка, на этот раз сообщающая об отсутствии маршрута к несуществующей странице

Чтобы создать новую страницу, необходимо добавить новую запись в файл с определениями маршрутов, а также создать новый набор файлов с контроллером, представлением и шаблоном. Откройте файл `web/router.ex` и найдите метод `scope`. Добавьте строку `get "/fall", FallController, :faller` сразу за определением маршрута `/welcome`:

```
scope "/", Fall do
  pipe_through :browser # Использовать стек браузера по умолчанию

  get "/welcome", FallController, :welcome
  get "/fall", FallController, :faller
  get "/", PageController, :index
end
```

Это помогло избавиться от ошибки, но теперь, после определения маршрута, Phoenix терпит неудачу из-за отсутствия переменной, как показано на рис. 15.9.



ArgumentError at GET

assign @choices not available in eex template.

Please make sure all proper assigns have been set. If this is a child template, ensure assigns are given explicitly by the parent template as they are not automatically forwarded.

Available assigns: `[:conn, :view_module, :view_template]`

Рис. 15.9 ❖ Теперь обнаружено отсутствие переменной

Чтобы добавить переменную `@choices`, необходимо включить одну новую строку кода и изменить вызов `render`:

```
def welcome(conn, params) do
  choices = ["Earth": 1, "Moon": 2, "Mars": 3]
  render conn, "welcome.html", choices: choices
end
```

Содержимое списка `choices` может показаться вам знакомым по ранним примерам, и теперь Phoenix имеет все необходимое, чтобы создать форму, как показано на рис. 15.10.

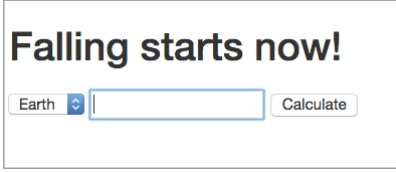


Рис. 15.10 ❖ Готовая форма

Отлично! Заглянув в форму, сгенерированную фреймворком Phoenix, можно получить представление, что делает функция `form_for`:

```
<form accept-charset="UTF-8" action="/fall" method="post">
<input name="_csrf_token" type="hidden"
value="bnAhLkknegAwUXENPw48ByQ5FTYmJgAAZDjE0C+mY4+KNyu00WwCHA==">
<input name="_utf8" type="hidden" value="✓">

<select id="calculation_planemo" name="calculation[planemo]">
  <option value="1">Earth</option>
  <option value="2">Moon</option>
  <option value="3">Mars</option>
</select>

<input id="calculation_distance" name="calculation[distance]"
type="text">

<input type="submit" value="Calculate">
</form>
```

Функция `form_for` сгенерировала довольно много типового кода разметки и немного, — характерного для данной конкретной формы. Значение для атрибута `action` взято из определения маршрута. Информация о кодировке UTF-8 (и скрытое поле `_utf8`), метод (POST) и дополнительное скрытое поле `_csrf_token` используются по умолчанию. Элемент `select` сконструирован из указанного содержимого,

а его имя получено объединением значения аргумента `as:` с именем формы. Это упростит выборку данных из формы позднее.

Теперь выберем планету, введем высоту и щелкнем на кнопке **Calculate**. Увы, мы не получили ожидаемого результата, как показано на рис. 15.11.

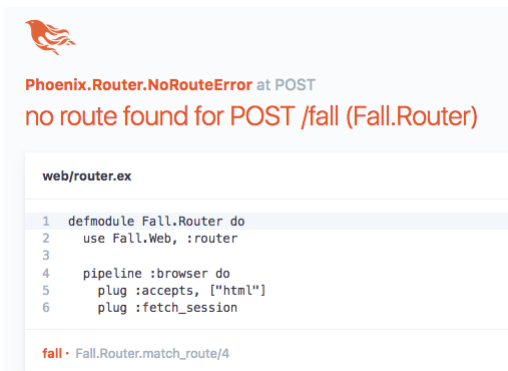


Рис. 15.11 ❖ Не найден маршрут для обработки запросов POST

Так как `form_for` создает форму, которая по умолчанию пересылает данные HTTP-методом POST, а добавленный прежде маршрут обрабатывает только GET-запросы, Phoenix не смог определить, куда направить форму (хотя можно сказать, что он сам создал такую ситуацию). Исправим эту проблему, добавив маршрут для метода POST в файл `web/router.ex`:

```
scope "/", Fall do
  pipe_through :browser # Использовать стек браузера по умолчанию

  get "/welcome", FallController, :welcome
  get "/fall", FallController, :faller
  post "/fall", FallController, :faller
  get "/", PageController, :index
end
```

Теперь маршрут определен, но, как показано на рис. 15.12, необходимо реализовать обработку формы.

Это последнее сообщение об ошибке, которое вы увидите. К счастью, исправить ее будет проще, потому что основные изменения коснутся только контроллера. Добавьте в файл `web/controllers/fall_controller.ex` новую функцию:

```
def faller(conn, params) do
  choices = ["Earth": 1, "Moon": 2, "Mars": 3]
  speed = 0
  render conn, "faller.html", speed: speed, choices: choices
end
```

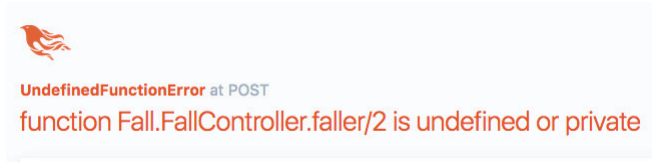


Рис. 15.12 ❖ *Необходима функция, обрабатывающая форму*

Да, она просто игнорирует данные, полученные с формой, но сейчас она позволит нам сконструировать шаблон и заодно убедиться, что все соединения работают исправно, а после этого мы со спокойной душой сможем углубиться в обработку параметров.

В каталоге *web/templates/fall* создайте новый файл *faller.html.eex* и заполните его содержимым файла *welcome.html.eex*, с одним важным исключением — добавьте ссылку в `@speed` в заголовок:

```
<h1>Speed at impact was <%= @speed %> m/s.</h1>

<p>Fall again?</p>

<%= form_for @conn, fall_path(@conn, :faller), [as: :calculation],
  fn f -> %>
  <%= select f, :planemo, @choices %>
  <%= text_input f, :distance %>
  <%= submit "Calculate" %>
<% end %>
```

Вообще, проверку формы после этих изменений можно пропустить, но давайте все же посмотрим, что получилось (см. рис. 15.13).



