Elixir в действии

Саша Юрич

При разработке программного обеспечения для решения критически важных задач огромную роль играет отказоустойчивость. Язык программирования Elixir позволяет создавать быстрые и надежные приложения, будь то крупная распределенная система, набор сервисов для бэкенда или простенькое веб-приложение. Элегантный синтаксис Elixir и функциональный образ мышления позволяют достичь простоты в написании, чтении и поддержке кода.

Книга «Elixir в действии» научит вас создавать готовые к промышленной эксплуатации распределенные приложения на языке Elixir. Автор Саша Юрич познакомит вас с этим мощным языком на примерах, подчеркивающих преимущества функционального и конкурентного программирования на Elixir. Вы узнаете, как при помощи фреймворка OTP освободиться от решения большого количества однообразных низкоуровневых задач. Также вы изучите эффективные подходы к реализации конкурентного выполнения по ходу превращения работающей системы в распределенную систему на нескольких компьютерах.

«Четкий, грамотный, увлекательно написанный учебник по Elixir, полный практических советов».

Джордж Томас, компания Manhattan Associates

«Лучшее введение в Elixir и Erlang/OTP. Обширность и глубина знаний впечатляют».

Иоган О'Доннелл, компания Trellis Energy

«С этой книгой от одного из ведущих специалистов по платформе Erlang можно открывать для себя Elixir/OTP снова и снова».

Мафинар Хан, компания Tread

Краткое содержание:

- Elixir версии 1.7;
- функциональное и конкурентное программирование;
- создание готовых к развертыванию релизов;
- устройство распределенных систем.

Вам понадобятся навыки в разработке клиент-серверных приложений и знание языков Java, C# или Ruby. Опыт разработки на Elixir не требуется.

Саша Юрич — разработчик с большим опытом в создании сложных систем на стороне сервера на Elixir и Erlang.

ISBN 978-5-97060-773-2

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК "Галактика"
books@alians-kniga.ru

www.дмк.рф

Elixir в действии

Elixir в действии

MANNING

Саша Юрич

Elixir в действии

Elixir in Action

SECOND EDITION

SAŠA JURIC

MANNING
Shelter Island

ЛАНЬ®

Elixir в действии

САША ЮРИЧ

Москва, 2020

УДК 004.43Elixir
ББК 32.972
Ю70

Юрич С.
Ю70 Elixir в действии / пер. с анг. Н. А. Нестеровой. – М.: ДМК Пресс, 2020. – 376 с.: ил.

ISBN 978-5-97060-773-2

В книге рассматриваются базовые характеристики и примеры использования языка Elixir, предназначенного для создания масштабируемых, распределенных и отказоустойчивых систем, работающих на основе виртуальной машины Erlang. Сегодня ее используют в самых разных областях: для создания инструментов совместной работы, систем открытых торгов в режиме реального времени, серверов баз данных, многопользовательских онлайн-игр и др.

Приведенные в книге примеры подчеркивают преимущества функционального и конкурентного программирования на языке Elixir, которые могут обеспечить бесперебойную работу систем, обслуживающих огромное количество пользователей со всего мира.

Издание предназначено для профессиональных разработчиков, имеющих опыт программирования на Java, C#, Ruby, C++ или другом языке общего назначения.

УДК 004.43Elixir
ББК 32.972

ISBN 978-1-617-29502-7 (анг.)
ISBN 978-5-97060-773-2 (рус.)

Содержание

Отзывы о первом издании

Увлекательно и познавательно… море практических советов.
– Вед Антани, компания Electronic Arts

Прекрасно показано на реальных примерах, на что способен Elixir по части распределенных вычислений.
– Кристофер Бэйли, компания HotelTonight

Если вы хотите научиться думать и решать проблемы как настоящий эликсирщик, эта книга для вас!
– Космас Чатзимикалис

Функциональное программирование стало понятнее.
– Мохсен Мостафа Джокар, газета Hamshari

Возможно, лучшее введение в Elixir и функциональное программирование.
– Покупатель интернет-магазина Amazon

Отличная книга для опытных разработчиков, желающих познакомиться с Elixir поближе.
– Покупатель интернет-магазина Amazon

Предисловие

В 2010 году передо мной стояла задача реализации системы для передачи систематических обновлений нескольким тысячам пользователей в близком к реальному масштабе времени. В моей компании в основном использовали Ruby on Rails, но мне нужно было что-то, более подходящее для такой задачи с высокой степенью конкурентности. Последовав совету технического директора, я обратился к языку Erlang, изучил литературу о нем, сделал прототип и провел нагрузочное тестирование. Я был впечатлен полученными результатами и решил реализовать уже реальный проект на Erlang. Пару месяцев спустя система была готова и с тех пор прекрасно работает.

Со временем я стал всё больше ощущать ценность языка и то, как он помог мне организовать управление такой сложной системой, а постепенно и вовсе предпочел его используемым ранее технологиям. Я начал знакомить людей с языком сначала внутри компании, а потом на местных мероприятиях. В итоге в 2012 году я стал вести блог «The Erlangist» (**http://theerlangelist.com**), где стараюсь продемонстрировать приверженцам ООП все преимущества Erlang.

Поскольку Erlang – особенный язык, я решил попробовать Elixir в надежде, что он поможет мне показать всю красоту Erlang более понятным для ООП-программистов способом. Несмотря на то что Elixir тогда был еще совсем молод (версия 0.8), я был просто поражен его зрелостью и легкой интеграцией с Erlang. Вскоре я начал разрабатывать на Elixir новые функции для своей системы на основе Erlang.

Спустя еще несколько месяцев на меня вышел Майкл Стивенс (Michael Stephens) из издательства Manning и поинтересовался, не хотел бы я написать книгу об Elixir. На тот момент о нем уже готовились две книги, и я подумал, что мог бы добавить к ним ещё одну, где язык рассматривался бы с другого ракурса: акцентируя внимание на конкурентной модели Elixir и философии OTP. Работать над книгой было непросто, но это того стоило.

По прошествии двух лет с момента публикации первого издания я согласился работать над вторым. По сути, это то же первое издание, приведенное в соответствие с последними обновлениями Elixir и Erlang. Наиболее важные изменения коснулись глав 8, 9 и 10 – значительные их части были переписаны, и теперь они включают новые методы работы с супервизорами и реестрами процессов.

Книга «Elixir в действии» содержит актуальную информацию и поможет вам изучить новейшие приемы разработки программного обеспечения на Elixir. Надеюсь, вам понравится моя книга, вы сможете многому научиться и применить свои знания на практике!

Благодарности

Прежде всего мне хотелось бы поблагодарить мою жену Ренату за нескончаемое терпение и поддержку в то продолжительное время, когда я днями и ночами работал над книгой.

Благодарю издательство Manning за публикацию книги. В частности, Майкла Стивенса (Michael Stephens) за то, что вышел со мной на связь, Марьян Бэйс (Marjan Bace) за предоставленную возможность написать эту книгу, Берта Бэйтса (Bert Bates) за то, что задал мне верное направление, Карен Миллер (Karen Miller) за то, что помогала не сбиться с пути, Александра Драгосавльевича (Aleksandar Dragosavljevic) за вычитку текста, Кевина Салливана (Kevin Sullivan) и Винсента Нордхауса (Vincent Nordhaus) за подготовку книги к публикации, Тиффани Тэйлор (Tiffany Taylor) и Энди Кэррола (Andy Carroll) за преобразование моего разговорного языка в литературный, а также Кэндис Гиллхули (Candace Gillhoolley), Ану Ромак (Ana Romac) и Кристофера Кауфманна (Christopher Kaufmann) за продвижение книги.

Качество содержимого данной книги удалось значительно повысить благодаря отзывам рецензентов и первых читателей. В первую очередь я хотел бы сказать спасибо Эндрю Джибсону (Andrew Gibson), давшему ценные комментарии и помогшему мне преодолеть последний рубеж. Также благодарю Алексея Шолика (Alexei Sholik) и Питера Минтена (Peter Minten) за своевременную помощь по технической части во время написания книги.

Выражаю огромную благодарность Ризе Фахми (Riza Fahmi) и всем остальным техническим редакторам: Элу Рахими (Al Rahimi), Алану Лентону (Alan Lenton), Алексею Галиуллину (Alexey Galiullin), Эндрю Кортеру (Andrew Courter), Аруну Кумару (Arun Kumar), Асхаду Дину (Ashad Dean), Кристоферу Бэйли (Christopher Bailey), Кристоферу Хаупту (Christopher Haupt), Кливу Харберу (Clive Harber), Даниэлю Куперу (Daniel Couper), Йогану О'Доннелу (Eoghan O'Donnell), Фредерику Шиллеру (Frederick Schiller), Габору Ласло Хашбе (Gábor László Hajba), Джорджу Томасу (George Thomas), Хизер Кэмпбелл (Heather Campbell), Джерону Бенкхушсену (Jeroen Benckhuijsen), Хорхе Дефлону (Jorge Deflon), Хосе Валиму (José Valim), Космасу Чатсимихалису (Kosmas Chatzimichalis), Мафинару Хану (Mafinar Khan), Марку Райалу (Mark Ryall), Матиасу Полигкайту (Mathias Polligkeit), Мохсену Мустафе Джокару (Mohsen Mostafa Jokar), Тому Гейденсу (Tom Geudens), Томеру Эльмалему (Tomer Elmalem), Веду Антани (Ved Antani) и Юрию Бодареву (Yurii Bodarev).

Я также хотел бы поблагодарить всех читателей – участников программы раннего доступа издательства «Маннинг» (Manning Early Access Program, MEAP), предоставивших справедливые замечания. Спасибо, что потратили время на прочтение моей писанины и оставили такие полезные отзывы.

Особого упоминания заслуживают люди, подарившие нам Elixir и Erlang, а именно сами создатели, участники их команды и помощники. Спасибо вам за эти замечательные продукты, благодаря которым мне стало проще и интереснее работать с кодом. И наконец, отдельная благодарность всем членам сообщества Elixir! Это самое прекрасное и дружелюбное сообщество программистов из всех существующих!

О книге

Elixir – современный функциональный язык программирования, предназначенный для создания масштабируемых, распределенных и отказоустойчивых систем, работающих на основе виртуальной машины Erlang. Этот язык привлекателен сам по себе, но взаимодействие с платформой разработки Erlang – несомненно, его огромное преимущество.

Платформа Erlang была создана как средство обеспечения высокой доступности. Изначально она была предназначена для разработки телекоммуникационных систем, но сегодня ее используют для создания инструментов совместной работы, систем открытых торгов в режиме реального времени, серверов баз данных, многопользовательских онлайн-игр и еще во многих других областях. Система, обслуживающая огромное количество пользователей со всего мира, должна работать непрерывно без ощутимых задержек, независимо от ошибок и проблем с аппаратными средствами, возникающих во время ее эксплуатации. Ни одному конечному пользователю не понравится испытывать частые и продолжительные перебои. Такая система ненадежна и непригодна для использования, а значит, не выполняет свою главную функцию. Высокая доступность – крайне важное свойство системы, и Erlang помогает его достичь.

Elixir призван улучшить и модернизировать разработку систем на основе Erlang. Он объединяет в себе функциональные особенности таких языков программирования, как Erlang, Clojure и Ruby. В его стандартную поставку входят инструменты, упрощающие процессы управления проектом, тестирования, упаковки и создания документации. Можно сказать, что Elixir снижает порог вхождения в Erlang и увеличивает скорость разработки. Благодаря лежащей в основе среде выполнения Erlang при создании систем на Elixir вам доступны любые библиотеки экосистемы Erlang, включая проверенный временем фреймворк OTP.

Для кого эта книга

Эта книга содержит обучающий материал, изучив который, вы сможете создавать на Elixir готовые к промышленной эксплуатации системы. Вы не найдете здесь подробной информации о каждом аспекте языка или нюансе работы виртуальной машины Erlang. Точность вычислений с плавающей запятой, специфика Unicode, файловые операции ввода-вывода, модульное тестирование и многие другие темы рассмотрены лишь поверхностно или опущены. Все это очень важно, но не является главным фокусом данной книги. При необходимости вы можете изучить эту информацию самостоятельно, а в рамках данной книги сосредоточиться на решении более интересных и необычных задач, а именно на том, как с помощью конкурентного программирования можно сделать систему масштабируемой, отказоустойчивой, распределенной и высокодоступной.

Представленные в книге подходы и решения также рассмотрены не полностью. Некоторые нюансы были опущены для краткости и смещения акцента в сторону главной проблемы. Моей целью было не охватить как можно больше деталей,

а рассказать о лежащих в основе принципах и их объединении в общую картину. Прочитав эту книгу, вы легко сможете получить недостающие вам знания: для этого на протяжении всей книги вставлены все необходимые ссылки и упоминания.

Поскольку темы, рассматриваемые в книге, далеко не для начинающих, вы должны иметь в виду некоторые требования, предъявляемые к читателю. Во-первых, вы должны быть профессиональным разработчиком, имеющим пару лет опыта. Ваш стек технологий значения не имеет. Вы можете писать на Java, C#, Ruby, C++ или любом другом языке программирования общего назначения. Приветствуется опыт в разработке бэкенда (серверной стороны) систем.

Вам не обязательно знать что-либо об Erlang, Elixir или других платформах конкурентного программирования и не нужно разбираться в функциональном программировании. Если вы программируете на объектно-ориентированном языке, вначале изучение Elixir может вызвать трудности. Но как ООП-разработчик с большим стажем спешу вас заверить, что бояться тут нечего. Лежащие в основе Elixir принципы функционального программирования довольно легки в понимании. Конечно, функциональное программирование сильно отличается от привычной вам картины, к нему нужно просто привыкнуть. Но это гораздо проще, чем кажется, и опытному разработчику не составит труда разобраться во всех представленных в книге основах.

Структура книги

Книга состоит из трех частей.

Первая часть – введение в Elixir. В ней описываются основы языка и подробно рассматриваются самые часто используемые идиомы функционального программирования:

- в главе 1 приведен обзор технологий Elixir и Erlang, описаны области их применения и их отличия от других языков и платформ;
- в главе 2 описываются такие структурные единицы Elixir, как модули, функции и система типов;
- в главе 3 в подробностях рассматривается сопоставление с образцом и использование его для управления потоком выполнения;
- в главе 4 показывается создание абстракций более высокого уровня на основе иммутабельных структур данных.

Вторая часть строится на изученных в первой части основах. Основное внимание уделяется конкурентной модели Erlang и ее основным преимуществам – масштабируемости и отказоустойчивости:

- в главе 5 представлена конкурентная модель Erlang, а также основные примитивы конкурентных вычислений;
- в главе 6 рассматриваются обобщенные серверные процессы – главные структурные элементы для создания высококонкурентных систем на Elixir/Erlang;
- в главе 7 показывается процесс создания более сложной конкурентной системы;

- в главе 8 вы изучите подходы к обработке ошибок, в которых особое внимание уделяется конкурентности системы;
- глава 9 содержит подробную информацию об изоляции всех типов ошибок и ограничении их влияния на этапе промышленной эксплуатации;
- в главе 10 представлено несколько альтернатив обобщенным серверным процессам, более подходящих для определенного рода ситуаций.

В третьей части рассказывается о системах на этапе промышленной эксплуатации:

- в главе 11 рассматриваются OTP-приложения, необходимые для упаковки повторно используемых компонентов;
- в главе 12 описываются распределенные системы, призванные улучшить отказоустойчивость и масштабируемость;
- в главе 13 показаны различные способы подготовки Elixir-систем к промышленной эксплуатации, основное внимание уделено OTP-релизам.

О листингах с кодом

Исходный код в данной книге представлен

`моноширинным шрифтом,`

выделяющим его на фоне остального текста. Код многих листингов сокращен в целях обращения внимания на рассматриваемые приемы. Чтобы код мог уместиться в свободное место на странице, в нем используются переносы строк и абзацные отступы.

Все приведенные в книге примеры можно найти в репозитории GitHub по адресу: **https://github.com/sasa1977/elixir-in-action**. Вы также можете загрузить их в сжатом виде со страницы издательства по адресу: **www.manning.com/books/elixir-in-action-second-edition**.

Форум книги

При покупке данной книги вы получаете бесплатный доступ к закрытому веб-форуму от издательства Manning Publications (**www.manning.com/books/elixir-in-action-second-edition**), где можете оставить отзыв о книге, задать технические вопросы и получить помощь от авторов и других пользователей. На странице **https://forums.manning.com/forums/about** вы можете узнать больше о форумах Manning и ознакомиться с правилами поведения на них.

Издательство Manning считает своим обязательством предоставить такое пространство, в котором каждый читатель смог бы вести конструктивный диалог с авторами книг. Данные форумы не преследуют цели продвижения того или иного автора, и любая помощь осуществляется им исключительно на добровольной основе. Задавайте авторам свои каверзные вопросы, чтобы их интерес не угасал!

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте **www.dmkpress.com**, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте **www.dmkpress.com** или **www.дмк.рф** на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты **dmkpress@gmail.com** со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Об авторе

Саша Юрич – разработчик с большим опытом в создании объемных конкурентных систем на стороне сервера, десктопных приложений для Windows, а также приложений для терминалов. После 20 лет программирования на объектно-ориентированных языках он открыл для себя Erlang и Elixir. С помощью обеих технологий он разработал масштабируемый отказоустойчивый HTTP push-сервер и соответствующую серверную систему. На данный момент является членом команды Aircloak, где использует Erlang для создания автоматически конфигурируемого программного решения с обеспечением конфиденциальности данных. Он также ведет свой блог об Elixir и Erlang (**http://theerlangelist.com**).

Иллюстрация на обложке

Рисунок на обложке книги называется «Русская девушка». Иллюстрация взята из книги в четырех томах «Коллекция костюмов различных народов, античных и современных» (A Collection of the Dresses of Different Nations, Ancient and Modern) Томаса Джефферис (Thomas Jefferys), изданной в Лондоне между 1757 и 1772 годом. Как отмечается на титульной странице, это раскрашенные вручную гравюры на меди, обработанные гуммиарабиком. Томас Джефферис (1719–1771) носил звание «Географ короля Георга III». Он был английским картографом и крупнейшим поставщиком карт в своем времени. Он выгравировал и напечатал множество карт для правительства и других официальных органов, а также изготовил целый ряд коммерческих карт и атласов, в особенности Северной Америки. Будучи картографом, Джефферис проявлял интерес к одежде местных жителей, которую безупречно отразил в своей четырехтомной коллекции.

В конце XVIII века путешествия в дальние страны ради удовольствия были относительно новым явлением, и коллекции вроде этой пользовались популярностью, так как знакомили самих путешественников и любителей с культурой других народов. Разнообразие рисунков в книгах Джефферис позволяет увидеть, насколько уникальными и своеобразными были народности во всем мире 200 лет назад. С тех пор дресс-код поменялся, и нам уже не увидеть столь богатого разнообразия в одежде отдельных регионов и стран. В наше время даже сложно отличить друг от друга жителей разных континентов. С оптимистичной точки зрения, можно сказать, что мы променяли культурное и внешнее разнообразие на более насыщенную личную жизнь или более богатую и интересную интеллектуальную и техническую деятельность.