Рис. 15.13 ❖ *Скорость может быть нулевой, но данные попали в нужный контроллер*

Чтобы выполнить вычисления и вернуть полученную скорость, нужно извлечь данные из полученной формы. В окне терминала, где мы запускали Phoenix, можно увидеть примерно такие строки:

```
[info] POST /fall
[debug] Processing by Fall.FallController.faller/2
  Parameters: %{"_csrf_token" => "bnAhLkknegAwUXENPw48ByQ5FTYmJgA
AZDjE0C+mY4+KNyu00WwCHA=", "_utf8" => "✓", "calculation" =>
%{"distance" => "20", "planemo" => "1"}}
  Pipelines: [:browser]
[info] Sent 200 in 664µs
```

Параметры `distance` и `planemo` были переданы в составе отображения, завернутого – как указано в `form_for` – в параметр `calculation`. На выбор есть несколько способов извлечения этих параметров. Можно воспользоваться функцией `Map.get`. Однако более идиоматическим для языка Elixir является способ на основе применения механизма сопоставления с образцом. К тому же этот прием упрощает реализацию контроллеров, обрабатывающих запросы без параметров.

Чтобы извлечь параметры, откройте файл `web/controllers/fall_controller.ex`. Добавьте еще одну функцию `faller`, использующую сопоставление с образцом, а не просто принимающую параметры, и несколько уже знакомых функций `fall_velocity`, выполняющих вычисления:

```
def faller(conn, %{"calculation" => %{"planemo" => planemo,
"distance" => distance}}) do
  calc_planemo = String.to_integer(planemo)
  calc_distance = String.to_integer(distance)
  speed = fall_velocity(calc_planemo, calc_distance)
  choices = ["Earth": 1, "Moon": 2, "Mars": 3]
  render conn, "faller.html", speed: speed, choices: choices
end

def fall_velocity(1, distance) do
  :math.sqrt(2 * 9.8 * distance)
end

def fall_velocity(2, distance) do
  :math.sqrt(2 * 1.6 * distance)
end

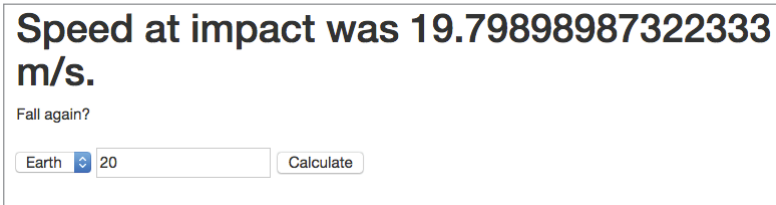
def fall_velocity(3, distance) do
  :math.sqrt(2 * 3.71 * distance)
end
```

Сопоставление с образцом, заменившее параметр `params` в заголовке функции, `%{"calculation" => %{"planemo" => planemo, "distance" =>`

`distance}}`}, обнаруживает совпадение с отображением `calculation`, которое должно поступать в случае успешной отправки формы. Оно также разбивает отображение на две составляющие: `planemo` и `distance`.

Следующие две строки решают другую проблему, связанную с обработкой параметров, получаемых из веб-форм: все данные поступают в виде строк. Функция `String.to_integer/1` выполняет необходимое преобразование, и в будущих своих проектах вам также придется явно преобразовывать параметры при необходимости. Дополнительно сюда можно добавить проверку допустимости значений, чтобы предотвратить ошибки, вызванные передачей нечисловых данных, но для начала данной реализации вполне достаточно.

Вычисления выполняются по знакомой схеме, и результат, изображенный на рис. 15.14, тоже должен выглядеть знакомым.



Speed at impact was 19.79898987322333 m/s.

Fall again?

Earth 20 Calculate

Рис. 15.14 ❖ Вывод вычисленной скорости

Очевидно, что можно продолжить развивать это приложение, например добавить возможность ввода и вывода данных в формате JSON, но обсуждение этой темы далеко выходит за рамки данной книги. Однако если у вас есть желание, попробуйте двинуться дальше. Онлайн-документация для Phoenix содержит достаточно подробностей, а книга «Programming Phoenix» (издательства Pragmatic Programmers) поможет вам глубже вникнуть в суть.

Продвижение Elixir

Эта глава завершает введение в Elixir, но помните, что Elixir – это молодой язык с развивающейся экосистемой, и он обладает множеством особенностей, которые вам еще предстоит освоить.

На первый взгляд кажется, что не составит никакого труда привести доводы в пользу Elixir. Широкая область применения, от отдельных компьютеров до сетевых и распределенных систем, а также

поддержка параллельных вычислений дают окружению Elixir/Erlang огромное преимущество практически перед любыми другими окружениями. Все чаще компьютерный мир сталкивается с трудностями, которые с легкостью преодолевают Elixir и Erlang. Ветераны, имеющие большой опыт борьбы с этими трудностями, могут вздохнуть с облегчением, потому что смогут перестать подбирать наборы инструментов, которые несут односистемные подходы в мультисистемный мир.

Но в то же время мы хотели бы поделиться с вами мнением Джо Армстронга (Joe Armstrong): «Новые технологии имеют больше шансов на развитие а) сразу после катастрофы или б) в начале нового проекта».

Мы допускаем мысль, что вы читаете эту книгу, потому что проект, над которым вы работали, постигла катастрофа (или вы подозреваете, что она скоро произойдет), однако намного проще и предпочтительнее начинать использовать Elixir для разработки новых проектов, желательно таких, где ошибки новичков не приведут к новым катастрофам.

Найдите проект, который будет интересен вам и полезен вашей организации или всему миру. Нет лучшего способа разрекламировать мощь языка программирования и его окружения, чем создать что-то значимое!

Приложение А

Каталог элементов языка Elixir

Как и во всех других языках, в Elixir имеется свой ящик, до отказа забитый разными интересными инструментами, многие из которых доступны благодаря Erlang.

В этом приложении перечислены наиболее типичные из них, и все они представлены с использованием соглашений об именах, принятых в Elixir. Если вам захочется (сильно-сильно) получить больше информации, загляните в руководство «Erlang User Guide» (http://erlang.org/doc/reference_manual/users_guide.html).

Команды интерактивной оболочки

В интерактивной оболочке можно использовать большую часть функций Elixir, но в табл. А.1 перечислены только команды, доступные исключительно в оболочке.

Таблица А.1. Команды интерактивной оболочки Elixir

Команда	Описание
<code>c(file)</code>	Компилирует указанный файл <i>file</i> в код на языке Erlang
<code>c(file, path)</code>	Компилирует указанный файл <i>file</i> и помещает объектный код в каталог <i>path</i>
<code>ls()</code>	Выводит список файлов в текущем каталоге
<code>ls(path)</code>	Выводит список файлов в указанном каталоге <i>path</i>
<code>cd(directory)</code>	Выполняет переход в указанный каталог <i>directory</i>
<code>pwd()</code>	Выводит путь к текущему каталогу
<code>clear()</code>	Очищает экран

Таблица A.1 (окончание)

Команда	Описание
<code>h()</code>	Выводит список доступных вспомогательных функций
<code>h(item)</code>	Выводит справку по указанному элементу <i>item</i>
<code>l(module)</code>	Загружает указанный модуль <i>module</i>
<code>m()</code>	Выводит список всех загруженных модулей
<code>r(module)</code>	Повторно компилирует указанный модуль <i>module</i>
<code>v()</code>	Выводит список всех команд, выполнявшихся в текущем сеансе, и возвращаемых ими значений
<code>v(n)</code>	Выводит значение, выведенное <i>n</i> -й командой
<code>flush()</code>	Выталкивает все сообщения, отправленные в оболочку

Зарезервированные слова

В Elixir имеется несколько зарезервированных слов, которые нельзя использовать за пределами отведенного им контекста.