В наше время, когда крайне непросто отличить одну техническую книгу от другой, издательство Manning подчеркивает изобретательность и инициативу сферы компьютерных технологий вот такими обложками, основанными на богатом разнообразии региональной жизни двухвековой давности, претворяя иллюстрации Джефферис в жизнь.

Глава 1

Первые шаги

В главе рассматривается:

- платформа разработки Erlang;
- преимущества Elixir.

Здесь начинается ваше путешествие в мир Elixir и Erlang – двух эффективных и удобных технологий, призванных значительно упростить разработку больших масштабируемых систем. Скорее всего, вы читаете эту книгу в целях изучения Elixir. Но так как Elixir работает поверх платформы Erlang и в значительной степени зависит от нее, стоит для начала ознакомиться с Erlang и ее сильными сторонами. Предлагаю кратко пробежаться по основам Erlang.

1.1. Общие сведения об Erlang

Erlang – это платформа для разработки надежных масштабируемых систем с небольшим или нулевым временем простоя. Громкие слова, но Erlang существует именно для этих целей. Идея создания Erlang зародилась в середине 1980-х годов в шведском телекоммуникационном гиганте Ericsson и была обусловлена потребностью в удовлетворении нужд телекоммуникационных систем компании, в которых ключевую роль играли такие свойства, как надежность, быстрое реагирование, масштабируемость и постоянная доступность. Телекоммуникационные системы должны функционировать бесперебойно, независимо от количества одновременных вызовов, непредвиденных ошибок и производимых обновлений ПО и аппаратуры.

Исходя из истории, платформа Erlang должна была стать специализированной технологией для телекоммуникационных систем, но этого не произошло. Она не имеет в своем арсенале определенных средств для программирования телефонов, коммутаторов и прочих телекоммуникационных устройств. Напротив, Erlang – это платформа разработки общего назначения, предоставляющая необходимые инструменты для создания систем, обладающих такими нефункциональными свойствами, как конкурентность, масштабируемость, отказоустойчивость и высокая доступность.

В конце 80-х – начале 90-х годов, когда большинство приложений было десктопными, в высокой доступности нуждались лишь специализированные системы, в частности телекоммуникационные. На сегодняшний день ситуация изменилась: интернет и веб вышли на первый план, и приложения в большинстве своем ра-

ботают на основе серверных систем, которые обрабатывают запросы, выполняют операции с данными и передают соответствующую информацию большому количеству подключенных клиентов. Популярные сегодня системы – социальные сети, системы управления контентом, мультимедиа по запросу (Media on Demand, MoD) и многопользовательские компьютерные игры – больше рассчитаны на общение и взаимодействие пользователей.

Все вышеперечисленные системы имеют общие нефункциональные требования. Во-первых, система должна сохранять способность к реагированию независимо от количества подключенных клиентов. Во-вторых, непредвиденные ошибки не должны оказывать влияния на целую систему. Не так страшно, если из-за ошибки не выполнится единичный запрос, но если свернется вся система, это уже большая проблема. В идеале система никогда не должна отказывать, даже во время обновления ПО. Она должна всегда предоставлять услуги своим пользователям, работая бесперебойно.

Может показаться, что достичь таких целей довольно сложно, но при разработке систем, которыми люди пользуются каждый день, это просто необходимо. Если система не удовлетворяет требованиям по надежности и стабильности работы, значит, она не выполняет свою функцию. Поэтому системы, работающие на стороне сервера, обязательно должны находиться в постоянном доступе.

Именно для этого и предназначена Erlang. Высокая доступность обеспечивается благодаря таким техническим принципам, как масштабируемость, отказоустойчивость и распределение вычислений. В отличие от других современных платформ разработки, Erlang создавалась с основным фокусом на эти принципы. Команда разработчиков Ericsson под руководством Джо Армстронга (Joe Armstrong) несколько лет потратила на проектирование, прототипирование и тестирование своей платформы. В начале 90-х она имела ограниченную область применения, а сейчас она может быть полезна практически каждой системе.

В последнее время к Erlang проявляется большой интерес. Уже более 20 лет она обеспечивает работу различных объемных систем, а именно: приложения для обмена сообщениями WhatsApp, распределенной базы данных Riak, облачной платформы Heroku, системы автоматизированного развертывания Chef, очереди сообщений RabbitMQ, финансовых систем и бэкендов сетевых игр. Erlang – поистине проверенная временем и масштабом технология. В чем же секрет Erlang? Давайте же разберемся, как создавать надежные системы с высокой доступностью при помощи Erlang.

1.1.1. Высокая доступность

Erlang изначально была создана для разработки высокодоступных систем – онлайн-систем, способных предоставлять услуги своим пользователям даже при возникновении непредвиденных обстоятельств. На первый взгляд задача может показаться незамысловатой, но, как вы, возможно, догадываетесь, на этапе эксплуатации многое может пойти не по плану. Чтобы обеспечить бесперебойную работу системы, потребуется удовлетворить ряд технических требований:

- *отказоустойчивость*. Система не должна прерывать работу в случае возникновения непредусмотренных ошибок: отказа отдельных компонентов, разрыва сетевого соединения, выключения устройства, на котором она

запущена. В любом случае необходимо локализовать негативные последствия ошибки, насколько это возможно, устранить их и вернуть систему к жизни;

- *масштабируемость*. Система должна быть готова к нагрузкам любой сложности. Разумеется, не нужно судорожно скупать всю существующую технику, если все население планеты внезапно захочет воспользоваться вашей системой. Но стоит позаботиться о случаях повышенной нагрузки путем выделения дополнительных аппаратных ресурсов, не меняя ничего в коде. В идеале это должно осуществляться без перезагрузки системы.
- *распределенные вычисления*. Для создания непрерывно работающей системы необходимо использовать несколько компьютеров. Так вы сможете увеличить общую надежность системы: при отказе одного компьютера другой придет ему на замену. Кроме того, становится возможным горизонтальное масштабирование – добавив больше машин, вы сможете справиться с проблемой повышенной нагрузки;
- *способность к реагированию*. Понятно, что системе всегда необходимо быть достаточно быстрой и отзывчивой. Время обработки запросов не должно существенно увеличиваться даже при большой нагрузке на систему или возникновении непредвиденных ошибок. В частности, периодически возникающие длительные операции не должны оказывать существенного влияния на производительность системы или выводить ее из строя;
- *автоматическое обновление*. В некоторых случаях бывает необходимо запустить новую версию программы, не перезагружая при этом серверы. К примеру, в системе телефонной связи недопустимо во время обновления программы прервать текущие вызовы.

Если вышеперечисленные требования будут удовлетворены, ваша система будет высокодоступной, предоставляя услуги пользователям, несмотря ни на что.

Erlang вооружена всеми необходимыми для этого инструментами, ведь она была разработана для этих целей. При помощи конкурентной модели Erlang система неизбежно станет высокодоступной, обладая всеми присущими такой системе свойствами.

Давайте же разберемся, как реализуется конкурентность в Erlang.

1.1.2. Конкурентная модель Erlang

Конкурентность – основное оружие разработанных на Erlang систем. Практически каждая нетривиальная Erlang-система является системой с высокой степенью многопоточности, а сам язык даже иногда называют конкурентно-ориентированным. Вместо использования тяжеловесных потоков и процессов операционной системы (ОС) Erlang реализует конкурентность по своим собственным правилам, как показано на рис. 1.1.

Простейший элемент этой схемы называется *процессом Erlang* (не путать с процессом ОС или потоком). В типовых Erlang-системах выполняются тысячи или даже миллионы таких процессов. Виртуальная машина Erlang *BEAM* (*Bogdan/Björn's Erlang Abstract Machine*) имеет свой планировщик, который распределяет выполняемые процессы по доступным ядрам центрального процессора (ЦП). То, как реализованы процессы, дает множество преимуществ.

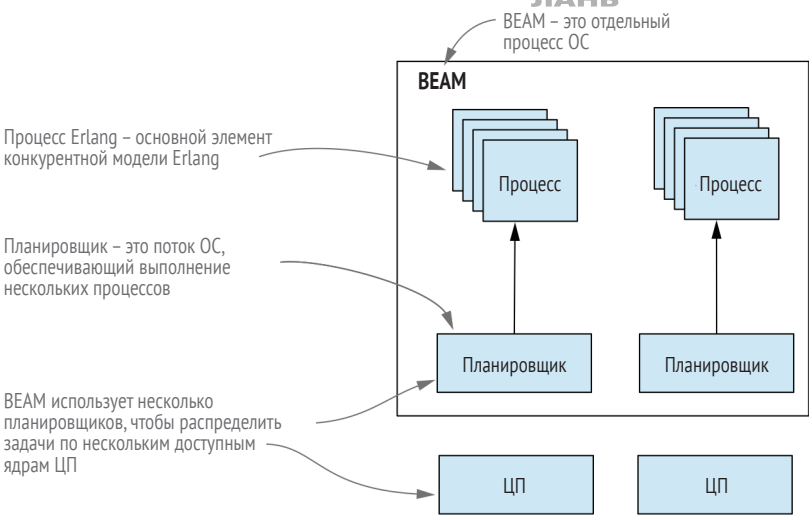


Рис. 1.1 ❖ Конкурентность в виртуальной машине Erlang

Отказоустойчивость

Процессы Erlang полностью изолированы друг от друга. У них нет общей памяти, а аварийное завершение одного процесса никак не сказывается на других. Именно это помогает ограничить последствия непредвиденной ошибки. Более того, Erlang предоставляет средства для обнаружения аварийного завершения процесса и последующего принятия решения. Обычно вместо старого процесса запускается новый.

Масштабируемость

Процессы, не имеющие общей памяти, взаимодействуют между собой посредством обмена асинхронными сообщениями. Это означает, что они обходятся без таких сложных механизмов синхронизации, как блокировки, мьютексы и семафоры. Соответственно, взаимодействие конкурентных объектов гораздо проще понять и реализовать.

Типовые Erlang-системы строятся на большом количестве конкурентных процессов, которые выполняют основную функцию системы (предоставляют услуги пользователям) путем взаимодействия между собой. Виртуальная машина по максимуму распараллеливает выполнение процессов, используя все доступные ядра процессора, что предоставляет Erlang-системам возможность масштабирования.

Распределенное выполнение

Взаимодействие между процессами происходит одинаково, независимо от того, где они находятся – в одном экземпляре BEAM или в двух разных экземплярах на двух разных удаленных компьютерах. Таким образом, написанная на Erlang система заведомо подготовлена к распределению вычислений между несколькими компьютерами, а это, в свою очередь, дает возможность горизонтального мас-

штабирования – использования группы компьютеров (кластера), разделяющих общую нагрузку системы. Помимо того, запуск программ на нескольких машинах делает систему поистине отказоустойчивой, ведь если одна машина выйдет из строя, другая сможет ее заменить.

Отзывчивость

Среда выполнения настроена таким образом, чтобы повысить общую отзывчивость системы. Как уже отмечалось ранее, в Erlang выполнение многочисленных процессов организовано специальными планировщиками. Реализуется вытесняющая многозадачность: планировщик предоставляет каждому процессу определенный временной промежуток, а по его истечении ставит его на паузу и запускает другой процесс. Поскольку промежуток совсем небольшой, процесс с долгим временем выполнения не останавливает всю систему. К тому же операции ввода-вывода выполняются в разных потоках, или используется служба опроса ядра ОС, если доступна. Это означает, что любой процесс, ожидающий выполнения операции ввода-вывода, не будет блокировать выполнение других процессов.

Даже процесс сбора мусора в Erlang направлен на улучшение отзывчивости системы. Напоминаю, процессы полностью изолированы и не имеют общей памяти. Сбор мусора осуществляется отдельно для каждого процесса. Это происходит намного быстрее, система не блокируется на долгое время и продолжает работать. В действительности в системах с многоядерными процессорами можно сделать так, чтобы одно ядро ЦП осуществляло быстрый сбор мусора, в то время как остальные ядра были бы заняты выполнением стандартных задач.

Как видите, конкурентность – ключевое понятие Erlang, и это не простой параллелизм. Благодаря своей реализации конкурентность обеспечивает отказоустойчивость, распределенное вычисление и отзывчивость системы. Типовые Erlang-системы выполняют множество конкурентных задач, используя тысячи или даже миллионы процессов. Особенно это может быть актуально при разработке систем на стороне сервера, которые в большинстве случаев можно полностью реализовать на Erlang.

1.1.3. Системы на стороне сервера

На Erlang разрабатывают различные системы и приложения. Существуют примеры десктопных приложений, написанных на Erlang, но в основном они используются во встраиваемых системах. На мой взгляд, больше всего платформа подходит для разработки систем на стороне сервера – систем, запускаемых на одном или более серверах и обслуживающих множество клиентов одновременно. Термин *система на стороне сервера* дает понять, что это нечто большее, чем простой обрабатывающий запросы сервер. Это целая система, которая вдобавок к обработке запросов должна выполнять различные фоновые задачи и управлять данными на сервере (server-wide), хранящимися в памяти, как показано на рис. 1.2.

Система на стороне сервера обычно функционирует на нескольких компьютерах, работающих совместно для решения единой бизнес-задачи. Чтобы обеспечить баланс нагрузки и добавить сценарии обработки отказа, можно разместить различные компоненты на разных машинах, а также развернуть их на разных серверах.

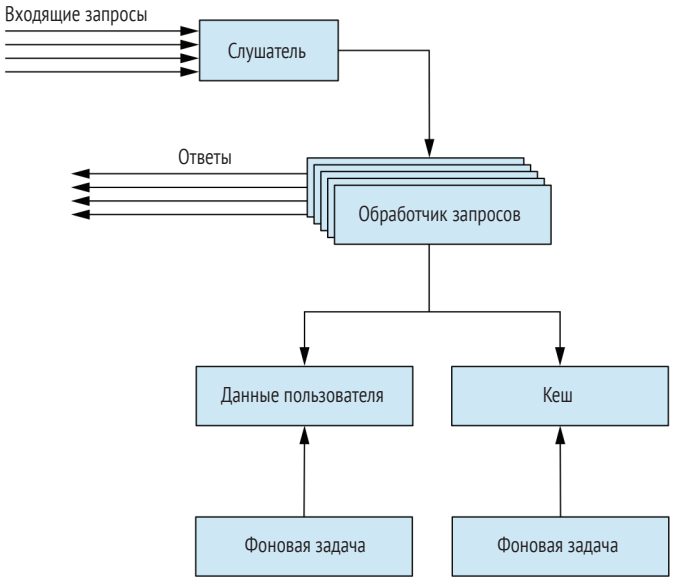


Рис. 1.2 ❖ Система на стороне сервера

Erlang может значительно упростить жизнь разработчику, ведь платформа предоставляет необходимые примитивы для написания конкурентных масштабируемых распределенных систем и позволяет реализовать всю систему с помощью только одной технологии. Каждый компонент на рис. 1.2 может быть реализован как Erlang-процесс, благодаря чему система становится масштабируемой и отказоустойчивой, а также в ней легко распределить вычисления. Используя примитивы Erlang для обнаружения ошибок и восстановления системы, вы сможете повысить надежность системы и устранить последствия непредвиденных ошибок.

Рассмотрим реальный пример. Я участвовал в разработке двух веб-серверов, имеющих похожие технические задачи: обслуживание большого количества пользователей, обработка запросов с длительным временем выполнения, управление глобальным (server-wide) состоянием, хранящимся в памяти, выполнение операций с данными, которые могут быть восстановлены после перезапуска процессов ОС и компьютера, и выполнение фоновых задач. В табл. 1.1 представлены использованные для разработки каждого сервера технологии.

Таблица 1.1. Технологии, используемые для разработки двух веб-серверов

Техническое требование	**Сервер А**	**Сервер Б**
HTTP-сервер	Nginx и Phusion Passenger	Erlang
Обработка запросов	Ruby on Rails	Erlang
Запросы с долгим временем выполнения	Go	Erlang
Глобальное (server-wide) состояние	Redis	Erlang
Постоянные данные	Redis и MongoDB	Erlang
Фоновые задачи	Cron, Bash-скрипты и Ruby	Erlang
Аварийное восстановление	Upstart	Erlang

Сервер А поддерживается различными технологиями, большинство из которых популярно в сообществе разработчиков. Они были выбраны неслучайно: каждая из них добавлялась для того, чтобы покрыть недостатки предыдущей. Например, Ruby on Rails обрабатывает конкурентные запросы в разных процессах ОС, но нам было необходимо, чтобы процессы имели доступ к общим данным, и мы подключили Redis. Аналогично MongoDB был использован для управления постоянными данными фронтенда, чаще всего данными, связанными с пользователем. Таким образом, для использования каждой технологии сервера А есть логическое обоснование, но в целом решение выглядит слишком сложным. Каждая его часть – это самостоятельный проект, развертывание каждого компонента производится отдельно, а локально запустить всю систему – задача не из простых. Нам для этого даже пришлось создать свой инструмент!

Сервер Б же удовлетворяет всем техническим требованиям с помощью одной-единственной технологии, используя специально созданный для этих целей и отлично подходящий для больших систем функционал. При этом сервер представляет собой единый проект, запускающийся в одном экземпляре BEAM: на этапе эксплуатации он работает в одном процессе ОС, используя несколько потоков. Конкурентность полностью реализуется планировщиком Erlang, а система является масштабируемой, отзывчивой и отказоустойчивой. Учитывая, что система – это единый проект, добавим к преимуществам легкое управление, развертывание и локальный запуск.

Важно отметить, что инструменты Erlang не всегда готовы полностью заменить такие зарекомендовавшие себя решения, как веб-серверы вроде Nginx, серверы баз данных наподобие Riak и размещаемые в памяти хранилища ключ/значение по типу Redis. Но в Erlang есть все необходимое для разработки системы, а если какого-либо функционала окажется недостаточно, всегда можно прибегнуть к использованию других технологий. Чем меньше их будет, тем проще разрабатывать и поддерживать систему.

Можно подумать, что Erlang – изолированный островок технологий, но это не так. Erlang может запускать C-код внутри процесса, а также взаимодействовать практически с любым внешним компонентом, будь то очереди сообщений, размещаемые в памяти хранилища ключ/значение или внешние базы данных. Следовательно, выбирая Erlang, вы не лишаете себя возможности использовать сторонние технологии. Напротив, вам предоставляется шанс использовать их тогда, когда основной платформе разработки не хватает функционала для решения той или иной задачи.

Теперь вы имеете представление о сильных сторонах Erlang, и мы можем перейти к более детальному изучению платформы.

1.1.4. Платформа разработки

Erlang – это не просто язык программирования. Это сформировавшаяся платформа разработки, состоящая из четырех отдельных частей: язык, виртуальная машина, фреймворк и инструменты.

Язык Erlang – язык, на котором преимущественно пишется исполняемый виртуальной машиной код. Это простой функциональный язык с базовыми примитивами конкурентности.

Написанный на Erlang исходный код компилируется в байт-код, а далее исполняется виртуальной машиной BEAM. Именно она распараллеливает выполнение конкурентных Erlang-программ и обеспечивает изолированность, распределение вычислений и общую отзывчивость системы. Настоящая магия.