Компилятор Elixir старается понять, что вы пытаетесь сделать, когда используете определенные ключевые слова в качестве имен функций (список таких слов приводится в табл. A.2). Он может интерпретировать ваши функции как код, и в результате вы будете получать очень странные ошибки.

Таблица A.2. Зарезервированные слова, требующие осторожного использования

after	and	catch	do	else	end	false	fn
in	nil	not	or	rescue	true	when	

Проблема решается просто: используйте что-нибудь другое.

Кроме того, имеется несколько атомов, часто используемых в качестве возвращаемых значений (см. табл. A.3). Они не являются зарезервированными словами, но все же лучше использовать их только там, где они ожидаются.

Таблица A.3. Атомы, часто используемые в качестве возвращаемых значений

Атом	Описание
<code>:ok</code>	Нормальное завершение метода. <i>Не</i> может считаться признаком успешного выполнения
<code>:error</code>	Что-то пошло не так. Обычно сопровождается подробным описанием
<code>:undefined</code>	Значение не было присвоено. Обычно используется в экземплярах записей

Таблица А.3 (окончание)

Атом	Описание
:reply	Ответ, сопровождающийся дополнительным возвращаемым значением
:noreply	Ответ без возвращаемого значения. Позднее может быть получен фактический ответ со значением
:stop	Используется в ОТР, чтобы сообщить, что сервер должен быть остановлен. Приводит к вызову функции <code>terminate</code>
:ignore	Возвращается процессом-супервизором ОТР, который не смог запустить дочерний процесс

Операторы

Таблица А.4. Логические (булевы) операторы

Оператор	Оператор	Описание
and	&&	Логическое «И»
or		Логическое «ИЛИ»
not	!	Логическое «НЕ» (унарный оператор)

Логический оператор `not` имеет наивысший приоритет. Операторы `and`, `&&`, `or` и `||` выполняются по короткой схеме. Они прекращают выполнение, как только ответ становится очевиден.

Операторы в первой колонке требуют, чтобы их аргументы были логическими значениями. Операторы во второй колонке принимают любые выражения: любые возвращаемые ими значения, кроме `false` и `nil`, интерпретируются как `true`. Операторы `&&` и `||` выполняются по короткой схеме и возвращают результат, как только он станет очевиден, не вычисляя оставшихся выражений. Например, `nil || 5` вернет 5, а `nil && 5` вернет `nil`.

Таблица А.5. Операторы сравнения

Оператор	Описание
<code>==</code>	Равно
<code>!=</code>	Не равно
<code><=</code>	Меньше или равно
<code><</code>	Меньше
<code>>=</code>	Больше или равно
<code>></code>	Больше
<code>===</code>	Строго равно
<code>!==</code>	Строго не равно

Операторы, перечисленные в табл. А.5, способны сравнивать элементы разных типов. В отношении разнотипных данных действуют следующие правила отношений «больше»/«меньше»:

число < атом < ссылка < функция < порт < идентификатор процесса < кортеж < список < битовая строка

Допускается сравнивать целые числа с вещественными, за исключением более специфических операторов `===` и `!==`, которые всегда возвращают `false` при сравнении чисел разных типов.

Допускается сравнивать между собой кортежи, даже содержащие нечисловые значения. Elixir выполнит перебор элементов сравниваемых кортежей слева направо и поочередно сравнит каждую пару элементов, прервав операцию, как только результат станет очевидным.

Таблица А.6. Арифметические операторы

Оператор	Описание
<code>+</code>	Унарный плюс (положительное значение)
<code>-</code>	Унарный минус (отрицательное значение)
<code>+</code>	Сложение
<code>-</code>	Вычитание
<code>*</code>	Умножение
<code>/</code>	Вещественное деление

Чтобы выполнить деление нацело и получить целочисленный остаток от деления, используйте функции `div` и `rem`. Например, `div(17, 3)` вернет 5, а `rem(17, 3)` вернет 2.

Таблица А.7. Двоичные операторы

Оператор	Оператор	Описание
<code>bnot</code>	<code>~</code>	Унарный, поразрядная инверсия
<code>band</code>	<code>&&&</code>	Поразрядное «И»
<code>bor</code>	<code> </code>	Поразрядное «ИЛИ»
<code>bxor</code>	<code>^</code>	Поразрядное «ИСКЛЮЧАЮЩЕЕ ИЛИ»
<code>bsl</code>	<code><<<</code>	Поразрядный сдвиг влево
<code>bsr</code>	<code>>>></code>	Поразрядный сдвиг вправо

Если у вас возникнет необходимость в этих операторах и функциях, импортируйте модуль `Bitwise`.

Таблица А.8. Приоритет операторов от большего к меньшему

Оператор	Ассоциативность
Унарные + - ! ^ not ~~~	Нет
=~ >	Правая
++ -- **	Правая
<>	Правая
* /	Левая
+ -	Левая
&&&	Левая
..	Левая
in	Левая
< > <= >= == === != !==	Левая
and	Левая
or	Левая
&&	Левая
	Левая
<-	Правая
=	Правая
	Правая
//	Правая
when	Правая
::	Правая
,	Левая
->	Правая
do	Левая
@	Нет

Операторы с высшим приоритетом вычисляются в выражениях первыми. Вычисление операторов с одинаковым приоритетом выполняется в направлении ассоциативности. (Левоассоциативные операторы выполняются слева направо, правоассоциативные – справа налево.)

Ограничители

Elixir разрешает использовать в выражениях ограничителей лишь некоторые функции и другие возможности, не имеющие побочных эффектов. Ниже приводится список элементов языка, допустимых в ограничителях:

- true;
- другие константы (оцениваемые как false);
- операторы сравнения (табл. A.5);
- арифметические выражения (табл. A.6 и табл. A.7);
- логические выражения и логические операторы: and и or;
- следующие функции: hd/1, 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_number/1, is_pid/1, is_port/1, is_record/1, is_record/2, is_reference/1, is_term/2, is_tuple/1.

Часто используемые функции

Таблица A.9. Математические функции

Функция	Результат
:math.pi/0	Число «пи»
:math.sin/1	Синус
:math.cos/1	Косинус
:math.tan/1	Тангенс
:math.asin/1	Арксинус
:math.acos/1	Аркосинус
:math.atan/1	Арктангенс
:math.atan2/2	Арктангенс от частного аргументов
:math.sinh/1	Гиперболический синус
:math.cosh/1	Гиперболический косинус
:math.tanh/1	Гиперболический тангенс
:math.asinh/1	Гиперболический арксинус
:math.acosh/1	Гиперболический аркосинус
:math.atanh/1	Гиперболический арктангенс
:math.exp/1	Экспонента
:math.log/1	Натуральный логарифм (по основанию e)
:math.log10/1	Десятичный логарифм (по основанию 10)
:math.pow/2	Возводит первый аргумент в степень второго аргумента
:math.sqrt/1	Квадратный корень
:math.erf/1	Функция ошибки
:math.erfc/1	Дополнение функции ошибки

Аргументы всех тригонометрических функций должны быть выражениями, возвращающими значения в радианах. Чтобы преобразовать градусы в радианы, разделите число градусов на 180 и умножьте на число «пи».



Функции `erf/1` и `erfc/1` могут быть не реализованы в Windows. Кроме того, документация Erlang предупреждает, что «не все функции реализованы на всех платформах», но эти две конкретные функции должны иметься в библиотеках языка C.

Таблица А.10. Функции высшего порядка для обработки коллекций и списков (принимают в аргументах перечислимые объекты и функции)

Функция	Возвращает	Описание
<code>Enum.each/2</code>	<code>:ok</code>	Может производить побочный эффект, обусловленный действиями передаваемой функции
<code>Enum.map/2</code>	Новый список	Применяет указанную функцию к элементам списка
<code>Enum.filter/2</code>	Подмножество	Создает список, содержащий элементы, для которых функция вернула <code>true</code>
<code>Enum.all?/2</code>	Логическое значение	Возвращает <code>true</code> , если функция вернула <code>true</code> , для всех элементов, иначе возвращает <code>false</code>
<code>Enum.any?/2</code>	Логическое значение	Возвращает <code>true</code> , если функция вернула <code>true</code> хотя бы для одного элемента, иначе возвращает <code>false</code>
<code>Enum.take_while/2</code>	Подмножество	Отбирает элементы с головы списка, пока функция возвращает <code>true</code>
<code>Enum.drop_while/2</code>	Подмножество	Удаляет элементы с головы списка, пока функция возвращает <code>true</code>
<code>List.foldl/3</code>	Аккумулятор	Передает функции значение из списка и аккумулятор, выполняя перебор элементов списка от начала к концу
<code>List.foldr/3</code>	Аккумулятор	Передает функции значение из списка и аккумулятор, выполняя перебор элементов списка от конца к началу
<code>Enum.partition/2</code>	Кортеж с двумя списками	Разбивает список с помощью условия, реализуемого функцией

Более подробно эти функции описываются в главе 8.

Таблица А.11. Экранированные последовательности для использования в строках

Последовательность	Производит
<code>\"</code>	Двойную кавычку
<code>\'</code>	Одиночную кавычку
<code>\\</code>	Обратный слеш (Backslash)
<code>\b</code>	Забой (Backspace)
<code>\d</code>	Удаление (Delete)

Таблица A.11 (окончание)

Последовательность	Производит
\e	Экранирующий символ (Escape)
\f	Перевод формата (Form feed)
\n	Перевод строки (Line feed)
\r	Возврат каретки (Carriage return)
\s	Пробел (Space)
\t	Табуляцию (Tab)
\v	Вертикальную табуляцию (Vertical tab)
\xXY	Код символа в шестнадцатеричном виде
\x{X...}	Код символа в шестнадцатеричном виде, где X... — один или несколько шестнадцатеричных цифр
^a...^z или ^A...^Z	От Ctrl+A до Ctrl+Z

Таблица A.12. Строковые метки

Метка	Описание
%c %C	Возвращает список символов
%r %R	Возвращает регулярное выражение
%s %S	Возвращает двоичную строку
%w %W	Возвращает список слов

Метки с буквой в нижнем регистре допускают экранирование и интерполяцию, как в обычных строках; метки с буквой в верхнем регистре создают строки в точности, как они определены в коде, не допуская экранирования и не выполняя интерполяции.

Типы данных для использования в документации и анализа

Таблица A.13. Основные типы данных для @spec и ExDoc

atom()	binary()	float()	fun()	integer()	list()	tuple()
union()	node()	number()	String.t()	char()	byte()	[] (nil)
any()	none()	pid()	port()	reference()		

Тип `String.t()` используется в Elixir для хранения двоичных данных; тип `string()` — для того, что в Erlang называется строками, но списками символов в Elixir. За дополнительными подробностями о типах обращайтесь к руководству пользователя.

Приложение В

Создание документации с помощью ExDoc

В главе 2 вы узнали, как добавлять описание в свои программы. Инструмент ExDoc способен извлекать эти описания и производить красиво отформатированную справочную документацию в виде веб-страниц. ExDoc работает совместно с Mix, инструментом создания, компиляции и тестирования проектов. Более полную информацию о Mix можно найти в главе 13.

Использование ExDoc вместе с Mix

Самый простой способ создать документацию – создать проект с использованием инструмента Mix командой:

```
mix new project_name
```

Вот как выглядит процесс создания документации для кода в примере 2.4:

```
$ mix new combined
* creating README.md
* creating .gitignore
* creating mix.exs
* creating lib
* creating lib/combined.ex
* creating test
* creating test/test_helper.exs
```



```
* creating test/combined_test.exs
```

Your mix project was created successfully.

You can use "mix" to compile it, test it, and more:

```
cd combined
mix test
```

Run "mix help" for more commands.

Перейдите в каталог *combined* и поместите все файлы с исходным кодом (*combined.ex*, *drop.ex* и *convert.ex* для этого примера) в каталог *lib*. Файл *combined.ex* при этом заменит одноименный файл, созданный Mix.