Вторая составляющая платформы – фреймворк под названием *OTP* (*Open Telecom Platform – открытая телекоммуникационная платформа*). Несмотря на свое название, он никак не связан с телекоммуникационными системами. Это фреймворк общего назначения, абстрагирующий многие стандартные задачи:

- модели конкурентности и распределения;
- обнаружение ошибок и восстановление конкурентных систем;
- упаковка кода в библиотеки;
- развертывание систем;
- горячее обновление кода.

Все вышеперечисленное можно реализовать и без помощи OTP, но это не имеет смысла. OTP проверен на практике многими системами, и он является настолько важной частью платформы, что сложно провести черту между ними. Даже официальный дистрибутив называется Erlang/OTP.

Инструменты используются для выполнения таких стандартных задач, как компиляция написанного на Erlang кода, запуск экземпляра BEAM, создание версий для развертывания, запуск интерактивной оболочки, подключение к работающему экземпляру BEAM и т. п. BEAM и сопутствующие ей инструменты являются кроссплатформенными: они поддерживаются самыми популярными операционными системами – Unix, Linux и Windows. Дистрибутив Erlang находится в свободном доступе на официальном сайте (**http://erlang.org**) или в репозитории GitHub (**https://github.com/erlang/otp**). Компания Erricson продолжает заниматься разработкой языка и выпускает новые версии раз в год.

На этом хвалебные оды Erlang заканчиваются. Вы спросите: «Раз Erlang так хорош, зачем мне Elixir?» Попробую ответить на данный вопрос в следующем разделе.

1.2. Общие сведения об Elixir

Elixir – это альтернативный язык для работы с виртуальной машиной Erlang, позволяющий создавать более понятный, компактный и информативный код. Написанные на Elixir программы запускаются в виртуальной машине BEAM.

Elixir – это проект с открытым исходным кодом, разработанный Хосе Валимом (José Valim). В отличие от Erlang, Elixir – продукт совместного творчества; на сегодняшний день около 700 человек внесли свой вклад в его разработку. Новые функции часто освещаются посредством почтовых рассылок и обсуждаются в сервисе отслеживания ошибок GitHub и на канале #elixir-lang IRC-сети freenode. Окончательное решение всегда остается за Хосе, но тем не менее Elixir представляет собой открытый совместный проект, привлекающий как опытных Erlang-разработчиков, так и способных новичков. Исходные файлы проекта можно найти в репозитории GitHub по адресу **https://github.com/elixir-lang/elixir**.

Elixir ориентирован на среду выполнения Erlang. Результатом компиляции Elixir-кода являются совместимые с BEAM файлы, содержащие байт-код. Они мо-

гут быть запущены в экземпляре BEAM и с легкостью взаимодействуют с Erlang-кодом – вы можете использовать Erlang-библиотеки при разработке на Elixir, и наоборот. Все возможности Erlang проецируются на Elixir, в том числе и производительность.

Elixir близок к Erlang семантически: многие конструкции языка повторяют конструкции своего аналога. Однако есть и дополнительные конструкции, которые помогают значительно сократить количество шаблонного кода и дублирований. Помимо этого, Elixir приводит в порядок некоторые значимые части стандартных библиотек, а также предоставляет синтаксический сахар и инструмент для создания и упаковки систем. Все, что возможно в Erlang, возможно и в Elixir, но, на мой взгляд, код, написанный на Elixir, чаще всего более прост в разработке и поддержке.

Давайте посмотрим, как Elixir расширяет некоторые возможности Erlang. Начнем с проблемы шаблонного кода.

1.2.1. Упрощенный код

Одно из наиболее значимых преимуществ Elixir – это его способность заметно сократить количество лишнего кода, что упрощает его разработку и поддержку. Сравним код на Elixir и Erlang, чтобы увидеть разницу.

В конкурентных системах Erlang часто используется такой структурный элемент, как *серверный процесс*. Серверные процессы – это некие конкурентные объекты, имеющие приватное состояние и взаимодействующие с другими процессами посредством сообщений. Будучи конкурентными, разные процессы могут работать параллельно. Типовые Erlang-системы прочно опираются на процессы, которых могут иметь тысячи или даже миллионы.

В следующем примере показано, как на Erlang реализован простой серверный процесс, складывающий два числа.

Листинг 1.1 ❖ Реализация на Erlang серверного процесса, складывающего два числа

```
-module(sum_server).
-behaviour(gen_server).

-export([
  start/0, sum/3,
  init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2,
  code_change/3
]).

start() -> gen_server:start(?MODULE, [], []).
sum(Server, A, B) -> gen_server:call(Server, {sum, A, B}).

init(_) -> {ok, undefined}.
handle_call({sum, A, B}, _From, State) -> {reply, A + B, State}.
handle_cast(_Msg, State) -> {noreply, State}.
handle_info(_Info, State) -> {noreply, State}.
terminate(_Reason, _State) -> ok.
code_change(_OldVsn, State, _Extra) -> {ok, State}.
```

Даже не зная языка, можно сказать, что требуется слишком много действий, чтобы просто сложить два числа. Откровенно говоря, несмотря на то что сложение

выполняется конкурентно, из-за загроможденности кода трудно разглядеть его структуру. Сходу тяжело разобраться, что он вообще делает, да и создавать такой код непросто. Даже после нескольких лет профессиональной разработки на Erlang у меня не получается писать код, не обращаясь к документации или не копируя части предыдущих проектов.

Проблема Erlang заключается в том, что шаблонные участки кода практически невозможно исключить, даже если они в большинстве своем идентичны (как постоянно случалось у меня). Язык не предоставляет никаких средств для решения этой проблемы. Справедливости ради отмечу, что существует один способ борьбы с шаблонностью – конструкция `parse transform`, но она неудобна и сложна в использовании. На практике Erlang-разработчики реализуют серверные процессы вышеописанным способом.

Поскольку серверные процессы – важный и часто используемый в Erlang инструмент, разработчикам приходится постоянно прибегать к дублированию и работать с захламленным кодом. Удивительно, но многие из них привыкают к этому, наверное, из-за тех преимуществ, что дает им BEAM. Говорят, Erlang упрощает сложное, но усложняет простое. В любом случае, код из предыдущего примера дает понять, что в нем есть от чего избавиться.

Давайте посмотрим, как с той же задачей справится Elixir.

Листинг 1.2 ❖ Реализация на Elixir серверного процесса, складывающего два числа

```
defmodule SumServer do
  use GenServer

  def start do
    GenServer.start(__MODULE__, nil)
  end

  def sum(server, a, b) do
    GenServer.call(server, {:sum, a, b})
  end

  def handle_call({:sum, a, b}, _from, state) do
    {:reply, a + b, state}
  end
 end
```

Кода в данном примере заметно меньше, а значит, его легче читать и поддерживать. Его назначение более понятно, и он не так замусорен. При этом он делает все то же самое, что и код на Erlang, и имеет абсолютно такое же поведение во время выполнения, полностью сохраняя семантику. Нет ничего такого в Erlang, с чем Elixir бы не справился.

Пусть код и получился более компактным, он все равно кажется избыточным для выполнения всего одной функции – сложения двух чисел. Все потому, что Elixir имеет семантическую связь с базовой Erlang-библиотекой, используемой для создания серверных процессов.

Однако Elixir предоставляет инструменты для дальнейшего исключения всего того, что вы посчитаете избыточным. К примеру, я создал свою Elixir-библиотеку *ExActor*, которая сокращает объявление серверного процесса, как показано ниже.

Листинг 1.3 ❖ Реализация серверного процесса на Elixir

```
defmodule SumServer do
  use ExActor.GenServer

  defstart start

  defcall sum(a, b) do
    reply(a + b)
  end
 end
```

Идея данного кода должна быть понятна даже разработчикам, не имеющим опыта программирования на Elixir. Во время выполнения код работает практически так же, как и две предыдущие версии. Поведение этого кода становится абсолютно идентичным на этапе компиляции: обычный код превращается в байт-код, и все три версии работают одинаково.

> **ПРИМЕЧАНИЕ** Я привел пример своей библиотеки только ради того, чтобы показать, как много всего можно абстрагировать в Elixir. В данной книге мы не будем использовать эту библиотеку, так как она является сторонней абстракцией и скрывает важнейшие подробности того, как работают серверные процессы. Чтобы извлечь настоящую выгоду из использования серверных процессов, необходимо понимать, как именно они приводятся в действие. Поэтому в книге рассматриваются абстракции более низкого уровня. Как только вы будете иметь представление о том, как работают серверные процессы, вы сможете сами принять решение, насколько вам необходима библиотека ExActor.

Последний пример реализации серверного процесса `sum` основан на работе макросов. *Макрос* – это код на Elixir, запускаемый во время компиляции. Макросы получают на входе машинное представление вашего кода и выдают альтернативный результат. Макросы Elixir были созданы по примеру Lisp, но их не стоит путать с C-образными макросами. В отличие от макросов C/C++, работающих с обычным текстом, макросы в Elixir работают с абстрактным синтаксическим деревом (АСД), что упрощает выполнение более сложных операций входного кода и получение альтернативного результата. И конечно, Elixir предоставляет вспомогательные конструкции, для того чтобы сделать процесс преобразования простым и понятным.

Взгляните еще раз, как в предыдущем примере определена операция суммирования:

```
defcall sum(a, b) do
  reply(a + b)
end
```

Обратите внимание на конструкцию `defcall` в самом начале. В Elixir нет такого ключевого слова, это специально разработанный макрос, трансформирующий данное объявление в нечто вроде этого:

```
def sum(server, a, b) do
  GenServer.call(server, {:sum, a, b})
end

def handle_call({:sum, a, b}, _from, state) do
  {:reply, a + b, state}
 End
```

Написанные на Elixir макросы – гибкий и мощный инструмент, они позволяют расширить возможности языка и добавить новые конструкции, которые выглядят очень натурально. Например, проект с открытым исходным кодом Ecto, целью которого является введение в Elixir запросов LINQ, также работает с помощью макросов и предоставляет выразительный синтаксис запросов, который выглядит так, будто является частью языка:

```
from w in Weather,
  where: w.prcp > 0 or w.prcp == nil,
  select: w
```

Благодаря поддержке макросов и продуманной архитектуре компилятора большая часть функционала Elixir написана на Elixir. Такие конструкции языка, как `if` и `unless`, а также поддержка структур прописаны с помощью макросов. Совсем небольшая база реализована на Erlang, а все остальное построено поверх нее на Elixir.

Макросы в Elixir – это своеобразная черная магия, но при этом они избавляют от шаблонного кода на этапе компиляции и помогают вам расширить возможности языка своими предметно-ориентированными конструкциями.

Тем не менее смысл Elixir не в одних только макросах. Еще одно важное преимущество – синтаксический сахар, значительно облегчающий процесс написания программ на функциональных языках.

1.2.2. Композиция функций

Erlang и Elixir – функциональные языки программирования. Их фундамент – иммутабельные данные и функции, преобразующие эти данные. Предположительным достоинством такого подхода является разбиение кода на несколько небольших композиционных повторно используемых функций.

К сожалению, реализация композиционных функций на Erlang выглядит громоздко. Приведу упрощенный пример из своей практики. Моя часть кода отвечает за поддержку расположенной в памяти модели и получение XML-сообщений, которые вносят изменения в модель. При получении XML-сообщения выполняются следующие действия:

- загрузить XML в память модели;
- применить изменения;
- сохранить модель.

Ниже приведен набросок кода на Erlang, реализующего соответствующую функцию:

```
process_xml(Model, Xml) ->
  Model1 = update(Model, Xml),
  Model2 = process_changes(Model1),
  persist(Model2).
```

Не знаю, как вам, но мне эта функция не кажется композиционной. Код довольно избыточен и уязвим для ошибок. Временные переменные `Model1` и `Model2` введены только для того, чтобы принять результат одной функции и передать его другой.

Конечно, можно обойтись и без временных переменных и использовать вложенные вызовы:

```
process_xml(Model, Xml) ->
  persist(
    process_changes(
      update(Model, Xml)
   )
).
```

Этот стиль называется лестничным (staircasing). Да, временных переменных тут нет, но сама реализация трудночитаема и выглядит громоздко. Чтобы понять, что делает код, придется распарсить его вручную.

Erlang-разработчики в какой-то степени ограничены такими неэстетическими решениями, а Elixir предлагает элегантный способ объединения нескольких вызовов функций:

```
def process_xml(model, xml) do
  model
   |> update(xml)
   |> process_changes
   |> persist
 end
```

Оператор конвейера (`|>`) принимает результат предыдущего выражения и передает его в следующее в качестве первого аргумента. С помощью этого оператора мы получаем аккуратный код без временных переменных. Он читается легко и непринужденно, как художественное произведение. На этапе компиляции данный код превращается в его «лестничную» версию. Все это возможно благодаря системе макросов Elixir.

Оператор конвейера иллюстрирует мощь функционального программирования. Функции рассматриваются как преобразования данных и затем комбинируются различными способами для достижения желаемой цели.

1.2.3. Выводы

Elixir еще много в чем превосходит Erlang. API стандартных библиотек очищен от всего лишнего и следует определенным соглашениям. Синтаксический сахар упрощает некоторые типовые идиомы. Присутствует лаконичный синтаксис для работы со структурированными данными. Операции над строками включают поддержку операций с Unicode в явном виде. По части инструментов Elixir предоставляет инструмент `mix`, упрощает стандартные задачи вроде создания приложений и библиотек, управления зависимостями, компиляции и тестирование кода. А еще доступен менеджер пакетов Hex (**https://hex.pm/**), с помощью которого удобно реализовывать упаковку, распространение и повторное использование зависимостей.

Преимущества можно перечислять бесконечно, но вместо этого я бы хотел высказать собственное мнение, основываясь на своем опыте профессиональной разработки. Мне больше нравится программировать на Elixir: код выглядит проще, читабельнее и чище благодаря отсутствию шаблонного кода и дублирования. На-

ряду с этим во время выполнения сохраняются все характеристики кода на Erlang, а также есть возможность использовать доступные стандартные и/или сторонние библиотеки экосистемы Erlang.

1.3. Недостатки

Идеальных технологий не существует, и здесь Erlang и Elixir не являются исключениями. Настало время поговорить об их слабых местах.

1.3.1. Скорость

Erlang – далеко не самая быстрая из существующих платформ. Если поискать в интернете результаты различных синтетических тестов, то вряд ли удастся увидеть Erlang на первых позициях списков. Написанные на Erlang программы запускаются в виртуальной машине и потому не могут достичь скорости компилируемых языков вроде C и C++. Но это происходит не случайно и из-за просчетов создателей Erlang/OTP.

Целью платформы является не достижение максимально возможного количества запросов в секунду, а поддержание производительности в определенных пределах. Показатели производительности Erlang-системы на конкретном компьютере не должны существенно ухудшаться. То есть не должно случаться внезапных провисаний системы из-за того, что, скажем, включился в работу сборщик мусора. Более того, как было освещено выше, процессы BEAM с долгим временем выполнения не блокируют систему и не оказывают значительного влияния на ее работоспособность. Наконец, с увеличением нагрузки виртуальная машина BEAM может задействовать столько аппаратных ресурсов, сколько ей понадобится, а если ресурсов недостаточно, система начнет постепенное торможение – запросы будут обрабатываться дольше, но система продолжит работать. Все это происходит благодаря вытесняющей природе планировщика BEAM, который отдает предпочтение процессам с коротким временем выполнения и обеспечивает частое переключение контекста, что поддерживает систему на плаву. Разумеется, более высокие нагрузки можно компенсировать, добавив больше аппаратных ресурсов.

Несмотря на все это, трудоемкие вычисления в ЦП не настолько эффективны, как их аналоги на C/C++, поэтому есть смысл реализовывать такие задачи на другом языке, а затем интегрировать соответствующий компонент в свою Erlang-систему. Если большая часть логики системы основана на ЦП, то лучше изначально выбрать другую технологию.

1.3.2. Экосистема

Выстроенная вокруг Erlang *экосистема* не такая уж и маленькая, но до экосистем некоторых языков ей еще очень далеко. На момент написания данной книги поиск по GitHub показывает около 20 000 репозиториев Erlang и примерно 36 000 репозиториев Elixir. Для сравнения: репозиториев Ruby нашлось около 1 500 000, а для JavaScript – 5 000 000.

Имейте в виду, что выбор библиотек не такой обширный, как вы, возможно, привыкли. На решение некоторых задач может потребоваться дополнительное время, в отличие от других языков. При этом не забывайте о преимуществах, которые дает вам Erlang. Как я уже говорил, Erlang – отличный выбор при разработке отказоустойчивых систем с минимальным временем простоя, платформа буквально заточена под это. И, несмотря на то что экосистема языка недостаточно зрелая, я считаю, что Erlang в значительной степени помогает решить сложные задачи, пусть иногда и не самым красивым способом. А в случае если вашей системе не требуется работать бесперебойно, быть отказоустойчивой и испытывать высокие нагрузки, лучше подобрать другой стек технологий с более развитой экосистемой.

Выводы

- Erlang – это технология для разработки высокодоступных систем с небольшим или нулевым временем простоя. Она успешно применяется в различных объемных системах уже более 20 лет.
- Elixir – современный язык, упрощающий разработку для платформы Erlang. Он помогает более эффективно организовать код и исключить шаблонность и дублирование.

Глава 2

Основы языка

В главе рассматривается:
- использование интерактивной оболочки;
- работа с переменными;
- организация кода;
- система типов;
- работа с операторами;
- среда выполнения.

Пришло время заняться изучением Elixir. В данной главе рассматриваются основные структурные элементы языка – модули, функции и система типов. Представленный в главе материал может показаться не таким увлекательным, но он послужит основой при дальнейшем изучении более интересных и сложных аспектов языка.

Для начала установите Elixir версии 1.7.x и Erlang версии 21. Установить Elixir можно несколькими различными способами, но лучше всего следовать указаниям на официальном сайте **https://elixir-lang.org/install.html**.

Теперь все готово для нашего путешествия, и первый пункт его назначения – интерактивная оболочка.

Подробная информация

В данной книге мы не будем подробно останавливаться на каких-либо особенностях языка и платформы – это слишком большой объем информации, которая быстро теряет свою актуальность. Тем не менее для более детального погружения в язык вы можете воспользоваться следующими источниками:

- руководство «Getting Started» на официальном сайте Elixir (**https://elixir-lang.org/getting-started/introduction.html**) поможет быстро освоить альтернативный синтаксис языка, а онлайн-документация даст более глубокие знания об этом (**https://hexdocs.pm/elixir**);
- форум Elixir (**https://elixirforum.com/**), канал #elixir-lang в IRC (**irc://irc.freenode.net/elixir-lang**) и канал в Slack (**https://elixir-slackin.herokuapp.com/**) помогут разобраться в нестандартных вопросах;
- *документация* Erlang (**http://www.erlang.org/doc**) также будет во многом полезна, а если вы незнакомы с синтаксисом Erlang, возможно, будет нелишним пройти краткий курс на официальном сайте Elixir (**https://elixir-lang.org/crash-course.html**).

2.1. Интерактивная оболочка

Самый простой способ познакомиться с особенностями любого языка – «поиграть» с его *интерактивной оболочкой*. Запустить ее можно, выполнив команду `iex` в командной строке:

```
$ iex
Erlang/OTP 21 [erts-10.0.8] [source] [64-bit] [smp:8:8] [ds:8:8:10]
  [async-threads:10] [hipe] [kernel-poll:false]

Interactive Elixir (1.7.3) - press Ctrl+C to exit
  (type h() ENTER for help)

iex(1)>
```

Данная команда запускает экземпляр BEAM, а затем и интерактивную оболочку внутри него. После этого на экран выводится информация о версиях Elixir и Erlang и строка приглашения, после которой можно вводить *Elixir-выражения*:

```
iex(1)> 1 + 2  <-- Elixir-выражение
3              <-- Результат выполнения выражения
```

Введенное выражение затем интерпретируется и выполняется, а его результат выводится на экран.

> **ПРИМЕЧАНИЕ** Любая конструкция в Elixir является выражением и имеет возвращаемое значение. Это касается не только вызовов функций, но и конструкций `if` и `case`.

> **ПОДСКАЗКА** В примерах данной книги, а особенно в первых главах, довольно часто используется `IEx`. Результат выражения чаще всего будет неважен, а потому будет опущен. Несмотря на это, не стоит забывать, что каждое выражение возвращает некий результат, и при выполнении примера в командной строке вы непременно его увидите.

Вводимый код может быть практически любым, главное, чтобы он представлял собой правильный код на Elixir. Например, можно выполнять вот такие более сложные многострочные выражения:

```
iex(2)> 2 * (
          3 + 1   | Продолжение выражения
        ) / 4     <-- Конец выражения, подсчет результата
2.0
```

Имейте в виду, выражение не будет выполнено, пока вы не допишете его до конца. В Elixir конец выражения обозначается не специальными символами вроде точки с запятой, а разрывом строки. Если при этом выражение не завершено, синтаксический анализатор командной строки не приступит к его выполнению, пока оно не будет введено полностью.

Быстрее всего выйти из оболочки, дважды нажав **Ctrl**+**C**. Это сразу завершит процесс ОС и все выполняющиеся фоновые задачи. Так как оболочка в основном используется для практики, а не для запуска реальных промышленных систем, такой способ вполне подойдет. Есть также и более щадящий метод завершения работы системы – команда `System.halt`.

> **ПРИМЕЧАНИЕ** Существует несколько способов запуска среды выполнения Elixir и Erlang, а также приложений на Elixir. К концу этой главы вы узнаете немного о каждом из этих способов. В первой части книги в основном будет использована оболочка IEx как наиболее простой и эффективный способ познания языка на практике.

С помощью оболочки можно делать многое, но в большинстве случаев она используется для вычисления результата какого-либо выражения. На что конкретно она способна, можно посмотреть в справке, выполнив команду h:

```
iex(4)> h
```

После этого на экране появится целый список возможных в IEx команд. Дополнительно можно ознакомиться с документацией *модуля IEx*, отвечающего за работу оболочки:


```
iex(5)> h IEx
```

Та же информация содержится в онлайн-документации по адресу **https://hexdocs.pm/iex**.

Теперь, когда в вашем распоряжении есть инструмент для экспериментов, пора перейти к изучению основных средств языка. Начнем с переменных.

2.2. Работа с переменными

Elixir – это динамический язык программирования, а значит, переменные и их тип не должны быть объявлены в явном виде. Тип переменной определяется по типу данных, которые она содержит в текущий момент. В Elixir присваивание значения называется *привязкой*. Когда вы задаете значение переменной, она привязывается к этому значению:

```
iex(1)> monthly_salary = 10000  ◄—— Привязка значения
10000  ◄—— Результат предыдущего выражения
```

Любое выражение в Elixir возвращает результат. В случае с оператором = результатом является то, что находится справа от него. После вычисления выражения оболочка выводит результат на экран.

Вы можете обратиться к переменной:

```
iex(2)> monthly_salary  ◄—— Выражение, возвращающее значение переменной
10000  ◄—— Значение переменной
```

Переменные также могут быть использованы в более сложных выражениях:

```
iex(3)> monthly_salary * 12
120000
```

В Elixir имя переменной всегда должно начинаться со строчной буквы или символа подчеркивания. После этого может следовать любая комбинация букв, цифр и подчеркиваний. Обычно используется следующее соглашение: имена переменных содержат только строчные буквы таблицы ASCII, числа и подчеркивания:

```
valid_variable_name
also_valid_1
```

```
validButNotRecommended
NotValid
```

Имена переменных могут оканчиваться знаком вопроса (?) или восклицания (!):

```
valid_name?
also_ok!
```

К переменной можно привязать новое значение:

```
iex(1)> monthly_salary = 10000  <-- Привязка начального значения
10000

iex(2)> monthly_salary          <-- Проверка значения
10000

iex(3)> monthly_salary = 11000  <-- Привязка нового значения
11000

iex(4)> monthly_salary          <-- Проверка нового значения
11000
```

При повторной привязке адрес значения в памяти не изменяется, а вместо этого символьное имя переменной переназначается на другой адрес.

> **ПРИМЕЧАНИЕ** Всегда помните о том, что данные *иммутабельны*. Как только данные помещаются в память, их адрес не может быть изменен до тех пор, пока память не будет освобождена. Однако к переменной можно привязать новое значение, которое будет иметь другой адрес. Таким образом, переменные изменяемы, а вот данные, на которые они указывают, – нет.

В Elixir сбор мусора осуществляется автоматически: когда переменная выходит за пределы области видимости, через какое-то время сборщик мусора освобождает занимаемую ею память.

2.3. Организация кода

Основой любого функционального языка являются функции. Благодаря иммутабельности данных типовые программы на Elixir состоят из множества небольших функций. Это станет очевидным в главах 3 и 4, когда вы изучите основные идиомы функционального программирования. Функции также могут быть объединены в модули.

2.3.1. Модули

Модуль – это группа функций, что-то вроде пространства имен. Функции в Elixir должны быть определены внутри модуля.

По умолчанию в Elixir доступна стандартная библиотека, содержащая множество модулей. Например, *модуль* `IO` может быть использован для осуществления различных операций ввода-вывода. При помощи функции `puts` этого модуля можно вывести текст на экран:

```
iex(1)> IO.puts("Hello World!")
Hello World!
:ok
```

Вызов функции puts модуля IO
Функция IO.puts выводит текст на экран
Возвращаемое значение функции IO.puts

Как видно из примера, чтобы вызвать какую-либо функцию модуля, нужно использовать следующий синтаксис: `ИмяМодуля.имя_функции(аргументы)`.

Чтобы определить свой собственный модуль, используйте конструкцию `defmodule`, а функции внутри модуля нужно объявлять при помощи конструкции `def`. Пример определения модуля приведен в следующем листинге.

Листинг 2.1 ❖ Определение модуля (geometry.ex)

```
defmodule Geometry do
  def rectangle_area(a, b) do
    a * b
  end
end
```

Начало определения модуля
Определение функции
Конец определения модуля

Модуль можно включить в работу двумя способами. Первый – скопировать этот код прямо в `iex`, ведь, как отмечалось ранее, в оболочке можно вводить практически любой код. Второй – дать оболочке команду интерпретировать файл, прописав:

```
$ iex geometry.ex
```

Не важно, какой способ вы выберете, результат будет одинаковым: осуществится компиляция кода, и модуль загрузится в среду выполнения, после чего к нему можно будет обращаться прямо из оболочки. Давайте попробуем:

```
$ iex geometry.ex

iex(1)> Geometry.rectangle_area(6, 7)
42
```

Вызов функции
Результат функции

Проще простого! Мы создали модуль `Geometry`, загрузили его в текущую сессию оболочки и воспользовались им, чтобы вычислить площадь прямоугольника.

> **ПРИМЕЧАНИЕ** Как вы могли заметить, имя файла имеет расширение .ex. Это общепринятый стандарт для файлов исходного кода Elixir.

Модули должны быть объявлены в отдельных файлах исходного кода. Каждый такой файл может содержать несколько определений модулей:

```
defmodule Module1 do
  ...
end

defmodule Module2 do
  ...
end
```

К имени модуля предъявляется ряд требований. Во-первых, оно должно начинаться с прописной буквы и следовать «верблюжьему» стилю именования (CamelCase). Имя модуля может содержать буквы и цифры, символы подчеркивания и точки. Точки обычно используются для организации иерархической структуры модулей:

```
defmodule Geometry.Rectangle do
  ...
end

defmodule Geometry.Circle do
  ...
end
```

Точка – всего лишь удобство, синтаксический сахар, позволяющий организовывать функции по областям видимости. После компиляции кода между модулями не возникает никакой иерархии, она не нужна для работы сервисов.

Можно также создавать вложенные модули:

```
defmodule Geometry do
  defmodule Rectangle do
    ...
  end
  ...
end
```

К дочернему модулю можно обратиться так: `Geometry.Rectangle`. Опять же, подобные вложения нужны только для удобства, а после компиляции между модулями `Geometry` и `Geometry.Rectangle` нет никакой связи.

2.3.2. Функции

Функции в Elixir всегда являются частью модуля. Правила их именования такие же, как и для переменных: первый символ – строчная буква или подчеркивание, а далее следует комбинация букв, цифр и подчеркиваний.

Имена функций могут оканчиваться символами ? и !, как и имена переменных. Символ вопроса обычно используется для обозначения функций, возвращающих значение *true* или *false*. Восклицательный знак ставится в конце имени функции, которая может вызвать ошибку во время выполнения кода. Это скорее соглашения, чем правила, но для достижения единообразия лучше им следовать.

Функцию можно определить с помощью макроса `def`:

```
defmodule Geometry do
  def rectangle_area(a, b) do   <-- Объявление функции
    ...                         <-- Тело функции
  end
end
```

Определение начинается с конструкции `def`, после которой следует имя функции, список ее аргументов и тело, помещенное в блок `do…end`. Поскольку Elixir – динамический язык, типы аргументов не указываются.

> **ПРИМЕЧАНИЕ** Обратите внимание, что `defmodule` и `def` не ключевые слова, а специальные конструкции – *макросы*. Далее в этой главе вы узнаете, как именно они работают, а пока не стоит об этом беспокоиться. Для простоты можно считать их ключевыми словами, но помните, что на самом деле это не так.

Если у функции нет аргументов, скобки можно опустить:

```
defmodule Program do
  def run do
    ...
  end
end
```

Поговорим немного о возвращаемом значении функций. Помните, в Elixir всё, что имеет возвращаемое значение, называется *выражением*. Возвращаемым значением функции в Elixir является результат ее последнего выражения, и функции не возвращают значение в явном виде.

> **ПРИМЕЧАНИЕ** Скорее всего, на этом месте вы задумались, как же тогда работают сложные функции. Эта тема будет раскрыта в третьей главе, где вы узнаете о ветвлении и условной логике. Главное – старайтесь создавать простые и короткие функции, и вы сможете с легкостью вычислить результат их последнего выражения и возвратить его.

В листинге 2.1 уже приводился пример возвращаемого значения, но давайте вернемся к нему:

```
defmodule Geometry do
  def rectangle_area(a, b) do
    a * b      <-- Расчет площади и возвращение результата
  end
end
```

Теперь проверим. Перезапустите оболочку и вызовите функцию `rectangle_area`:

```
$ iex geometry.ex

iex(1)> Geometry.rectangle_area(3, 2)    <-- Вызов функции
6      <-- Возвращаемое значение функции
```

Небольшие определения функций можно помещать в одну строку:

```
defmodule Geometry do
  def rectangle_area(a, b), do: a * b
end
```

Чтобы вызвать функцию, определенную в другом модуле, необходимо указать имя модуля, а затем имя функции:

```
iex(1)> Geometry.rectangle_area(3, 2)
6
```

Разумеется, результат функции можно присваивать переменной:

```
iex(2)> area = Geometry.rectangle_area(3, 2)    <-- Вызов функции и присваивание ее результата переменной
6

iex(3)> area    <-- Проверка значения переменной
6
```

Скобки в Elixir являются необязательными, их всегда можно опустить:

```
iex(4)> Geometry.rectangle_area 3, 2
6
```

По моему мнению, опуская скобки, вы делаете код двусмысленным. Поэтому я советую всегда использовать их при вызове функций.

Инструмент форматирования кода

Начиная с версии 1.6 в составе Elixir поставляется *инструмент форматирования кода*, позволяющий привести код в соответствие и не беспокоиться об отступах и расстановке скобок.

Например, следующий фрагмент кода:

```
defmodule Client do
def run do
Geometry.rectangle_area 3,2
end
end
```

после форматирования будет выглядеть так:

```
defmodule Client do
  def run do
     Geometry.rectangle_area(3, 2)
  end
end
```

Отформатировать код можно с помощью задачи `mix format` (**https://hexdocs.pm/mix/Mix.Tasks.Format.html**) или установив инструмент форматирования в ваш редактор кода.

Если определение функции находится в том же модуле, в котором она вызывается, префикс модуля можно опустить:

```
defmodule Geometry do
  def rectangle_area(a, b) do
    a * b
  end

  def square_area(a) do
    rectangle_area(a, a)
  end
end
```

Вызов функции, находящейся в том же модуле

Так как Elixir – функциональный язык программирования, часто возникает необходимость объединения функций через передачу результата одной функции в качестве аргумента другой. В Elixir специально для этого есть встроенный оператор |>, называемый *оператором конвейера*:

```
iex(5)> -5 |> abs() |> Integer.to_string() |> IO.puts()
5
```

Во время компиляции этот код превращается в следующее выражение:

```
iex(6)> IO.puts(Integer.to_string(abs(-5)))
5
```

В общем случае оператор конвейера помещает результат предыдущего вызова на место первого аргумента следующего вызова. Таким образом, следующий код:

```
prev(arg1, arg2) |> next(arg3, arg4)
```

во время компиляции выглядит так:

```
next(prev(arg1, arg2), arg3, arg4)
```

Пожалуй, версия с конвейером более читабельна, потому что действия выполняются по порядку слева направо. Это выглядит особенно элегантно в файлах исходного кода, когда конвейер можно разбить на несколько строк:

```
-5 ◄—— Начальное значение –5
|> abs() ◄—— Подсчет модуля числа
|> Integer.to_string() ◄—— Конвертация числа в строку
|> IO.puts() ◄—— Вывод значения на экран
```

> **ПРИМЕЧАНИЕ** Оболочка не поддерживает многострочные конвейеры. Такое оформление доступно только в файлах исходного кода, а если перенести этот код в оболочку, то возникнет ошибка. Это происходит потому, что первая строка является независимым Elixir-выражением и незамедлительно выполняется оболочкой, а следующие за ней строки синтаксически неверны, поскольку начинаются с оператора |>.

2.3.3. Арность функций

Арность – термин для обозначения количества передаваемых функции аргументов. Функция однозначно определяется своим именем, именем содержащего ее модуля и арностью. Взгляните на следующую функцию:

```
defmodule Rectangle do
  def area(a, b) do ◄—— Функция с двумя аргументами
    ...
  end
end
```

Функции `Rectangle.area` передаются два аргумента, значит, ее арность равна двум. В Elixir принято обозначать функции как `Rectangle.area/2`, где /2 – это арность.

Зачем это нужно? Затем, что две функции с одинаковым именем и разной арностью – это две абсолютно разные функции. Рассмотрим пример.

Листинг 2.2 ❖ Определение функций с одинаковыми именами и разной арностью (arity_demo.ex)

```
defmodule Rectangle do
  def area(a), do: area(a, a) ◄—— Функция Rectangle.area/1

  def area(a, b), do: a * b ◄—— Функция Rectangle.area/2
end
```

Подгрузите этот модуль в оболочку и попробуйте ввести следующее:

```
iex(1)> Rectangle.area(5)
25

iex(2)> Rectangle.area(5,6)
30
```

Как видите, эти две функции работают совершенно по-разному, хотя и имеют одинаковые имена. Их арность отличается, поэтому они считаются двумя разными функциями с разными реализациями.

Объявлять две разные функции с одинаковыми именами и прописывать отдельно их реализации обычно не имеет смысла. Чаще всего функция с меньшей арностью делегирует функции с большей арностью, передавая стандартные аргументы. Именно это и происходит в листинге 2.2, когда функция `Rectangle.area/1` делегирует функции `Rectangle.area/2`.

Рассмотрим еще один пример.

Листинг 2.3 ❖ Определение функций с одинаковыми именами, различной арностью и стандартными параметрами (arity_calc.ex)

```
defmodule Calculator do
  def sum(a) do
    sum(a, 0)        <-- Calculator.sum/1 делегирует Calculator.sum/2
  end

  def sum(a, b) do   <-- Реализация содержится в Calculator.sum/2
    a + b
  end
end
```

Повторю, функция с меньшей арностью реализуется на основе функции с большей арностью. Этот шаблон проектирования используется так часто, что в Elixir даже есть специальный оператор \\, после которого указывается начальное значение аргумента:

```
defmodule Calculator do
  def sum(a, b \\ 0) do   <-- Определение начального значения аргумента b
    a + b
  end
end
```

Таким образом можно определить две функции, точно как в листинге 2.3.

Начальные значения можно задать нескольким аргументам:

```
defmodule MyModule do
  def fun(a, b \\ 1, c, d \\ 2) do   <-- Задание значений по умолчанию нескольким аргументам
    a + b + c + d
  end
end
```

Помните о том, что объявление функции с аргументами по умолчанию создает несколько функций с таким же именем и разной арностью. Код из предыдущего примера генерирует три функции: `MyModule.fun/2`, `MyModule.fun/3` и `MyModule.fun/4`.

Так как функции с одинаковыми именами различить можно только по арности, число аргументов функции не может изменяться, как в языках C и JavaScript.

2.3.4. Видимость функций

Объявляя функцию с помощью макроса `def`, вы делаете ее общедоступной, и она может быть использована где угодно. В Elixir этот процесс называется *экспорти-*

рованием. Также можно сделать функцию *приватной*, объявив ее при помощи макроса `defp`. Приватную функцию можно вызывать только в том модуле, в котором она определена. Рассмотрим это на примере.

Листинг 2.4 ❖ Модуль с публичными и приватными функциями (private_fun.ex)

```
defmodule TestPrivate do
  def double(a) do      <-- Публичная функция
    sum(a, a)           <-- Вызов приватной функции
  end

  defp sum(a, b) do     <-- Приватная функция
    a + b
  end
end
```

Модуль `TestPrivate` содержит определения двух функций. Функция `double` экспортируется и может быть вызвана за пределами модуля, а внутри него она вызывает приватную функцию `sum`.

Давайте попробуем сделать это в оболочке. Подгрузите модуль `TestPrivate` и выполните следующее:

```
iex(1)> TestPrivate.double(3)
6

iex(2)> TestPrivate.sum(3, 4)
** (UndefinedFunctionError) function TestPrivate.sum/2
...
```

Как видите, приватные функции можно вызывать за пределами модуля.

2.3.5. Импорты и псевдонимы

Вызывая функции из другого модуля, необходимо каждый раз указывать имя модуля, что загромождает код. Если вы часто используете в своем модуле функции какого-либо другого модуля, просто импортируйте его целиком. Так вы сможете вызывать его публичные функции без префикса модуля:

```
defmodule MyModule do
  import IO       <-- Импорт модуля

  def my_function do
    puts "Calling imported function."   <-- Теперь можно вызывать puts вместо IO.puts
  end
end
```

Импортировать можно и несколько модулей. На самом деле модуль стандартной библиотеки `Kernel`, содержащий наиболее часто используемые функции, по умолчанию импортируется в каждый новый модуль.