Теперь отредактируйте *mix.exs*, изменив функцию *deps*, как показано ниже:

```
def deps do
  [{:ex_doc, github: "elixir-lang/ex_doc"}]
end
```

Выполните команду `mix deps.get`, чтобы установить ExDoc в каталог *deps*. Если вы еще не установили Hex (диспетчер пакетов для Elixir), Mix предложит сделать это. Теперь можно скомпилировать сразу все файлы командой `mix compile`:

```
$ mix compile
==> ex_doc
Compiled lib/ex_doc/cli.ex
Compiled lib/ex_doc.ex
Compiled lib/ex_doc/markdown/cmark.ex
Compiled lib/ex_doc/markdown/earmark.ex
Compiled lib/ex_doc/markdown.ex
Compiled lib/ex_doc/markdown/hoedown.ex
Compiled lib/ex_doc/markdown/pandoc.ex
Compiled lib/mix/tasks/docs.ex
Compiled lib/ex_doc/formatter/html.ex
Compiled lib/ex_doc/retriever.ex
Compiled lib/ex_doc/formatter/html/templates.ex
Compiled lib/ex_doc/formatter/html/autolink.ex
Generated ex_doc app
==> combined
Compiled lib/convert.ex
Compiled lib/combined.ex
Compiled lib/drop.ex
Generated combined app
Consolidated List.Chars
Consolidated IEx.Info
```

```
Consolidated String.Chars
Consolidated Collectable
Consolidated Enumerable
Consolidated Inspect
```

После этого можно сгенерировать документацию командой `mix docs`. Если у вас установлен процессор Markdown, эта процедура должна пройти гладко. Если нет (у одного из авторов книги он отсутствовал в системе Linux), может появиться сообщение об ошибке:

```
** (RuntimeError) Could not find a markdown processor to be used on ex_doc.
You can either:
```

1. Add `{:markdown, github: "devinus/markdown"}` to your `mix.exs` deps
2. Ensure pandoc (<http://johnmacfarlane.net/pandoc>) is available in your system

(Перевод:

```
** (RuntimeError) Не найден процессор markdown для использования в ex_doc.
Вы можете:
```

1. Добавить `{:markdown, github: "devinus/markdown"}` в функцию `deps`, в своем файле `mix.exs`
2. Установить pandoc (<http://johnmacfarlane.net/pandoc>) в своей системе

В данном случае первый вариант кажется проще, поэтому мы изменили функцию `deps` в файле `mix.exs`, как показано ниже:

```
defp deps do
  [ {:ex_doc, github: "elixir-lang/ex_doc"},
    {:markdown, github: "devinus/markdown"} ]
end
```

После этого мы повторно выполнили команду `mix deps.get`:

```
* Getting markdown (git://github.com/devinus/markdown.git)
Cloning into '/Users/code/ex6-docs/combined/deps/markdown'...
remote: Reusing existing pack: 83, done.
remote: Total 83 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (83/83), 12.52 KiB | 0 bytes/s, done.
Resolving deltas: 100% (34/34), done.
Checking connectivity... done.
* Getting hoedown (git://github.com/hoedown/hoedown.git)
Cloning into '/Users/code/ex6-docs/combined/deps/hoedown'...
remote: Counting objects: 1869, done.
remote: Compressing objects: 100% (805/805), done.
remote: Total 1869 (delta 1050), reused 1869 (delta 1050)
Receiving objects: 100% (1869/1869), 504.60 KiB | 481.00 KiB/s, done.
Resolving deltas: 100% (1050/1050), done.
Checking connectivity... done.
```

Затем заново выполнили команду `mix docs` (которая скомпилировала процессор Markdown), вновь скомпилировали файлы с исходным кодом на Elixir и, наконец, создали документы. Ниже приводится вывод этой команды без сообщений компилятора:

```
Compiled lib/markdown.ex
Generated markdown.app
==> combined
Compiled lib/convert.ex
Compiled lib/combined.ex
Compiled lib/drop.ex
Generated combined.app
%{green}Docs successfully generated.
%{green}View them at "doc/index.html".
```

Естественно, теперь в списке каталогов присутствует папка *doc*, содержащая файл *index.html*. На рис. В.1 показано, как выглядит документация в окне браузера.

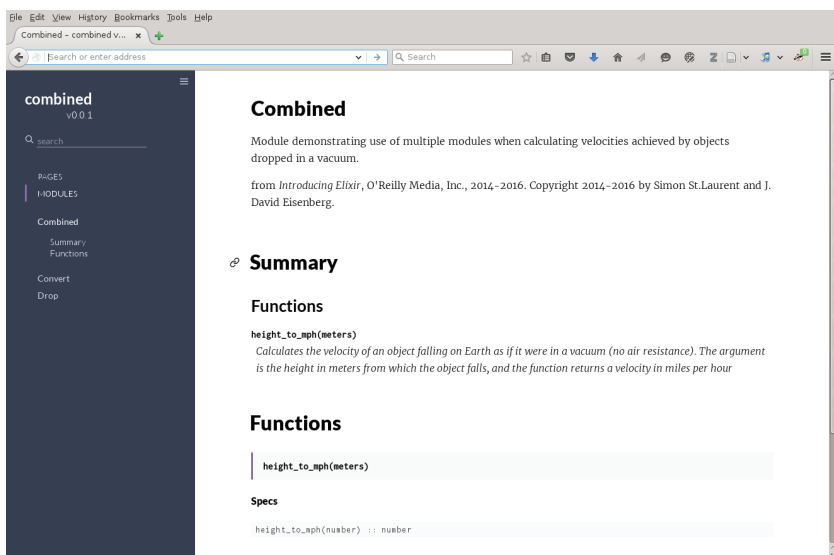


Рис. В.1 ❖ Пример веб-страницы с документацией, сгенерированной с помощью ExDoc

Предметный указатель

Символы

\, 80, 82
dbg, модуль, 159
debug, уровень журналирования, 156
error, уровень журналирования, 156, 240
ets.delete/1, функция, 183
ets.delete, функция, 190
ets.first, функция, 190
ets.fun2ms, функция, 190
ets.give_away/3, функция, 190
ets.info/1, функция, 182
ets.insert/2, функция, 184
ets.i(), функция, 186
ets.last, функция, 190
ets.lookup/2, функция, 187
ets.match, функция, 190
ets.new/2, функция, 181, 184
ets.next, функция, 190
ets.select, функция, 190
ets.tab2list/1, функция, 185
ignore, ответ, 241
infinity, атом, 203
info, уровень журналирования, 156
keypos, кортеж, 182
noop (no operation – пустая операция), 36
noreply, ответ, 203, 241
observer.start(), функция, 186
ok, ответ, 36, 74, 240
one_for_all, функция, 206
one_for_one, функция, 206
private, уровень доступа, 181
protected, уровень доступа, 181
public, уровень доступа, 181
reply, ответ, 241
stop, ответ, 203, 241
terminate/2, метод, 203
undefined, ответ, 240
warn, уровень журналирования, 156