> **ПРИМЕЧАНИЕ** Список доступных функций модуля `Kernel` можно найти в онлайн-документации по адресу **https://hexdocs.pm/elixir/Kernel.html**.

Ещё одна конструкция – `alias` – позволяет обращаться к модулю под другим именем:

```
defmodule MyModule do
  alias IO, as: MyIO    ←——— Создание псевдонима для модуля IO

  def my_function do
    MyIO.puts("Calling imported function.")    ←——— Вызов функции с помощью псевдонима
  end
end
```

Псевдонимы будут особенно уместны для модулей с длинными именами. Если в вашем приложении существует несколько уровней иерархии модулей, их полные имена будут выглядеть громоздко. Например, возьмем модуль `Geometry.Rectangle`. Дадим ему псевдоним и обратимся к нему в другом модуле с помощью этого псевдонима:


```
defmodule MyModule do
  alias Geometry.Rectangle, as: Rectangle    ←——— Даем модулю псевдоним

  def my_function do
    Rectangle.area(...)    ←——— Вызываем модуль, используя его псевдоним
  end
end
```

В предыдущем примере псевдонимом модуля `Geometry.Rectangle` является последняя часть его имени. Чаще всего псевдонимы выбираются именно таким образом, и в этом случае можно опустить `as:`:


```
defmodule MyModule do
  alias Geometry.Rectangle    ←——— Даем модулю псевдоним

  def my_function do
    Rectangle.area(...)    ←——— Вызываем модуль, используя его псевдоним
  end
end
```

Псевдонимы помогают сделать код более емким, исключая из длинных имен модулей все лишнее.

2.3.6. Атрибуты модулей

Атрибуты модулей можно использовать как константы времени компиляции и регистрировать их для обращения во время выполнения программы. Рассмотрим это на примере модуля, предоставляющего базовые функции для работы с кругом:

```
iex(1)> defmodule Circle do
            @pi 3.14159    ←——— Определение атрибута модуля

            def area(r), do: r*r*@pi
            def circumference(r), do: 2*r*@pi    | Использование атрибута модуля
          end

iex(2)> Circle.area(1)
3.14159

iex(3)> Circle.circumference(1)
6.28318
```

Имейте в виду: модули можно объявлять прямо в оболочке. Это избавит вас от необходимости хранить файлы на диске.

Важно заметить, что константа `@pi` существует только во время компиляции модуля, когда обрабатываются ссылающиеся на нее строки.

Кроме того, атрибут можно *зарегистрировать*, то есть поместить в сгенерированный бинарный файл и обратиться к нему во время выполнения программы. Некоторые атрибуты модулей регистрируются в Elixir по умолчанию. Например, атрибуты `@moduledoc` и `@doc` используются для предоставления документации к модулям и функциям:

```
defmodule Circle do
  @moduledoc "Implements basic circle functions"
  @pi 3.14159

  @doc "Computes the area of a circle"
  def area(r), do: r*r*@pi

  @doc "Computes the circumference of a circle"
  def circumference(r), do: 2*r*@pi
end
```

Для того чтобы выполнить этот код, необходимо создать скомпилированный файл. Наиболее быстрый способ сделать это – скопировать код в файл circle.ex, сохранить его, а затем выполнить команду `elixirc circle.ex`. После этого сгенерируется файл Elixir.Circle.beam. Затем запустите оболочку из папки, где хранится созданный вами файл. Теперь атрибут можно получить во время выполнения:

```
iex(1)> Code.get_docs(Circle, :moduledoc)
{1, "Implements basic circle functions"}
```

Самое интересное здесь то, что другие инструменты экосистемы Elixir тоже умеют работать с атрибутами. Например, опцию поиска в `IEx` можно использовать для просмотра документации:

```
iex(2)> h Circle   <-- Документация модуля Circle

Implements basic circle functions

iex(3)> h Circle.area   <-- Документация функции def area(r)

Computes the area of a circle
```

Также можно воспользоваться инструментом `ex_doc` (**https://github.com/elixir-lang/ex_doc**) для написания HTML-документации своего проекта. Именно так создается документация Elixir, и если вы планируете разработку сложных проектов, особенно таких, в которых будет задействовано большое количество клиентов, стоит обратить внимание на `@moduledoc` и `@doc`.

Что важно, зарегистрированные атрибуты можно использовать для передачи модулю метаинформации, чтобы затем получить ее в иных инструментах Elixir и даже Erlang. Существуют и другие зарегистрированные по умолчанию атрибуты, а также можно регистрировать и свои собственные. За более подробной информацией обратитесь к документации модуля `Module` (**https://hexdocs.pm/elixir/Module.html**).

Спецификации типов

Спецификации типов – еще одно немаловажное свойство языка, работа которого основана на атрибутах. Оно позволяет добавлять к функциям информацию о типе, которая далее будет проанализирована инструментом статического анализа кода `dialyzer` (**http://erlang.org/ doc/man/dialyzer.html**).

Добавьте в модуль `Circle` спецификации типов:

```
defmodule Circle do
  @pi 3.14159

  @spec area(number) :: number      ← Спецификация типа для функции area/1
  def area(r), do: r*r*@pi

  @spec circumference(number) :: number      ← Спецификация типа для функции circumference\1
  def circumference(r), do: 2*r*@pi
end
```

Здесь атрибут `@spec` используется для того, чтобы обозначить, что обе функции принимают и возвращают число.

Спецификации типов – своеобразный способ компенсировать отсутствие статической системы типов. Очень удобно использовать их в связке с инструментом `dialyzer` для проведения статического анализа программ. Более того, спецификации типов позволяют улучшить читабельность документации функций. Elixir – динамический язык, а значит, входные и выходные параметры функций сложно определить просто по их сигнатуре. Спецификации типов помогут разрешить эту проблему, а также стоит отметить, что в стороннем коде гораздо проще разобраться, если в нем присутствуют спецификации типов.

Например, посмотрите на эту спецификацию типа для функции Elixir:

```
List.insert_at/3:
@spec insert_at(list, integer, any) :: list
```

Даже не обращаясь к коду или документации, можно легко догадаться, что данная функция вставляет терм любого типа (третий аргумент) в список (первый аргумент) в указанное место (второй аргумент) и возвращает новый список.

В данной книге в целях упрощения кода спецификации типов использованы не будут. Но имейте в виду, что в случае разработки более сложных систем к ним однозначно следует присмотреться. Более подробную информацию можно получить из официальной документации, перейдя по адресу: **https://hexdocs.pm/elixir/typespecs.html**.

2.3.7. Комментарии

В Elixir для обозначения комментариев используется символ #, после которого любой текст считается *комментарием*. Многострочные комментарии не поддерживаются. Если необходимо прокомментировать несколько строк, создавайте отдельный комментарий для каждой из них:

```
# Комментарий.
a = 3.14      # Еще один комментарий.
```

На этом погружение в базовые функции и модули закончено. Теперь вы знакомы с основными приемами организации кода, и мы можем перейти к изучению системы типов Elixir.

2.4. Понятие системы типов

В основе *системы типов Elixir* лежит система типов Erlang. Соответственно, обычно интеграция с библиотеками Erlang осуществляется легко. Сама система типов вполне простая, но если вы перешли на Elixir с классического объектно-ориентированного языка, она покажется совершенно иной, не такой, к какой вы привыкли. В данном разделе будут рассмотрены основные типы Elixir и приведено несколько реализаций иммутабельности. Начнем с чисел.

2.4.1. Числа

Числа могут быть целыми или вещественными, и выглядят они довольно привычно:

```
iex(1)> 3          ◄——— Целое число
3

iex(2)> 0xFF       ◄——— Целое число в шестнадцатеричной системе
255

iex(3)> 3.14       ◄——— Вещественное число
3.14

iex(4)> 1.0e-2     ◄——— Экспоненциальная форма записи вещественного числа
0.01
```

Поддерживаются стандартные арифметические операции:

```
iex(5)> 1 + 2 * 3
7
```

Оператор деления / работает не совсем обычно. Он всегда возвращает вещественное число:

```
iex(6)> 4/2
2.0

iex(7)> 3/2
1.5
```

Чтобы после деления получить целое число или остаток, можно использовать функции модуля `Kernel`, импортированные по умолчанию:

```
iex(8)> div(5,2)
2

iex(9)> rem(5,2)
1
```

Знаком подчеркивания можно визуально разделять классы числа:

```
iex(10)> 1_000_000
1000000
```

Размер целых чисел не имеет верхнего предела: можно использовать сколь угодно большие числа:

```
iex(11)>99999999999999999999999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999999999999999999999
```

Хранение чисел в памяти лучше всего описано в официальном руководстве Erlang (**http://erlang.org/doc/efficiency_guide/advanced.html**). Целое число занимает столько памяти, сколько ему необходимо, тогда как вещественное занимает либо 32, либо 64 бита в зависимости от архитектуры виртуальной машины. Внутри компьютера вещественные числа представлены в формате IEEE 754-1985 (с двойной точностью).

2.4.2. Атомы

Атомы – это именованные константы. Они напоминают символы в Ruby и перечисления в C/C++. Атомы начинаются с двоеточия и могут содержать буквы, цифры и нижние подчеркивания:

```
:an_atom
:another_atom
```

Следующий синтаксис позволяет использовать пробелы в имени атома:

```
:"an atom with spaces"
```

Атом состоит из двух частей – *текста* и *значения*. Текст атома – это все то, что находится после двоеточия. На этапе выполнения этот текст помещается в *таблицу атомов*. Значение – это данные, с которыми связывается переменная, это не что иное, как ссылка на таблицу атомов.

Вот почему атомы – наиболее подходящий вариант представления именованных констант. Они эффективны как с точки зрения памяти, так и с точки зрения производительности:

```
variable = :some_atom
```

В переменной хранится не весь текст, а лишь ссылка на таблицу атомов. Благодаря этому памяти требуется немного, сравнения выполняются быстро, и код остается читабельным.

Псевдонимы

Существует альтернативный синтаксис атомов: можно опустить двоеточие и начать с прописной буквы:

```
AnAtom
```

Такое объявление называется *псевдонимом*, и во время компиляции оно превращается в следующее:

```
:"Elixir.AnAtom":
iex(1)> AnAtom == :"Elixir.AnAtom"
true
```

Если вы используете псевдоним, компилятор неявно добавляет префикс `Elixir.` к тексту псевдонима и вставляет в него атом. Если псевдоним уже содержит этот префикс, то он не добавляется. То есть следующее выражение также истинно:

```
iex(2)> AnAtom == Elixir.AnAtom
true
```

Как вы уже знаете, модулям тоже можно давать псевдонимы:

```
iex(3)> alias IO, as: MyIO

iex(4)> MyIO.puts("Hello!")
Hello!
```

Эти два термина названы одинаково не случайно. Строка `alias IO, as: MyIO` дает компилятору указание превратить имя `MyIO` в `IO`. Конечным результатом этого действия является то, что в созданном бинарном файле будет фигурировать атом `:Elixir.IO`. Проверим это:

```
iex(5)> MyIO == Elixir.IO
true
```

Все это может показаться странным, однако имеет скрытые мотивы: псевдонимы помогают организовать выполнение кода модулей должным образом. Мы изучим это подробнее в конце данной главы, когда вернемся к модулям и рассмотрим, как они подгружаются во время выполнения.

Логические атомы

Возможно, вас удивит то, что в Elixir нет логического типа как такового. Вместо него используются атомы :true и :false. И здесь не обошлось без синтаксического сахара: ссылаться на эти атомы можно без начального символа двоеточия:

```
iex(1)> :true == true
true

iex(2)> :false == false
true
```

Термин «логическое значение» все равно используется в Elixir, но ему соответствует любой *атом*, имеющий значение `:true` или `:false`. С такими атомами работают стандартные логические операции:

```
iex(1)> true and false
false

iex(2)> false or true
true

iex(3)> not false
true

iex(4)> not :an_atom_other_than_true_or_false
** (ArgumentError) argument error
```

Всегда помните о том, что любое логическое значение – это всего лишь атом со значением `true` или `false`.

Nil и истинные значения

Атом `:nil` – еще один особый атом, значение которого в чем-то схоже со значением `null` в других языках программирования. Обращаться к нему можно без двоеточия:

```
iex(1)> nil == :nil
true
```

Атом `nil` в Elixir играет роль дополнительного средства для проверки *истинности* так же, как это происходит в популярных языках вроде C/C++ и Ruby. Атомы `nil` и `false` считаются *ложными* значениями, а все отличные от них – *истинными*.

Это свойство можно использовать вместе с выполняющимися по короткой схеме операторами Elixir `||`, `&&`, и `!`. Оператор `||` возвращает первое неложное значение:

```
iex(1)> nil || false || 5 || true
5
```

Так как `nil` и `false` – ложные выражения, оператор возвращает число 5. Важно отметить, что все последующие выражения вычисляться не будут. Если все выражения окажутся ложными, то будет возвращен результат последнего из них.

Оператор `&&` возвращает второе выражение, но только если первое является истинным. В противном случае будет возвращено первое выражение, а второе останется без внимания:

```
iex(1)> true && 5
5

iex(2)> false && 5
false

iex(3)> nil && 5
nil
```

Операторы, выполняющиеся по короткой схеме, – элегантный способ связывания нескольких операций. Например, если необходимо получить значение из кеша, локального диска или удаленной базы данных, можно сделать следующее:

```
read_cached || read_from_disk || read_from_database
```

Аналогично можно воспользоваться оператором `&&`, чтобы удостовериться в выполнении каких-либо условий:

```
database_value = connection_established? && read_data
```

И в первом, и во втором примерах эти операторы позволяют отказаться от использования сложных многоуровневых условных конструкций, что делает код более емким.

2.4.3. Кортежи

Кортежи представляют собой что-то вроде нетипизированных структур или записей и обычно используются для группировки заданного количества элементов. Следующий фрагмент кода определяет кортеж, содержащий имя и возраст человека:

```
iex(1)> person = {"Bob", 25}
{"Bob", 25}
```

Чтобы извлечь из кортежа какой-либо элемент, воспользуйтесь функцией `Kernel.elem/2`; ее аргументы – это кортеж и индекс элемента (отсчет элементов начи-

нается с нуля). Так как модуль Kernel импортируется по умолчанию, достаточно вызвать `elem` вместо полного имени `Kernel.elem`:

```
iex(2)> age = elem(person, 1)
25
```

Изменить элемент кортежа можно с помощью функции `Kernel.put_elem/3`, которая в качестве аргументов принимает кортеж, индекс и новое значение элемента в заданной позиции:

```
iex(3)> put_elem(person, 1, 26)
{"Bob", 26}
```

Функция `put_elem` не изменяет кортеж, а возвращает его новую версию, оставляя предыдущую нетронутой. Помните, данные в Elixir иммутабельны, а значит, значение в памяти изменить нельзя. Убедимся, что предыдущее действие не изменило значение переменной `person`:

```
iex(4)> person
{"Bob", 25}
```

Как же тогда использовать функцию `put_elem`? Просто присвойте ее результат другой переменной:

```
iex(5)> older_person = put_elem(person, 1, 26)
{"Bob", 26}

iex(6)> older_person
{"Bob", 26}
```

К переменным также можно привязывать значение повторно:

```
iex(7)> person = put_elem(person, 1, 26)
{"Bob", 26}
```

Таким образом мы привязываем переменную `person` к новой области памяти. На старый адрес больше не ссылается ни одна переменная, а значит, память будет вскоре очищена сборщиком мусора.

> **ПРИМЕЧАНИЕ** Возможно, здесь вы задумались, насколько данный подход эффективен в смысле управления памятью. В большинстве случаев данные будут копироваться в минимальном объеме, и две переменные будут разделять как можно больше памяти. Это будет подробно разобрано далее в этой главе, когда речь пойдет об иммутабельности.

Кортежи больше всего подходят для объединения фиксированного количества элементов. Если вас интересуют динамические коллекции, то лучше использовать списки.

2.4.4. Списки

Списки в Erlang используются для управления динамическими коллекциями данных переменного размера. Синтаксис крайне похож на синтаксис массивов в других языках:

```
iex(1)> prime_numbers = [2, 3, 5, 7]
[2, 3, 5, 7]
```

Может, списки и выглядят как массивы, но работают они как односвязные списки. Чтобы выполнить какое-либо действие со списком, его необходимо перебрать. Поэтому большинство операций имеет сложность O(n), включая функцию `Kernel.length/1`, которая обходит весь список и вычисляет его длину:

```
iex(2)> length(prime_numbers)
4
```

Вспомогательные функции списков

Над списками можно проделывать множество операций, однако в этом разделе будет рассмотрено только несколько самых базовых из них. Для получения более детальной информации обратитесь к документации модуля `List` (**https://hexdocs.pm/elixir/List.html**). Модуль `Enum` (**https://hexdocs.pm/elixir/Enum.html**) также предлагает большое количество полезных сервисов.

Возможности модуля `Enum` не ограничиваются операциями над списками, а позволяют организовать работу со многими перечисляемыми структурами данных. Концепция перечисляемых структур будет подробно разъяснена в главе 4 в разделе, посвященном протоколам.

Функция `Enum.at/2` позволяет получить элемент списка:

```
iex(3)> Enum.at(prime_numbers, 3)
7
```

Функция `Enum.at` также является операцией со сложностью O(n): она обходит каждый элемент с начала списка, пока не дойдет до нужного элемента. Списки – не лучшее решение для случаев, когда необходим прямой доступ к данным. Для этих целей больше подойдут кортежи, словари или структуры данных более высокого уровня.

Проверить, содержит ли список тот или иной элемент, можно с помощью оператора `in`:

```
iex(4)> 5 in prime_numbers
true

iex(5)> 4 in prime_numbers
false
```

Функции модуля `List` помогут организовать управление списками. Например, функция `List .replace_at/3` заменяет элемент в заданной позиции:

```
iex(6)> List.replace_at(prime_numbers, 0, 11)
[11, 3, 5, 7]
```

Точно так же, как и в случае с кортежами, функция не изменяет значение переменной, а возвращает новое значение, которое нужно присвоить другой переменной:

```
iex(7)> new_primes = List.replace_at(prime_numbers, 0, 11)
[11, 3, 5, 7]
```

Или можно повторно привязать его к той же переменной:

```
iex(8)> prime_numbers = List.replace_at(prime_numbers, 0, 11)
[11, 3, 5, 7]
```

Вставить новый элемент в указанную позицию можно с помощью функции `List.insert_at`:

```
iex(9)> List.insert_at(prime_numbers, 3, 13)
[11, 3, 5, 13, 7]
```

Вставка нового элемента на четвертую позицию

Чтобы добавить элемент в конец, укажите его позицию отрицательным числом:

```
iex(10)> List.insert_at(prime_numbers, -1, 13)
[11, 3, 5, 7, 13]
```

Значение –1 означает, что элемент должен быть добавлен в конец списка

Изменение произвольного элемента, как и большинство операций со списками, имеет сложность O(n). В частности, добавление элемента в конец списка проходит в *n* шагов, где *n* – длина списка.

Также доступен специальный оператор ++, объединяющий два списка:

```
iex(11)> [1, 2, 3] ++ [4, 5]
[1, 2, 3, 4, 5]
```

И его сложность тоже O(n), где *n* – длина левого списка (того, к которому добавляется новый). В общем случае стоит избегать добавления элементов к концу списка. Более эффективными действиями являются вставка элементов в начало списка и извлечение его начальных элементов. А чтобы понять, почему, обратим внимание на рекурсивную структуру списков.

Определение рекурсивного списка

Список также можно представить в виде рекурсивной структуры, состоящей из *головы* и *хвоста*, где голова – это первый элемент списка, а хвост «указывает» на голову и хвост оставшихся элементов, как показано на рис. 2.1. Если вы знакомы с языком Lisp, то ту же самую концепцию в нем представляет понятие пунктирной пары.

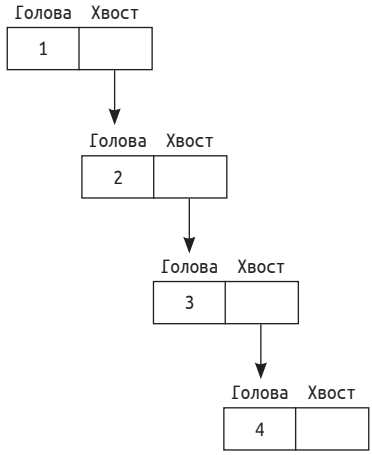


Рис. 2.1 ❖ Рекурсивная структура списка [1, 2, 3, 4]

В Elixir существует особый синтаксис для определения рекурсивного списка:

```
a_list = [head | tail]
```

Голова (head) может содержать данные любого типа, в то время как хвост (tail) представляет собой список. Если tail содержит пустой список, это сигнализирует о конце списка.

Приведу несколько примеров:

```
iex(1)> [1 | []]
[1]

iex(2)> [1 | [2 | []]]
[1, 2]

iex(3)> [1 | [2]]
[1, 2]

iex(4)> [1 | [2, 3, 4]]
[1, 2, 3, 4]
```

Это всего лишь еще один синтаксис для определения списка, показывающий, что на самом деле он из себя представляет. Это пара из двух значений – головы и хвоста, где хвост сам по себе является списком.

Следующий фрагмент кода – это классическое определение рекурсивного списка:

```
iex(1)> [1 | [2 | [3 | [4 | []]]]]
[1, 2, 3, 4]
```

Никто не заставляет вас использовать такие конструкции, но крайне важно постоянно помнить о том, что изнутри списки выглядят как рекурсивные структуры, состоящие из нескольких пар «голова–хвост».

Получить голову списка можно с помощью функции hd, а хвост – с помощью функции tl:

```
iex(1)> hd([1, 2, 3, 4])
1

iex(2)> tl([1, 2, 3, 4])
[2, 3, 4]
```

Обе операции имеют сложность O(1), так как они считывают одно значение из пары «голова–хвост».

> **ПРИМЕЧАНИЕ** Для полноты картины стоит отметить, что хвост не обязательно должен быть списком, он может быть абсолютно любого типа. В случае если он не является списком, это означает, что список неправильный, и большинство операций не будет над ним выполняться. Неправильные списки используются в особых случаях, но в этой книге они рассмотрены не будут.

Зная о внутреннем рекурсивном устройстве списков, добавить элемент в начало можно более простым и эффективным способом:

```
iex(1)> a_list = [5, :value, true]
[5, :value, true]
```

```
iex(2)> new_list = [:new_element | a_list]
[:new_element, 5, :value, true]
```

Составление нового списка `new_list` – операция со сложностью O(1), копирования памяти не происходит – хвостом списка `new_list` является список `a_list`. Чтобы понять, как все это работает, необходимо немного погрузиться в понятие иммутабельности.

2.4.5. Иммутабельность

Как отмечалось ранее, данные в Elixir не могут быть изменены. Каждая функция возвращает новую версию входных данных. Необходимо ввести новые переменные для хранения измененных значений или заново привязать их к тому же символьному имени. В любом случае, результат будет помещен в другую область памяти. Некоторые данные будут скопированы, но в основном старая и новая версии будут разделять общую память.

Рассмотрим данный алгоритм подробнее.

Модификация кортежей

Начнем с кортежей. Модифицированный кортеж является полной поверхностной копией предыдущей версии. Посмотрите на следующий код:

```
a_tuple = {a, b, c}
new_tuple = put_elem(a_tuple, 1, b2)
```

Как показано на рис. 2.2, переменная `new_tuple` содержит точную копию кортежа `a_tuple`, за исключением второго элемента.

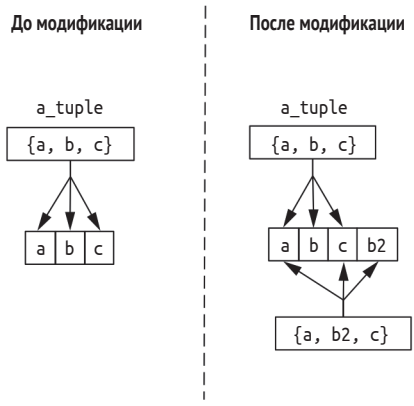


Рис. 2.2 ❖ Создание копии кортежа при его модификации

Оба кортежа ссылаются на переменные `a` и `c`, а данные, хранящиеся в этих переменных, не дублируются, а являются общими для обоих кортежей.

А что будет, если привязать к переменной новое значение? В этом случае после повторной привязки переменная `a_tuple` будет ссылаться на новую область памяти, а старая будет недоступна и впоследствии очищена сборщиком мусора.

То же самое произойдет и с переменной b, на которую ссылается предыдущая версия кортежа, как видно по рис. 2.3.

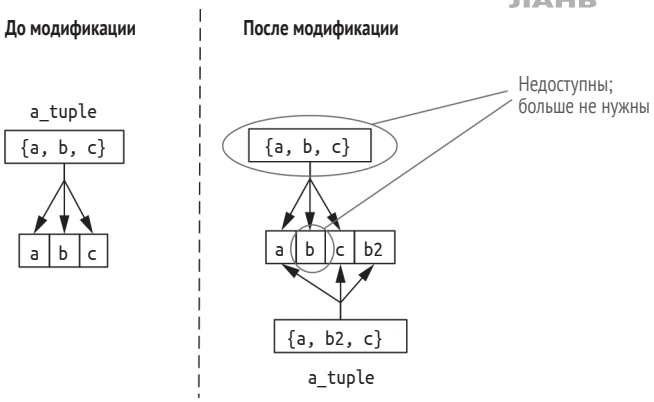


Рис. 2.3 ❖ Повторная привязка кортежа и освобождение памяти

Итак, при модификации кортежей всегда происходит их поверхностное копирование. Со списками же дела обстоят иначе.

Модификация списков

После модификации n-го элемента списка его новая версия будет содержать поверхностные копии первых n – 1 элементов и следующего за ними измененного элемента, а хвост списка будет общим. Это подробно показано на рис. 2.4.

Вот почему добавление элементов в конец списка требует много памяти: чтобы выполнить такую операцию, необходимо обойти весь список и сделать его поверхностную копию.

На рис. 2.5 можно видеть, что для вставки элементов в начало списка не требуется копирование данных, и память расходуется более разумно. В данном случае хвостом нового списка является предыдущий список. Такая схема часто используется в программах на Elixir для итеративного создания списков. При этом лучше всего добавлять следующие друг за другом элементы в начало, а когда список будет готов, изменить порядок его элементов на обратный в одно действие.

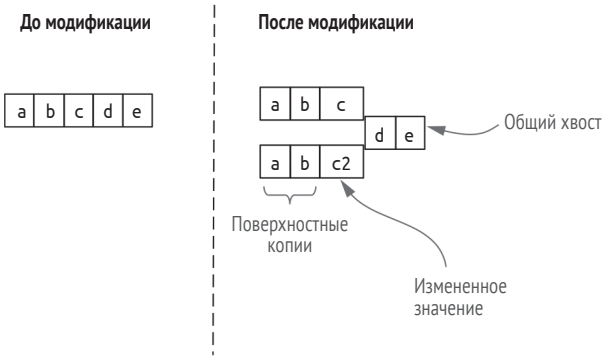


Рис. 2.4 ❖ Модификация списка

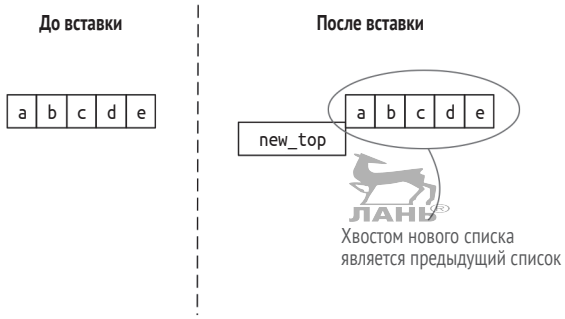


Рис. 2.5 ❖ Вставка нового элемента в начало списка

Преимущества иммутабельности

Организация иммутабельности может показаться странной, но у нее есть два важных преимущества – функции без побочных эффектов и согласованность данных.

Поскольку данные не могут быть изменены, большинство функций можно считать не имеющими побочных эффектов преобразованиями, которые принимают данные и возвращают результат. Сложные программы состоят из комбинаций простых преобразований:

```
def complex_transformation(data) do
  data  |> transformation_1(...)
  |> transformation_2(...)
  ...
  |> transformation_n(...)
end
```

Основой данного кода является оператор конвейера, который связывает функции, передавая результат предыдущего вызова в качестве аргумента следующему вызову.

Функции без побочных эффектов проще понимать, анализировать и тестировать; их входные и выходные данные четко определены. При вызове функции переменные не будут неявно изменены. Какие бы действия ни выполнялись функцией, необходимо получить ее результат и использовать его определенным образом.

> **ПРИМЕЧАНИЕ** Elixir не полностью функциональный язык, поэтому функции все же могут иметь побочные эффекты. Например, в результате выполнения функции произведена запись в файл и вызов сетевой функции или функции базы данных, что породило побочный эффект. Но можно не бояться: функция не изменит значение ни одной переменной.

Скрытым последствием иммутабельности данных является хранение всех версий структур данных программы. Это, в свою очередь, позволяет выполнять атомарные операции в памяти. Допустим, у вас есть функция, осуществляющая ряд преобразований:

```
def complex_transformation(original_data) do
  original_data
  |> transformation_1(...)
```

```
  |> transformation_2(...)
  ...
end
```

В самом начале в конвейер передаются исходные данные, затем они проходят несколько преобразований, каждое из которых возвращает новую, измененную версию исходных данных. При возникновении непредвиденных ошибок функция `complex_transformation` может возвратить исходные данные `original_data`, тем самым откатив все выполненные преобразования. Это становится возможным, так как ни одно преобразование не затрагивает память, в которой хранятся данные `original_data`.

Это все, что хотелось бы сказать в рамках базовой теории иммутабельности. У вас могли остаться вопросы о том, как правильно использовать иммутабельные данные в более сложных программах. Мы вернемся к этой теме в четвертой главе, когда начнем изучение структур данных более высокого уровня.

2.4.6. Словари

Словарь – это хранилище пар ключ/значение, в качестве которых могут выступать любые термы. Словари в Elixir используются в двух случаях: для поддержки динамических структур ключ/значение и для управления простыми записями – несколькими четко определенными связанными друг с другом полями. Теперь рассмотрим каждый их этих случаев отдельно.

Динамические словари

Пустой словарь можно создать с помощью конструкции `%{}`:

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

Синтаксис для словаря, содержащего несколько значений, следующий:

```
iex(2)> squares = %{1 => 1, 2 => 4, 3 => 9}
```

Словарь можно заполнить с помощью функции `Map.new/1`. Она принимает на вход перечисление, каждый элемент которого является кортежем с двумя элементами:

```
iex(3)> squares = Map.new([{1, 1}, {2, 4}, {3, 9}])
%{1 => 1, 2 => 4, 3 => 9}
```

Чтобы получить значение, соответствующее указанному ключу, можно воспользоваться следующим методом:

```
iex(4)> squares[2]
4

iex(5)> squares[4]
nil
```

Результатом второго выражения является значение `nil`, потому что ключу 4 не соответствует ни одно значение.

Похожий результат можно получить, воспользовавшись функцией `Map.get/3`. С первого взгляда, по своему поведению она напоминает оператор `[]`, однако она

позволяет указать значение, которое будет возвращаться всякий раз, когда заданный ключ не найден. Значение по умолчанию можно и не указывать, и тогда будет возвращено значение `nil`:

```
iex(6)> Map.get(squares, 2)
4

iex(7)> Map.get(squares, 4)
nil

iex(8)> Map.get(squares, 4, :not_found)
:not_found
```

Обратите внимание, что результат последнего выражения неоднозначен: он свидетельствует о том, что не существует либо соответствующего заданному ключу значения, либо самого ключа. Если вы хотите докопаться до истины, воспользуйтесь функцией `Map.fetch/2`:

```
iex(9)> Map.fetch(squares, 2)
{:ok, 4}

iex(10)> Map.fetch(squares, 4)
:error
```

Как видно из примера, в случае успеха возвращается кортеж `{:ok, value}`, и можно точно распознать случай, когда ключ не найден.

Иногда имеет смысл сделать так, чтобы код продолжал выполняться, только когда ключ присутствует в словаре, и вызывать ошибку в противном случае. Это можно сделать с помощью функции `Map.fetch!/2`:

```
iex(11)> Map.fetch!(squares, 2)
4

iex(12)> Map.fetch!(squares, 4)
** (KeyError) key 4 not found in: %{1 => 1, 2 => 4, 3 => 9}
       (stdlib) :maps.get(4, %{1 => 1, 2 => 4, 3 => 9})
```

Функция `Map.put/3` добавляет в словарь новый элемент:

```
iex(13)> squares = Map.put(squares, 4, 16)
%{1 => 1, 2 => 4, 3 => 9, 4 => 16}

iex(14)> squares[4]
16
```

Модуль `Map` также содержит множество других полезных функций, таких как `Map.update/4` и `Map.delete/2`. Официальную документацию модуля можно найти по адресу: **https://hexdocs.pm/elixir/Map.html**. Кроме того, словарь является перечисляемой структурой данных, а значит, все функции модуля `Enum` могут использоваться и со словарями.

Структурированные данные

Словари – наиболее подходящий тип для управления структурами данных ключ/значение произвольного размера, но в Elixir они также довольно часто применяются для объединения нескольких полей в единую структуру. Почти то же самое

делают и кортежи, но у словарей есть небольшое преимущество: к полям можно обращаться по имени.

Рассмотрим пример. Создадим словарь с данными о конкретном человеке:

```
iex(1)> bob = %{:name => "Bob", :age => 25, :works_at => "Initech"}
```

Если ключи представить в виде атомов, то код можно немного сократить:

```
iex(2)> bob = %{name: "Bob", age: 25, works_at: "Initech"}
```

Чтобы вывести значение какого-либо поля, используйте оператор `[]`:

```
iex(3)> bob[:works_at]
"Initech"

iex(4)> bob[:non_existent_field]
nil
```

Ключи-атомы имеют особый синтаксис. Например, вот так можно получить значение, соответствующее ключу `:age`:

```
iex(5)> bob.age
25
```

Если попытаться вывести несуществующее поле с этим синтаксисом, то возникнет ошибка:

```
iex(6)> bob.non_existent_field
** (KeyError) key :non_existent_field not found
```

Изменить значение поля можно следующим образом:

```
iex(7)> next_years_bob = %{bob | age: 26}
%{age: 26, name: "Bob", works_at: "Initech"}
```

То же самое справедливо и для изменения нескольких значений:

```
iex(8)> %{bob | age: 26, works_at: "Initrode"}
%{age: 26, name: "Bob", works_at: "Initrode"}
```

Изменять можно только существующие в словаре значения. Именно поэтому такой синтаксис – лучшее решение для работы со словарями, содержащими структурированные данные. Если в имени поля будет допущена опечатка, сразу же возникнет ошибка времени выполнения:

```
iex(9)> %{bob | works_in: "Initech"}
** (KeyError) key :works_in not found
```

Использование словарей для хранения структурированных данных – часто встречающийся в Elixir шаблон проектирования. При создании словаря обычно указываются все его поля с ключами-атомами. Если значение какого-либо поля недоступно, можно установить для него `nil`. Изменить значения в таком словаре можно с помощью приведенного выше синтаксиса, а получить значение определенного поля – с помощью синтаксиса `a_map.some_field`.

Для таких словарей также применимы функции модуля `Map`, включая `Map.put/3` и `Map.fetch/2`, но они более актуальны для случаев, когда словари используются для управления динамическими структурами типа ключ/значение.

2.4.7. Бинарные данные и битовые строки

Бинарные данные – это набор байтов. Создать бинарную последовательность можно при помощи операторов << и >>. В примере ниже создается бинарная последовательность из трех байтов:

```
iex(1)> <<1, 2, 3>>
<<1, 2, 3>>
```

Каждое число – это значение соответствующего байта. Если указать число больше 255, оно усекается до размера байта:

```
iex(2)> <<256>>
<<0>>

iex(3)> <<257>>
<<1>>

iex(4)> <<512>>
<<0>>
```

Вы можете задать размер каждого элемента последовательности и тем самым сообщить компилятору о том, сколько битов для него необходимо использовать:

```
iex(5)> <<257::16>>
<<1, 1>>
```

Данное выражение помещает число 257 в область памяти размером 16 бит последовательно. Результат показывает, что использовано 2 байта, каждый из которых имеет значение 1. Все потому, что двоичное представление числа 257 в 16-битной форме выглядит как 00000001 00000001.

Размер указывается в битах и должен быть кратен 8. В следующем примере приведено создание бинарной последовательности, содержащей два значения размером 4 бита каждое:

```
iex(6)> <<1::4, 15::4>>
<<31>>
```

Полученное в результате значение занимает 1 байт и выведено в нормализованном представлении – 31 (0001 1111).

Если общий размер всех значений не кратен 8, бинарное представление называют битовой строкой (последовательностью битов):

```
iex(7)> <<1::1, 0::1, 1::1>>
<<5::size(3)>>
```

Объединить две бинарные последовательности или битовые строки можно при помощи оператора <>:

```
iex(8)> <<1, 2>> <> <<3, 4>>
<<1, 2, 3, 4>>
```

Над бинарными данными возможны и другие операции, но пока остановимся на этом. Самое важное, что вам необходимо знать, – это то, что они являются последовательностью байтов и лежат в основе организации работы строк.

2.4.8. Строки

Пожалуй, вы удивитесь, но Elixir не предоставляет строкового типа данных. Строки реализуются либо бинарными последовательностями, либо списками.

Бинарные строки

Наиболее часто используемый способ определения строк – это заключение их в привычные всем двойные кавычки:

```
iex(1)> "This is a string"
"This is a string"
```

Результат выглядит как строка, но на самом деле это бинарные данные, то есть последовательность байтов.