@doc, тег, 45
-export, директива, 40
_from, аргумент, 203
-import, директива, 40
__MODULE__, объявление, 201
@opaque, 166
@spec, 165
@spec, тег, 45
@typep/@type, 166
@vsns, тег, 46
| (вертикальная черта), 93, 109
- (вычитание), оператор, 24
? (знак вопроса), 125
[] (квадратные скобки), 91
> (конвейер) оператор, 40, 90
<< нотация, 158
/ (обратный слеш), оператор, 24
' (одиночные кавычки), 83
-> оператор, 35, 63
#{ } оператор, 80
%{ } оператор, 108
++ оператор, 83, 92
<> оператор, 80, 83
& (оператор захвата), 33, 38, 119
<= оператор сравнения, 74
== оператор сравнения, 74, 81
=== оператор сравнения, 74, 81
>= оператор сравнения, 74
_ (подчеркивание), 55, 63
+ (сложение), оператор, 24
~ (тильда), 84
\; (точка с запятой), 30
* (умножение), оператор, 24
{ } (фигурные скобки), 80, 110
(хештег), знак, 44

А

Аккумуляторы, 73, 126
Анализ, статический, 161
Анонимные функции, 33

Аргументы

- значения по умолчанию, 42
- игнорирование с помощью подчеркивания, 55
- точная обработка с ограничителями, 52

Арифметические операторы, 242**Ассоциативные массивы, 106****Атомы, 48**

- и подчеркивание (`_`), 55
- и сопоставление с образцом, 49
- и строки, 79
- как ключи в отображениях, 108
- логические, 50
- синтаксис, 48

Б**Бесконечный цикл, 71****В****Вещественные числа, 24, 26****Взаимодействие с человеком, 79, 91, 103, 118**

- Ask, модуль, 85
- ввод символов, 85
- ввод строк, 86

Времени выполнения, ошибки, 153**Встроенные документы (heredocs), 82****Высшего порядка, функции**

- определение, 118
- основы, 118
- создание новых списков, 121

Г**Генераторы списков, 121, 123****Головы, извлечение из списков, 93****Д****Двоичные операторы, 242****Документация**

- встроенные тесты, 172
- комментарии в коде, 44
- модулей, 46
- создание с помощью ExDoc, 247
- типы данных, 246
- функций, 44

З**Замыкания, 120****Записи**

- достоинства и недостатки, 173
- использование в функциях, 177
- создание и чтение, 175

Запросы, 187**Зарезервированные слова, 240****Захвата оператор (&), 33, 38****Захвата, оператор (&), 119****И****Идентификатор процесса (pid), 129****Импортирование функций, 41****Исключения, ошибки и отладка**

- виды ошибок, 152
- восстановление работоспособности, 153
- журналирование результатов выполнения и ошибок, 156
- трассировка вызовов функций, 159
- трассировка сообщений, 157

Истинность, 66**К****Каталоги**

- определение текущего, 36
- смена, 36

Ключи, списки, 103, 107, 109**Команды интерактивной оболочки, 239****Комментарии в коде, 44****конвейер (`|>`) оператор, 90****конкатенация**

- списков, 92
- списков символов, 83

кортежи

- `:keypos`, 182
- в макросах, 214
- и списки, 92
- и структуры, 109
- смешивание списков и кортежей, 99
- с несколькими ключами, списки, 105

Л**Легковесные процессы, 135****Логика**

- if или else, 66
- обзор, 62

побочные эффекты, 69
присваивание переменным в case
и if, 68

Логика и рекурсия, 62

Логические (булевы) операторы, 241

Логические ошибки, 152, 161

М

Макросы, 213

и функции, 213, 219

недостатки, 219

программное создание функций, 217

простой пример, 214

создание множества функций, 218

создание новой логики, 216

условия и параметры, 217

Математические функции, 244

Метки, 84

Множества (:set), 180

Модули

деление кода на модули, 38

документирование, 46

импортирование функций, 41

определение, 34

порождение процессов из, 132

Модульные тесты, 167

настройка, 170

тесты, встроенные

в документацию, 172

Мультимножества (:bag), 180

Н

Неправильные списки, 97

О

Обратный отсчет, 71

Ограничители, 52, 243

в конструкции case, 64

в логике обратного отсчета, 72

и побочные эффекты, 71

Оператор захвата (&), 33, 38

Операторы, 241

арифметические, 242

двоичные, 242

логические атомы, 50

логические (булевы), 241

приоритет, 243

сравнения, 241

Операторы сравнения, 74, 81

Определение модулей, 34

Открытая телекоммуникационная
платформа (Open Telecom Platform,
ОТР), 198

видеовведение, 199

преимущества, 198

простой супервизор, 204

создание служб с GenServer, 199

Отображения и списки, 107

Ошибки времени

выполнения, 153, 161

П

Пары имя/значение, 103

от отображений к структурам, 109

добавление поведения

в структуры, 114

использование структур

в сопоставлениях, 111

использование структур

в функциях, 112

объявление структур, 110

отображения и структуры, 109

расширение существующих

протоколов, 116

создание структур, 110

чтение структур, 110

от списков к отображениям, 107

изменение отображений, 108

отображения и списки, 107

создание отображений, 108

чтение отображений, 109

словари, 106

списки ключей, 103

списки кортежей с несколькими
ключами, 105

Паскаля, треугольник, 99

Переменные, присваивание в case
и if, 68

Побочные эффекты, 69

Повторяющиеся мультимножества
(:duplicate_bag), 180

Подчеркивание (_, 55

«Позволь потерпеть аварию»,
философия, 95, 138

Полные квалифицированные
имена, 176

Предикаты, 125

Приоритет операторов, 243

Программы, развертывание, 129

Протоколы, 114

Процессы

аварийное завершение, 145

взаимодействие между

процессами, 138

дополнительные средства

управления, 151

идентификатор процесса (pid), 129

и таблицы, 188

когда останавливаются, 137

легковесные, 135

наблюдение, 141

наблюдение за движением

сообщений между, 143

определение, 129

порождение процессов

из модулей, 132

регистрация, 136

связи между, 146

Прямой отсчет, 73

Р

Рекурсия, 71

бесконечная, 71

обратный отсчет, 71

простая, 71

прямой отсчет, 73

с возвратом значения, 74

хвостовая, 77

С

Связи

двунаправленные, 147

установка между процессами, 146

Семантические ошибки, 161

Символы, ввод пользователя, 85

Синтаксические ошибки, 161

Система времени выполнения Erlang (Erlang Runtime System, ERTS), 37

Система управления базами

данных, 191

Словари, 106

Соглашения об именовании, 125

Сообщения

вывод/удаление, 131

и идентификатор процесса, 131

наблюдение за движением

между процессами, 143

синтаксис, 131

трассировка, 157

чтение, 131

Сопоставление с образцом

внутри функций, 62

и атомы, 49

и структуры, 111

Спецификации типов, 164

Списки

деление головы и хвосты, 93

добавление в конец, 92

допустимое количество

элементов, 91

допустимые элементы, 91

и кортежи, 92

и отображения, 107

ключей, 103, 107, 109

конкатенация, 92

кортежей с несколькими

ключами, 105

неправильные, 97

обработка содержимого, 94

обработка с помощью функций, 122

определение, 91

получение информации о, 121

проверка условий, 124

пустые, 95

разбиение, 125

свертка, 126

синтаксис, 91

смешивание списков и кортежей, 99

создание из головы и хвоста, 96

создание одного списка

из нескольких, 92

создание списка списков, 99

создание с помощью функций

высшего порядка, 121

списков, 92

фильтрация, 123

Списки символов, 83

конкатенация, 83

Справка, получение, 22

Сравнения операторы, 241

Статический анализ, 161

Строки

- UTF-8, 82
- ввод пользователя, 86
- документирования, 82
- и атомы, 79
- интерполяция, 80
- метки, 84
- многострочные, 82
- обработка в Elixir, 79
- #{} оператор, 80
- <> оператор, 80
- операторы сравнения, 81
- создание новых, 80
- списки символов, поддержка, 83
- экранированные
- последовательности в, 80