Elixir поддерживает встроенные в строки выражения. Такое выражение сразу же вычисляется, а на его место помещается его строковое представление. Чтобы встроить выражение в строку, воспользуйтесь комбинацией #{}:

```
iex(2)> "Embedded expression: #{3 + 0.14}"
"Embedded expression: 3.14"
```

Классический способ экранирования символом обратного слеша (\) работает привычным образом:

```
iex(3)> "\r \n \" \\"
```

Текст строки может быть многострочным:

```
iex(4)> "
        This is
        a multiline string
        "
```

Elixir предлагает альтернативный синтаксис для объявления строк – так называемые *сигилы*, при которых строка должна быть заключена в комбинацию символов ~s():

```
iex(5)> ~s(This is also a string)
"This is also a string"
```

Такой синтаксис пригодится, если строка содержит кавычки:

```
iex(6)> ~s("Do... or do not. There is no try." -Master Yoda)
"\"Do... or do not. There is no try.\" -Master Yoda"
```

Существует еще одна версия этого синтаксиса – версия с прописной буквой (~S), при использовании которой интерполяция и экранирование не производятся:

```
iex(7)> ~S(Not interpolated #{3 + 0.14})
"Not interpolated \#{3 + 0.14}"
iex(8)> ~S(Not escaped \n)
"Not escaped \\n"
```

Наконец, для форматирования многострочных строк используется синтаксис *встроенных документов*. Встроенные документы открываются и закрываются

тремя двойными кавычками. Завершающие кавычки должны стоять в отдельной строке:

```
iex(9)> """
        Heredoc must end on its own line """
        """
"Heredoc must end on its own line \"\"\"\n"
```

Так как строки являются бинарными данными, их можно объединять, используя оператор конкатенации <>:

```
iex(10)> "String" <> " " <> "concatenation"
"String concatenation"
```

Для работы с бинарными строками существует множество функций, большинство из которых находится в модуле `String` (**https://hexdocs.pm/elixir/String.html**).

Списки символов

Еще один способ объявления строк – заключение их содержимого в одиночные кавычки:

```
iex(1)> 'ABC'
'ABC'
```

Таким образом создаются *списки символов*, представляющие собой список целых чисел, где каждый элемент – это отдельный символ.

Если вы вручную составите список целых чисел, то получите тот же результат:

```
iex(2)> [65, 66, 67]
'ABC'
```

Из примера очевидно, что среда выполнения воспринимает список целых чисел и список символов одинаково. Если список состоит из целых чисел, представляющих печатаемые символы, он выводится на экран в виде строки.

Как и для бинарных строк, для определения символьных строк в Elixir существуют различные варианты синтаксиса:

```
iex(3)> 'Interpolation: #{3 + 0.14}'
'Interpolation: 3.14'

iex(4)> ~c(Character list sigil)
'Character list sigil'

iex(5)> ~C(Unescaped sigil #{3 + 0.14})
'Unescaped sigil \#{3 + 0.14}'

iex(6)> '''
        Heredoc
        '''
'Heredoc\n'
```

Списки символов не совместимы с бинарными строками, и большинство операций модуля `String` с ними не работают. По возможности следует отдавать предпочтение бинарным строкам. Иногда некоторые функции будут работать только со списками символов. Как правило, это может произойти с библиотеками, пол-

ностью написанными на Erlang. В таких случаях бинарную строку следует преобразовать в список символов, используя функцию `String.to_charlist/1`:

```
iex(7)> String.to_charlist("ABC")
'ABC'
```

Чтобы произвести обратное преобразование, воспользуйтесь функцией `List.to_string/1`.

В целом старайтесь всегда использовать бинарные строки, а списки символов включать в свой код только тогда, когда этого требует какая-либо сторонняя библиотека (обычно библиотека Erlang).

2.4.9. Функции первого класса

В Elixir функции считаются объектами первого класса. Их можно присваивать переменным, но это присваивание не означает простой вызов функции и хранение ее результата в переменной. Вместо этого переменной присваивается само определение функции, и чтобы вызвать функцию, достаточно обратиться к переменной.

Рассмотрим пару примеров. Чтобы создать переменную-функцию, используйте конструкцию `fn`:

```
iex(1)> square = fn x ->
          x * x
        end
```

Теперь переменная `square` содержит функцию, возводящую число в квадрат. Поскольку функция не привязана к глобальному имени, она называется *анонимной функцией*, или *лямбда-функцией*.

Обратите внимание, что в определении функции в скобках не указаны параметры. С технической точки зрения, наличие пустых скобок не является ошибкой, но более распространенным решением, поддерживаемым в том числе и инструментом форматирования Elixir, является отказ от использования скобок. Однако список аргументов именованной функции обязательно должен быть заключен в скобки. С первого взгляда это кажется противоречивым, но такое соглашение введено не случайно, и причины этому будут озвучены в третьей главе.

Анонимную функцию можно вызвать, указав имя переменной, после которого должен следовать символ точки (`.`) и аргументы:

```
iex(2)> square.(5)
25
```

> **ПРИМЕЧАНИЕ** Главная цель оператора точка – сделать вызов функции более явным. Встретив выражение `square.(5)` в исходном коде, вы сразу поймете, что вызывается анонимная функция, а выражение `square(5)` – это вызов именованной функции, определенной ранее в модуле. Без оператора точка, чтобы понять, какая функция вызывается, окружающий вызов код будет необходимо пропарсить.

Функции можно хранить в переменных, а значит, они так же могут быть переданы другим функциям в виде аргументов. Обычно это используется для параметризации общей логики клиента. Например, функция `Enum.each/2` реализует

обобщенную итерацию – перебор любого перечисляемого типа, в числе которых списки. Функция `Enum.each/2` принимает два аргумента: перечисляемую структуру данных и анонимную функцию с арностью 1. Она перебирает элементы перечисления и вызывает лямбда-функцию для каждого из них. Клиент предоставляет лямбда-функцию и таким образом определяет, какие операции необходимо проделать над каждым элементом.

В следующем примере функция `Enum.each` выводит элементы списка на экран:

```
iex(3)> print_element = fn x -> IO.puts(x) end
iex(4)> Enum.each(
          [1, 2, 3],
          print_element
        )
1
2
3
:ok
```

Определение анонимной функции

Передача анонимной функции в виде аргумента функции `Enum.each`

Результат выполнения анонимной функции

Результат выполнения функции `Enum.each`

Создавать временную переменную только для того, чтобы передать анонимную функцию в виде аргумента функции `Enum.each`, совсем не обязательно:

```
iex(5)> Enum.each(
          [1, 2, 3],
          fn x -> IO.puts(x) end
        )
1
2
3
```

Передача анонимной функции напрямую

Обратите внимание, анонимная функция передает все аргументы функции `IO.puts` и больше не выполняет никаких действий. В подобных случаях в Elixir существует возможность обращаться к функции напрямую, тем самым делая ее определение компактнее. Вместо `fn x → IO.puts(x) end` можно просто написать `&IO.puts/1`.

Оператор `&`, также известный как *оператор захвата*, принимает полное составное имя функции, включающее имя модуля, имя функции и ее арность, и превращает эту функцию в анонимную, которую можно присвоить переменной. Используйте оператор захвата, чтобы упростить вызов `Enum.each`:

```
iex(6)> Enum.each(
          [1, 2, 3],
          &IO.puts/1
        )
```

Передача анонимной функции, делегирующей функции `IO.puts`

Оператор захвата поможет еще больше упростить определение анонимной функции путем исключения явного объявления параметров. Например, следующее определение:

```
iex(7)> lambda = fn x, y, z -> x * y + z end
```

можно записать в более компактной форме:

```
iex(8)> lambda = &(&1 * &2 + &3)
```

Этот код создает анонимную функцию с арностью 3, на месте каждого аргумента которой стоит комбинация `&n`, где *n* – номер аргумента. Вызвать такую функцию можно обычным способом:

```
iex(9)> lambda.(2, 3, 4)
10
```

Возвращаемое функцией значение 10 эквивалентно 2 * 3 + 4, как и указано в ее определении.

Замыкания

Анонимная функция может ссылаться на любую переменную вне ее области видимости:

```
iex(1)> outside_var = 5
5

iex(2)> my_lambda = fn ->
          IO.puts(outside_var)
        end
```

Анонимная функция ссылается на переменную вне ее области видимости

```
iex(3)> my_lambda.()
5
```

Во время вызова функции `my_lambda` переменная `outside_var` остается доступной. Это явление также называется *замыканием*: ссылаясь на анонимную функцию, вы получаете доступ ко всем используемым ею переменным, даже если они находятся за пределами ее области видимости.

Замыкание всегда требует особого расположения в памяти. Повторная привязка переменной никак не влияет на ранее определенную анонимную функцию, ссылающуюся на то же символьное имя:

```
iex(1)> outside_var = 5
iex(2)> lambda = fn -> IO.puts(outside_var) end
iex(3)> outside_var = 6
iex(4)> lambda.()
5
```

Лямбда-функция занимает текущее положение переменной outside_var в памяти

Повторная привязка переменной не влияет на замыкание

Проверка: значение переменной осталось прежним

Код выше наглядно демонстрирует еще один важный момент. Как правило, после повторной привязки значения 6 к переменной outside_var первоначальная область памяти впоследствии очищается сборщиком мусора. Но так как эта же область (та, в которой хранится значение 5) выделяется для функции `lambda` и функция вызывается в коде, область не доступна для сборщика мусора.

2.4.10. Прочие встроенные типы

В Elixir существует еще несколько типов, которые не будут подробно рассмотрены в этой книге. Но все же стоит упомянуть о них для полноты картины.

- *Ссылка* (*reference*) – практически уникальный тип в среде выполнения BEAM. Создается она вызовом функции `Kernel.make_ref/0` (или `make_ref`). Согласно документации Elixir, ссылка может повториться приблизительно через 2^82 вызовов, но если перезапустить экземпляр BEAM, она будет сгенерирована

снова, поэтому ее уникальность гарантирована только в течение жизненного цикла экземпляра виртуальной машины.

- *Идентификатор процесса* (*pid*) служит для идентификации процесса. Этот тип в основном необходим для организации взаимодействия конкурентных задач. Подробнее идентификаторы процессов будут рассмотрены в пятой главе вместе с процессами Erlang.
- *Идентификатор порта* – важный тип во время работы с портами, представляющий собой механизм Erlang для связи с внешним миром. С помощью портов реализованы запись в файл и чтение из файла, а также взаимодействие с внешними программами. В данной книге идентификаторы портов рассмотрены не будут.

Теперь можно сказать, что список базовых типов Elixir на этом заканчивается. Как видите, система типов языка довольно простая и насчитывает не так много типов данных.

Разумеется, доступны и типы данных более высокого уровня, построенные на основе простых типов и обеспечивающие дополнительную функциональность. Давайте рассмотрим несколько самых важных из них.

2.4.11. Типы данных более высокого уровня

Описанные выше встроенные типы пришли в Elixir из мира Erlang, ведь код, написанный на Elixir, выполняется виртуальной машиной BEAM, поэтому система типов языка обусловлена принципами Erlang. Однако Elixir предоставляет несколько абстракций более высокого уровня, основанных на базовых типах. Наиболее часто используемые из них – это диапазоны (range), ключевые списки, MapSet, дата и время, NaiveDateTime и DateTime.

Диапазоны

Диапазон – это абстракция, представляющая диапазон чисел. Для определения этого типа в Elixir даже имеется специальный синтаксис:

```
iex(1)> range = 1..2
```

Чтобы проверить, находится ли какое-либо число в заданном диапазоне, используйте оператор `in`:

```
iex(2)> 2 in range
true

iex(3)> -1 in range
false
```

Диапазоны считаются перечисляемыми структурами, поэтому для них применимы функции из модуля `Enum`. Несколько разделов назад вы познакомились с функцией `Enum.each/2`, которая перебирает все элементы перечислений. В следующем примере эта функция анализирует каждый элемент диапазона и выводит первые три натуральных числа:

```
iex(4)> Enum.each(
          1..3,
          &IO.puts/1
        )
1
2
3
```

Важно понимать, что диапазон – это не отдельный тип, а абстракция, изнутри представляющая собой словарь с верхней и нижней границами. Знания о подробностях реализации диапазона могут вам и не пригодиться, но помнить о том, что объем занимаемой диапазоном памяти совсем небольшой, независимо от его размера, все же стоит. Диапазон, включающий тысячи элементов, – всего лишь маленький словарь.

Ключевые списки

Ключевой список – это особый список, каждым элементом которого является кортеж из двух элементов, где первый из них – это атом, а второй может быть представлен любым типом данных. Рассмотрим пример:

```
iex(1)> days = [{:monday, 1}, {:tuesday, 2}, {:wednesday, 3}]
```

Elixir позволяет использовать более простой синтаксис для определения ключевых списков:

```
iex(2)> days = [monday: 1, tuesday: 2, wednesday: 3]
```

Обе конструкции реализуют одно и то же – список пар, только, пожалуй, второй выглядит элегантнее.

Ключевые списки часто используются для представления структур ключ/значение небольших размеров с атомами в качестве ключей. В модуле `Keyword` (**https://hexdocs.pm/ elixir/Keyword.html**) вы найдете множество полезных функций для работы с ними. Например, функция `Keyword.get/2` поможет получить соответствующее ключу значение:

```
iex(3)> Keyword.get(days, :monday)
1

iex(4)> Keyword.get(days, :noday)
nil
```

Как и со словарями, для получения значения можно использовать оператор `[]`:

```
iex(5)> days[:tuesday]
2
```

Но не спешите радоваться, вы все еще имеете дело со списками, а значит, данная операция имеет сложность O(n).

Ключевые списки чаще всего используются для предоставления клиентам возможности передавать произвольное количество дополнительных аргументов. К примеру, можно управлять функцией `IO.inspect`, выводящей на консоль строковое представление терма, добавив несколько опций в ключевой список:

```
iex(6)> IO.inspect([100, 200, 300])
[100, 200, 300]
```

Поведение по умолчанию

```
iex(7)> IO.inspect([100, 200, 300], [width: 3])
[100,
 200,
 300]
```

Передача дополнительных опций

Этот шаблон настолько прижился в Elixir, что разрешается опускать квадратные скобки, если последний аргумент является ключевым списком:

```
iex(8)> IO.inspect([100, 200, 300], width: 3, limit: 1)
[100,
 ...]
```

Здесь в функцию `IO.inspect/2` передаются два аргумента – число и ключевой список из двух элементов. Но пример показывает, как нужно имитировать дополнительные аргументы. Ключевой список можно передать функции в качестве последнего аргумента и сделать его по умолчанию пустым:


```
def my_fun(arg1, arg2, opts \\ []) do
  ...
end
```

Теперь клиенты смогут передавать дополнительные значения через последний аргумент. Вы сами можете проверять содержимое аргумента `opts` и реализовывать условную логику в зависимости от того, какие данные были отправлены вызывающим клиентом.

Должно быть, вы уже раздумываете о том, чтобы вместо ключевых списков использовать словари для передачи дополнительных аргументов. Во-первых, ключевой список позволяет хранить несколько значений для одного и того же ключа. Во-вторых, порядок элементов ключевого списка можно изменять, что невозможно со словарями. Наконец, многие функции из стандартных библиотек Elixir и Erlang содержат опции в виде ключевых списков. Вот почему все же лучше придерживаться существующего соглашения и передавать дополнительные параметры с помощью ключевых списков.

MapSet

`MapSet` – это реализация набора уникальных значений, которые могут быть представлены любыми типами.

Посмотрите на следующий пример:

```
iex(1)> days = MapSet.new([:monday, :tuesday, :wednesday])
#MapSet<[:monday, :tuesday, :wednesday]>
```

Создание экземпляра `MapSet`

```
iex(2)> MapSet.member?(days, :monday)
true
```

Проверка вхождения существующего элемента

```
iex(3)> MapSet.member?(days, :noday)
false
```

Проверка вхождения несуществующего элемента

```
iex(4)> days = MapSet.put(days, :thursday)
#MapSet<[:monday, :thursday, :tuesday, :wednesday]>
```

Добавление нового элемента в `MapSet`

Как видите, набором значений можно управлять, используя функции из модуля MapSet. Более подробную информацию о модуле ищите в официальной документации по адресу: **https://hexdocs.pm/elixir/MapSet.html**.

Наборы MapSet также являются перечислениями, поэтому их можно передавать в виде аргументов функциям из модуля Enum:


```
iex(5)> Enum.each(days, &IO.puts/1)
monday
thursday
tuesday
wednesday
```

Как заметно из результата, MapSet не сохраняет порядок элементов.

Дата и время

В Elixir существует несколько модулей для работы с типами данных даты и времени: Date, Time, DateTime и NaiveDateTime.

Даты создаются с помощью сигила ~D. В следующем примере создается дата 31 января 2018 г.:

```
iex(1)> date = ~D[2018-01-31]
~D[2018-01-31]
```

После этого можно вывести значения отдельных полей:


```
iex(2)> date.year
2018

iex(3)> date.month
1
```

Аналогично, с помощью сигила ~T, объявляется тип времени, при этом указываются часы, минуты, секунды и микросекунды:

```
iex(1)> time = ~T[11:59:12.00007]

iex(2)> time.hour
11

iex(3)> time.minute
59
```

В модулях Date (https://hexdocs.pm/ elixir/Date.html) и Time (https://hexdocs.pm/ elixir/Time.html) вы сможете найти еще множество полезных функций.

В дополнение к этим двум типам представить дату и время помогут функции из модулей NaiveDateTime и DateTime. Дата и время без часового пояса (naive) создаются при помощи сигила ~N:

```
iex(1)> naive_datetime = ~N[2018-01-31 11:59:12.000007]

iex(2)> naive_datetime.year
2018

iex(3)> naive_datetime.hour
11
```

Модуль DateTime понадобится для работы с датой и временем с указанием часового пояса. В отличие от предыдущих типов, этому не соответствует никакой сигил. Для создания этого типа используются функции модуля DateTime:

```
iex(4)> datetime = DateTime.from_naive!(naive_datetime, "Etc/UTC")

iex(5)> datetime.year
2018

iex(6)> datetime.hour
11

iex(7)> datetime.time_zone
"Etc/UTC"
```

Стоит изучить документацию этих модулей, находящуюся по адресам **https://hexdocs.pm/elixir/NaiveDateTime.html** и **https://hexdocs.pm/elixir/DateTime.html**, чтобы научиться работать с представленными выше типами данных.

2.4.12. Списки ввода-вывода

Список ввода-вывода – особый тип списка, применяемый для постепенного формирования результата, который затем передается в устройство ввода-вывода – сеть или файл. Элементами списка ввода-вывода могут быть:

- целое число от 0 до 255;
- бинарные данные;
- другой список ввода-вывода.

Другими словами, списки ввода-вывода – это глубоко вложенные структуры, конечными элементами которых являются байты (или бинарные данные, которые, по сути, являются последовательностью байтов). Например, вот так выглядит текст «Hello, world», представленный сложным списком:

```
iex(1)> iolist = [['H', 'e'], "llo,"], " worl", "d!"]
```

Обратите внимание, как в примере списки символов и бинарные строки вместе образуют глубоко вложенный список.

Многие функции ввода-вывода могут работать напрямую с такими данными. Например, попробуем вывести их на экран:

```
iex(2)> IO.puts(iolist)
Hello, world!
```

Внутри структура обрабатывается и превращается в удобочитаемый результат. Вы получите тот же эффект, передав список ввода-вывода в файл или сетевой сокет.