Строковые метки, 246**Супервизор, 204****Т****Таблицы**

- ETS
 - и процессы, 188
 - создание и заполнение таблиц, 181
 - удаление, 190
- Mnesia, создание, 192

Текстовые данные, чтение, 86**Типы, спецификации, 164****Типы данных, для использования в документации и анализа, 246****У****Упорядоченные множества (:ordered_set), 180****Ф****Факториалы, 74****Функции**

- анонимные, 33
- аргументы со значениями по умолчанию, 42
- включение нескольких инструкций, 32
- вызов, 32
- высшего порядка, 118, 245
- документирование, 44
- и конструкция cond, 65
- и макросы, 219
- импортирование, 41

использование записей, 177**использование структур в, 112****логика внутри, 62****макросы, 213****математически, 244****отладка, 159****передача функций из модулей, 121****рекурсия, 71****создание программно, 217****создание с ключевым словом****fn, 31, 118****ссылка на уже определенную****функцию, 38****трассировка вызовов, 159****экранированные****последовательности в строках, 80****Х****Хвостовая рекурсия, 77****Хвосты, извлечение из списков, 93****Хранение данных****Mnesia, 191****в Erlang Term Storage, 179****Хранилище данных, 173****записи, 173****Хранилище термов Erlang (ETS)****виды коллекций, 180****запросы, 187****изменение значений, 187****преимущества, 179****создание и заполнение таблиц, 181****таблицы и процессы, 188****храняемая таблица поиска (Persistent Lookup Table, PLT), 162****Ц****Целые числа, 26****Цикл, бесконечный, 71****Э****Экранированные****последовательности, 80, 245****Ю****Юникод (UTF-8) строки, 82****А****after, конструкция, 134****Ask, модуль, 85**

В

BEAM (Bogdan's Erlang Abstract Machine – абстрактная машина Богдана для Erlang), 37, 162

С

CaseClauseError, 64, 155
 case...end, инструкция, 132
 case, конструкция, 62, 88
 cd(directory), команда, 239
 cd(pathname), команда, 36
 cd(), команда, 24
 c(file, path), команда, 239
 c(file), команда, 239
 clear(), команда, 239
 cond, конструкция, 62, 65
 countdown(), функция, 71
 countup(), функция, 73
 c (компиляция), команда, 37

D

defmacro, объявление, 213
 defmodule, объявление, 34, 110, 201
 defrecord, объявление, 174
 defstruct, объявление, 110
 Dialyxr, инструмент, 162
 Dialyzer (DIscrepancy AnaLYzer for Erlang – анализатор несоответствий для программ на Erlang)
 и спецификации типов, 164
 статический анализ, 161
 divide, функция, 127
 div(), функция, 25
 do...end, синтаксис, 35

E

EEx (Embedded Elixir – встроенный Elixir), 229
 Elixir
 атомы, кортежи и сопоставление с образцом, 48
 базовые операции, 22
 взаимодействие с человеком, 79, 91, 103
 запуск, 21
 и Erlang, 69, 83
 исключения, ошибки и отладка, 152
 и фреймворк Phoenix, 221

команды интерактивной оболочки, 239
 макросы, 213
 пары имя/значение, 103
 переменные, 29
 преимущества, 237
 процессы, 129
 словарь программиста, 239
 списки, 91
 статический анализ, спецификации типов и тестирование, 161, 173
 установка, 20
 функции высшего порядка и генераторы списков, 118
 функции и модули, 31
 хранилище данных, 173
 числа, 26
 elixirc, команда, 38
 ElixirOTP (Open Telecom Platform – открытая телекоммуникационная платформа), 198
 Emacs, текстовый редактор, 23
 end, ключевое слово, 32
 Enum, модуль
 Enum.all?/2, 124
 Enum.any?/2, 124
 Enum.concat/1, функция, 93
 Enum.drop_while?/2, 125
 Enum.each/2, 121
 Enum.filter/2, 123
 Enum.map/2, 122
 Enum.partition/2, 123
 Enum.reverse, функция, 94
 Enum.take_while?/2, 125
 Enum.unzip/1, функция, 99
 Enum.zip/2, функция, 99
 функции высшего порядка, 121
 Erlang
 Observer, инструмент, 141, 185
 и Elixir, 69, 83
 руководство «Erlang User Guide», 239
 совместимость со старым кодом, 103
 файлы BEAM, 37, 162
 Erlang, установка, 19
 ERTS (Erlang Runtime System – система времени выполнения Erlang), 37

excerpt, аргумент, 42
 ExDoc, инструмент, 47, 246, 247
 ExUnit, модуль, 167

F

false, значение, 66
 flush(), функция, 131, 240
 fn, ключевое слово, 31, 118
 f_test, функция, 214
 FunctionClauseError, 64

G

GenServer (generic server – обобщенный сервер), 199

H

handle_call/3, функция, 202
 handle_cast/2, функция, 202
 handle_info/2, функция, 203
 hd/1, функция, 187
 heredocs (встроенные документы), 82
 Hex, диспетчер пакетов, 248
 h(item), команда, 240
 h(), команда, 240

I

IEEx, интерактивная оболочка, 21
 вызов функций, 25
 достоинства, 22
 как калькулятор, 24
 навигация по тексту и командам, 23
 навигация по файлам, 23
 переменные, 28
 if или else, инструкции, 66
 if, конструкция, 62, 72, 216
 import, команда, 41
 init/1, функция, 204
 input, переменная, 89
 insert_into_table/1, функция, 190
 Inspect.Algebra, модуль, 117
 inspect, функция, 81, 116
 IO.getn, функция, 85
 IO.gets, функция, 87
 IO.puts, функция, 69, 75, 80, 121, 156
 и символ перевода строки, 70
 IO.write/1, функция, 139
 is_, предикаты, 125

K

Keyword, модуль, 104

L

line, функция, 87
 List, модуль
 List.foldl/3, 126
 List.foldr/3, 126
 l(module), команда, 240
 Logger, модуль, 156
 ls(path), команда, 239
 ls(), команда, 239

M

math, модуль, 26
 math, модуль (Erlang), 121, 181
 Mix, инструмент
 работа с модулями, 34
 упаковка приложения, 209
 Mnesia
 и фреймворк Phoenix, 224
 настройка, 191
 преимущества, 191
 создание таблиц, 192
 чтение данных, 196
 m(), команда, 240

N

nil, значение, 66

O

Observer (Erlang), инструмент, 141
 Observer, инструмент (Erlang), 185
 only, аргумент, 42
 OTP (Open Telecom Platform – открытая телекоммуникационная платформа), 198
 видеовведение, 199
 преимущества, 198
 простой супервизор, 204
 создание служб с GenServer, 199

P

PATH, переменная, 20
 Phoenix, фреймворк
 вычисления, 230
 достоинства, 221
 начало, 224

онлайн-документация, 237
представление страницы, 224
 маршрутизация, 225
 простое представление, 228
 простой контроллер, 227
ресурсы для изучения, 237
структура простого
 приложения, 224
 установка базовых
 компонентов, 221

Process.register/2, функция, 136
Process.registered/0, функция, 137
Process.whereis/1, функция, 137
product/1, функция, 95
product/2, функция, 95
pwd(), команда, 36, 239

Q

quote, функция, 215

R

receive...end, конструкция, 131, 132
recompile, команда, 35, 50
record_info/2, функция, 193
Regex (регулярные выражения), 82
rem(), функция, 25
report/0, функция, 133
r(module), команда, 240
r(), команда, 50

S

self(), функция, 129
send/2, функция, 130
spawn/3, функция, 132
start..end, нотация, 122
start_link/0, функция, 201
start_link, функция, 204
State, структура, объявление, 201
supervise/2, функция, 206
s(), функция, 46

T

trace_pattern, функция, 157
trace, функция, 157
try...catch, конструкция, 184
try...rescue, конструкция, 153

U

unless, конструкция, 216
unquote/1, функция, 215

V

v(n), команда, 240
v(), команда, 240

W

when, ключевое слово, 52
worker/3, функция, 206
wxwidgets, библиотека, 141

Об авторах

Симон Сенлорен (Simon St. Laurent), администратор контента на сайте LinkedIn Learning, в основном занимающийся клиентской частью. В прошлом сопредседатель конференций Fluent и OSCON. Автор и соавтор нескольких книг, включая «Introducing Elixir», «Introducing Erlang», «Learning Rails 3» и «XML Pocket Reference. 3rd Edition», «XML: A Primer, and Cookies».

Другие его работы о программировании, квакерстве и городе Драйдене (Канада) можно найти на сайте <http://simonstl.com/>.

Дэвид Эйзенберг (J. David Eisenberg), программист и преподаватель. Живет в городе Сан-Хосе, штат Калифорния. Дэвид – талантливый преподаватель и лектор. Он разработал курсы изучения HTML, CSS, JavaScript и Perl. Читает курс «Компьютерные информационные технологии» в колледже Evergreen Valley в городе Сан-Хосе и разработал несколько сетевых вводных курсов для самообучения, которые были переведены на корейский, греческий и русский языки. Занимается разработкой обучающего программного обеспечения с 1975 года, когда принял участие в проекте «Modern Foreign Language» в университете штата Иллинойс, занявшись разработкой технологий компьютерного обучения в системе PLATO. Является соавтором книги «SVG Essentials» (O'Reilly). Помимо программирования Дэвид также увлекается цифровой фотографией, ухаживает за бродячими кошками на работе и любит погонять на мотоцикле.

Колофон

Животное на обложке книги «Введение в Elixir» – это четырехрогая антилопа (*Tetracerus quadricornis*), обитающая в Индии и Непале. Эта антилопа, которую также называют *чужинга*, – самая маленькая из азиатских полорогих. Она достигает высоты 55–60 сантиметров в холке и веса 20–25 килограммов. Имеет стройное тело, тонкие ноги и короткий хвост. Окрас желто-коричневый или слегка красноватый, сменяющийся белым внизу живота и вдоль внутренней поверхности ног. Вдоль каждой ноги также проходит черная полоса. Отличительной особенностью антилопы является наличие четырех рогов у самцов: два между ушей, которые начинают расти с возраста всего в несколько месяцев, и два на лбу, рост которых начинается в возрасте 10–14 месяцев. Передняя пара рогов может вырастать до 5, а задняя – до 10 сантиметров.

Четырехрогие антилопы предпочитают селиться близ воды, в районах с обильным растительным покровом, например с высокой травой или густым подлеском. Это, как правило, одиночное животное, хотя изредка они могут образовывать группы до четырех особей, и избегают мест обитания человека. В брачный период – с мая по июль – самцы могут вести себя агрессивно по отношению к другим самцам. Беременность у самок длится около восьми месяцев, и обычно рождается один-два детеныша, которые остаются со своими матерями до достижения годовалого возраста. Половая зрелость наступает в возрасте двух лет.

Антилопы общаются между собой, издавая предупредительные сигналы и помечая растения в области обитания запахом секрета из глазных желез (а также оставляя кучки экскрементов).

Из-за того, что ареал их обитания находится в густонаселенном районе мира, естественная среда обитания четырехрогих антилоп постоянно сокращается из-за расширения человеком сельскохозяйственных угодий. Этот вид признан исчезающим Международным союзом охраны природы и природных ресурсов из-за сокращения среды обитания. Они также являются мишенью для охотников за необычным четырехрогим черепом. По оценкам экспертов в дикой природе осталось не более 10 000 особей этого вида; многие живут на защищенных территориях заповедников. Четырехрогая антилопа охраняется законом о защите дикой природы Индии.

Многие животные, изображенные на обложках книг издательства O'Reilly, находятся на грани исчезновения. Все они важны для нашего мира. Чтобы узнать, как помочь их сохранению, посетите страницу animals.oreilly.com.

Изображение для обложки взято из «Wood's Animate Creation». Для надписей на обложке использованы шрифты URW Typewriter и Guardian Sans.

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: 115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес. Эти книги вы можете заказать и в интернет-магазине: **www.aliants-kniga.ru**.

Оптовые закупки: тел. **(499) 782-38-89**.

Электронный адрес: **books@aliants-kniga.ru**.

Симон Сенлорен и Дэвид Эйзенберг

Введение в Elixir

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Перевод *Киселев А. Н.*

Корректор *Синяева Г. И.*

Верстка *Чаннова А. А.*

Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16.

Гарнитура «Петербург». Печать офсетная.

Усл. печ. л. 13.125. Тираж 200 экз.

Веб-сайт издательства: **www.дмк.рф**