Списки ввода-вывода пригодятся для формирования потока байтов. Обычные списки не подойдут в этом случае, потому что добавление элемента в конец списка – операция со сложностью O(n). Благодаря вложениям сложность этой операции для списков ввода-вывода – O(1). Рассмотрим это на примере:

```
iex(3)> iolist = []    ← Инициализация списка ввода-вывода
        iolist = [iolist, "This"]
        iolist = [iolist, " is"]
        iolist = [iolist, " an"]
        iolist = [iolist, " IO list."]
```

Добавление элементов в конец списка

```
[[[[[], "This"], " is"], " an"], " IO list."]   <-- Конечный список
```

При добавлении элемента в конец списка создается новый список с двумя элементами – предыдущей версией списка ввода-вывода и новым элементом. Каждая такая операция имеет сложность O(1), что очень эффективно. Разумеется, эти данные затем можно передать в функцию ввода-вывода:

```
iex(4)> IO.puts(iolist)
This is an IO list.
```

На этом описание системы типов подходит к концу. Теперь вы знакомы с основными типами и далее по ходу изучения книги расширите свои знания. В следующем разделе мы поговорим об операторах Elixir.

2.5. Операторы

На протяжении всей главы вы сталкивались в коде с различными операторами, и пришло время рассмотреть наиболее часто используемые из них. Большинство операторов определено в модуле `Kernel`, в документации к которому вы найдете более подробное их описание.

Начнем, пожалуй, с арифметических операторов – это всем знакомые `+`, `-`, `*` и `/`. Работают они практически так же, как и в других языках, за исключением оператора деления, который всегда возвращает вещественное число, как было показано в начале этой главы.

Операторы сравнения, приведенные далее в таблице, тоже работают привычным образом:

Таблица 2.1. Операторы сравнения

Оператор	Описание
`===`, `!==`	Строгое равенство/неравенство
`==`, `!=`	Равенство/неравенство
`<`, `>`, `≤`, `≥`	Меньше, больше, меньше или равно, больше или равно

Единственное, на что стоит обратить внимание, – это разница между обычным и строгим сравнением. Строгое используется только для сравнения целого и вещественного чисел:

```
iex(1)> 1 == 1.0    <-- Обычное сравнение
true

iex(2)> 1 === 1.0   <-- Строгое сравнение
false
```

Логические операторы (`and`, `or` и `not`), которые уже упоминались ранее в этой главе, предназначены для работы с логическими атомами.

В отличие от логических операторов, операторы, выполняющиеся по короткой схеме, опираются на понятие истинности: все, кроме атомов `false` и `nil`, считается истинным. Оператор `&&` возвращает второе выражение, только если первое не является ложным. Оператор `||` возвращает первое выражение, если оно истинно,

а в противном случае он возвращает второе выражение. Унарный оператор ! возвращает false, если значение истинно, и true в обратном случае.

Представленные в данном разделе операторы – самые распространенные, но, кроме них, доступны и другие (например, оператор конвейера |>).

Операторы – это функции?

Стоит отметить, что многие операторы Elixir на самом деле являются функциями. Например, a+b можно спокойно заменить на Kernel.+(a,b). Такой код выглядит непривлекательно, но из операторов-функций можно извлечь выгоду, превратив их в анонимные функции. Например, создать лямбда-функцию с арностью 2, складывающую два числа, вызвав &Kernel.+/2 или более короткий вариант &+/2. В третьей главе будет рассмотрено, как подобные функции могут взаимодействовать с различными перечисляемыми типами и с функциями потоков.

Итак, небольшое путешествие в мир основ Elixir практически подошло к концу, наша последняя остановка – макросы.

2.6. Макросы

Макросы по праву считаются наиболее сильной возможностью языка Elixir в сравнении с Erlang. Они позволяют выполнять эффективные преобразования кода во время компиляции, тем самым уменьшая количество повторений и предоставляя изящные мини-DSL-конструкции.

Макрос – довольно сложное явление, заслуживающее целой отдельной книги. Поскольку данная книга в основном направлена на изучение среды выполнения и виртуальной машины BEAM, а макросы – это нечто вроде дополнительной возможности языка, которой не стоит увлекаться, останавливаться на них подробно мы не будем. Однако вам следует хотя бы обзавестись общим представлением о принципе их работы, так как они лежат в основе большей части функционала Elixir.

Макрос представляет собой код Elixir, способный изменить семантику передаваемого ему кода. Макрос всегда вызывается на этапе компиляции, получая на входе проанализированный код Elixir и возвращая альтернативную его версию на выходе.

Рассмотрим это на примере. unless (аналог if not) – простой макрос Elixir:

```
unless some_expression do
  block_1
else
  block_2
end
```

unless – это не ключевое слово, а макрос (то есть функция Elixir), преобразующая следующий после него код в нечто подобное:

```
if some_expression do
  block_2
```

```
else
  block_1
end
```

Макросы языка C не справятся с таким преобразованием, ведь код выражения может быть довольно сложным и заключенным в несколько пар скобок. А макросы Elixir (созданные по подобию макросов Lisp) позволяют работать с уже проанализированным кодом и иметь доступ к выражению и к двум блокам, находящимся в разных переменных.

На самом деле благодаря макросам многие элементы языка Elixir написаны на Elixir, включая конструкции `unless` и `if`, а также `defmodule` и `def`. В то время как в других языках для достижения такого функционала используются ключевые слова, в Elixir, имеющем довольно маленькое языковое ядро, он выстраивается за счет макросов.

Главное, что вам нужно запомнить, – это то, что макросы преобразуют код во время компиляции.

> **Специальные формы**
>
> Некоторые конструкции языка воспринимаются компилятором по-особенному. Они называются *специальными формами* (**https://hexdocs.pm/elixir/Kernel.SpecialForms.html**). Среди них можно выделить синтаксис захвата `&(…)`, генераторы `for` (глава 3), конструкцию `receive` (глава 5) и блоки `try` (глава 8).

Для получения более подробной информации вам следует обратиться к официальному руководству по метапрограммированию (**https://elixir-lang.org/getting-started/meta/quote-and-unquote.html**). На этом обзор основ языка окончен, но напоследок рассмотрим важнейшие аспекты среды выполнения Elixir.

2.7. Среда выполнения

Как отмечалось ранее, средой выполнения Elixir является экземпляр виртуальной машины BEAM. По завершении компиляции и после запуска системы в игру вступает Erlang. Чтобы понимать, как именно функционирует система, очень важно ознакомиться с подробностями работы виртуальной машины.

Прежде всего определим значение модулей в среде выполнения.

2.7.1. Модули и функции в среде выполнения

Каким бы образом вы ни запустили среду выполнения, для экземпляра BEAM всегда запускается процесс ОС, и весь код работает в рамках этого процесса. Это актуально даже для интерактивной оболочки `iex`. Данный процесс ОС называется `beam`.

После запуска системы вы выполняете код, вызывая функции из модулей. Но как среда выполнения получает доступ к коду? Виртуальная машина отслеживает все загруженные в память модули, и когда вы вызываете функцию, BEAM сначала проверяет, подгружен ли он. Если да, выполняется код вызываемой

функции, в противном случае виртуальная машина пытается отыскать скомпилированный файл модуля (байт-код) на диске и затем подгрузить его и выполнить код функции.

> **ПРИМЕЧАНИЕ** По описанию процесса выше становится очевидно, что каждый скомпилированный файл модуля находится в отдельном файле. Он имеет расширение .beam (Bogdan/ Björn's Erlang Abstract Machine), а имя файла совпадает с именем модуля.

Имена модулей и атомы

Давайте вспомним, как определяются модули:

```
defmodule Geometry do
  ...
end
```

Также из предыдущих разделов вам известно, что `Geometry` – это псевдоним имени `Elixir.Geometry`:

```
iex(1)> Geometry == :"Elixir.Geometry"
true
```

Это не случайно. При компиляции исходного кода, содержащего модуль `Geometry`, сгенерированный на диске файл получает название `Elixir.Geometry.beam` независимо от имени файла с исходным кодом. На самом деле, когда в файле с исходным кодом определено несколько модулей, компилятор также создает несколько файлов .beam, соответствующих этим модулям. Можете проверить это самостоятельно, вызвав компилятор Elixir (`elixirc`) в командной строке:

```
$ elixirc source.ex
```

Позаботьтесь о том, чтобы в файле `source.ex` было объявлено несколько модулей. Если в вашем коде не будет синтаксических ошибок, то вы увидите, как на диске сгенерируется несколько файлов `.beam`.

Во время выполнения кода имена модулей заменяются на псевдонимы, а псевдонимы – это атомы. Когда вы вызываете функцию первый раз, BEAM выполняет поиск соответствующего файла сначала в текущей папке, а потом в путях выполнения кода.

Когда вы запускаете BEAM через инструменты Elixir (`IEx` и т. п.), некоторые пути выполнения заранее определены. Чтобы добавить новые пути, укажите их после параметра `-pa`:

```
$ iex -pa my/code/path -pa another/code/path
```

Проверить, какие пути используются во время выполнения, можно с помощью функции Erlang `:code.get_path`.

Если модуль уже подгружен, среда выполнения не производит его поиск на диске. Чтобы модули подгружались автоматически, при запуске оболочки выполните следующую команду:

```
$ iex my_source.ex
```

Файл с исходным кодом скомпилируется, и все сгенерированные модули будут незамедлительно подгружены. Заметьте, в данном случае файлы с расширением

.beam не сохраняются: инструмент IEx выполняет их компиляцию в оперативной памяти.

Схожим образом можно определять модули прямо в оболочке:

```
iex(1)> defmodule MyModule do
          def my_fun, do: :ok
        end
```

Генерация кода в оперативной памяти и загрузка модуля

```
iex(2)> MyModule.my_fun
:ok
```

Здесь байт-код по тому же принципу не сохраняется на диске.

Модули Erlang

Прежде вы ознакомились с синтаксисом вызова функций из модулей, полностью написанных на Erlang:

```
:code.get_path
```

Вызов функции get_path из модуля Erlang :code

В Erlang модули также находятся в соответствии с атомами. Где-то на диске хранится файл с именем code.beam, содержащий скомпилированный код модуля :code. Файлы Erlang имеют простые имена, поэтому при вызове функции используется такой синтаксис. По правде говоря, модули Erlang и модули Elixir – это одно и то же, просто имена модулей Elixir выглядят привлекательнее (Elixir.MyModule).

Вы можете создавать модули с простыми именами и в Elixir (хотя делать это не рекомендуется):

```
defmodule :my_module do
  ...
end
```

После компиляции файла с таким определением будет создан файл my_module.beam.

Итак, главное, что вам следует вынести из последних абзацев, – это то, что на этапе выполнения программы имена модулей являются атомами и на диске создается файл *xyz*.beam, где *xyz* – это псевдоним в развернутой форме (например, Elixir.MyModule для модуля с именем MyModule).

Динамический вызов функций

Логично было бы продолжить предыдущую тему обсуждением возможности динамического вызова функций во время выполнения программы. Это можно осуществить с помощью функции Kernel.apply/3:

```
iex(1)> apply(IO, :puts, ["Dynamic function call."])
Dynamic function call.
```

В функцию Kernel.apply/3 передаются три аргумента: атом модуля, атом функции и список передаваемых функции аргументов. Три аргумента функции Kernel.apply/3, часто называемых МФА (MFA – модуль, функция, аргументы), содержат информацию, необходимую для вызова экспортируемой (публичной) функции. Kernel.apply/3 может быть полезна, когда во время выполнения программы вы принимаете решение о том, какую функцию следует вызвать.

2.7.2. Запуск среды выполнения

Существует несколько способов запуска BEAM. До этого момента вы пользовались оболочкой IEx и пока продолжите делать то же самое. Но все же стоит пробежаться по всем возможным способам запуска среды выполнения.

Интерактивная оболочка

Запуская интерактивную оболочку, вы запускаете экземпляр BEAM, а оболочка берет над ним управление. Оболочка принимает входные данные, интерпретирует их и выводит результат на экран.

Важно понимать, что входные данные именно интерпретируются, а это значит, что производительность такого метода будет ниже, чем у того же скомпилированного кода. И чаще всего это не проблема, так как оболочка обычно используется для практики языка. Не стоит даже пытаться измерять производительность напрямую через IEx.

С другой стороны, код модулей всегда компилируется, и если определить модуль в оболочке, он будет скомпилирован и загружен в память. Производительность при этом не пострадает.

Запуск скриптов

Команда `elixir` используется для запуска отдельных файлов с исходным кодом на Elixir. Она имеет следующий синтаксис:

```
$ elixir my_source.ex
```

После выполнения команды происходят такие действия:

- запускается экземпляр BEAM;
- выполняется компиляция файла `my_source.ex` в оперативной памяти, а получившиеся модули подгружаются в виртуальную машину, при этом на диске не генерируются файлы с расширением .beam;
- интерпретируется код, следующий за определением модуля;
- BEAM завершает работу по окончании всех действий.

Обычно это и происходит при запуске скриптов. Рекомендуется создавать такие скрипты с расширением .exs, где s – признак того, что файл является скриптом.

В листинге ниже приведен код простого скрипта Elixir.

Листинг 2.5 ❖ Скрипт Elixir

```
defmodule MyModule do
  def run do
    IO.puts("Called MyModule.run")
  end
end

MyModule.run    <-- Код за пределами модуля выполняется незамедлительно
```

Выполнить такой скрипт можно прямо в командной строке:

```
$ elixir script.exs
```

Данная команда сначала выполняет компиляцию модуля `MyModule` в памяти, а затем запускает его вызовом `MyModule.run`. Когда связанный с вызовом код выполнится, экземпляр BEAM завершается.

Если вам необходимо, чтобы экземпляр продолжил существовать, укажите при выполнении команды параметр `--no-halt`:

```
$ elixir --no-halt script.exs
```

В основном это может понадобиться в ситуации, когда главный код (код за пределами модуля) реализует конкурентные задачи, на которых строится вся логика программы. Обычно главный вызов и экземпляр BEAM завершаются перед началом выполнения конкурентных задач, и они остаются невыполненными. Указав параметр `--no-halt`, вы дадите команду поддерживать систему в рабочем состоянии.

Инструмент Mix

Инструмент `mix` применяется для управления проектами, содержащими несколько файлов с исходным кодом. `Mix` поможет сформировать систему, полностью готовую к промышленному использованию.

Чтобы создать пустой проект mix, выполните команду `mix new project_name`:

```
$ mix new my_project
```

После ввода команды будет создана новая папка с именем `my_project`, содержащая несколько подпапок и файлов. Теперь перейдите в каталог my_project и скомпилируйте весь проект:

```
$ cd my_project
$ mix compile
```

```
$ mix compile
Compiling 1 file (.ex)
Generated my_project app
```

Скомпилируются все файлы из папки lib, и получившиеся файлы с расширением .beam будут помещены в папку ebin.

Вы можете выполнять различные команды `mix` с одним и тем же проектом. Представим, что генератор создал модуль `MyProject` с единственной функцией `hello/0`. Вызовем эту функцию с помощью команды `mix run`:

```
$ mix run -e "IO.puts(MyProject.hello())"
world
```

Генератор также создал пару тестов, которые можно выполнить командой `mix test`:

```
$ mix test
..

Finished in 0.03 seconds
2 tests, 0 failure
```

Независимо от того, как вы запустите проект mix, папка ebin (где будут храниться файлы с расширением .beam) будет находиться в пути загрузки, чтобы виртуальная машина смогла найти ваши модули.

Как только вы начнете разрабатывать сложные системы, вы будете часто прибегать к использованию инструмента `mix`. А пока не будем сильно углубляться в эту тему.

Выводы

- Код на Elixir состоит из модулей и функций.
- Elixir – динамический язык. Тип переменной определяется по значению, которое она содержит.
- Данные иммутабельны, то есть не могут быть изменены. Функция может возвращать измененную версию входных данных, которая хранится в другой области памяти. Измененная версия и исходные данные имеют несколько общих ячеек памяти, когда это возможно.
- Наиболее значимые примитивные типы данных – это числа, атомы и бинарные данные.
- Логического типа не существует. Вместо него используются атомы `true` и `false`.
- Неопределенного значения нет. Есть атом `nil`.
- Строковый тип не представлен. Для этого используются либо бинарные последовательности (рекомендуется), либо списки (в случае необходимости).
- Имеются встроенные сложные типы – кортежи, списки и словари. Кортежи используются для группировки небольших наборов полей фиксированного размера, списки помогают управлять коллекциями с переменным размером, а словари – это структура данных ключ/значение.
- Диапазоны, ключевые списки, `MapSet`, дата и время, `NaiveDateTime` и `DateTime` – это абстракции, построенные на основе существующих встроенных типов.
- Функции считаются объектами первого класса.
- Имена модулей на самом деле представлены атомами (или псевдонимами), они совпадают с именами файлов с расширением .beam.
- Существует несколько способов запуска программ: оболочка `iex`, команда `elixir` и инструмент `mix`.

Глава 3

Поток управления

В главе рассматривается:

- сопоставление с образцом;
- функции с несколькими предложениями;
- использование условных выражений;
- работа с циклами.

В прошлой главе вы познакомились с основами Elixir, теперь самое время погрузиться в изучение таких типовых идиом языка, как условные выражения и циклы, которые, как вы увидите, в Elixir работают иначе, чем в наиболее современных императивных языках программирования.

Классические условные конструкции `if` и `case` чаще всего заменяются функциями с несколькими предложениями, а циклов `while` вообще не существует. Несмотря на это, на Elixir все равно можно решить задачи любой сложности, и код этого решения будет не сложнее кода стандартного решения на объектно-ориентированном языке.

Такой подход достаточно сильно отличается от привычного вам, поэтому условные выражения и циклы были выведены мной в отдельную главу. Но, прежде чем перейти к более подробному их изучению, стоит обратить особое внимание на то, что обеспечивает их реализацию, а именно – сопоставление с образцом.

3.1. Сопоставление с образцом

Как было упомянуто во второй главе, оператор `=` не является оператором присваивания. Выражение `a = 1` означает, что к переменной `a` привязывается значение `1`. Оператор `=` называется *оператором сопоставления*, а приведенное в качестве примера выражение – *сопоставлением с образцом*.

Сопоставление с образцом – одна из основных конструкций Elixir, значительно упрощающая работу со сложными переменными (кортежами, списками и др.), а также позволяющая создавать емкие условные выражения и циклы декларативного типа. Понять это в полной мере вам удастся к концу этой главы, а в данном разделе мы поговорим о том, как устроено сопоставление с образцом изнутри.

Начнем с оператора сопоставления.

3.1.1. Оператор сопоставления

До этого момента вы уже наблюдали простейшие случаи использования оператора сопоставления:

```
iex(1)> person = {"Bob", 25}
```

Смысл таких выражений воспринимался вами как нечто, схожее присваиванию, но на самом деле все не так просто. Во время выполнения левая сторона выражения сопоставляется с правой. Левую сторону принято называть *образцом*, в то время как правая представляет собой терм Elixir.

В данном примере переменная `person` сопоставляется с термом `{"Bob", 25}`. Переменная всегда соответствует терму, значение которого *привязывается* к ней. Теперь предлагаю перейти от сухой теории к более сложному случаю – использованию оператора сопоставления с кортежами.

3.1.2. Сопоставление кортежей

Рассмотрим простой пример сопоставления кортежей:

```
iex(1)> {name, age} = {"Bob", 25}
```

В данном выражении терм справа – кортеж с двумя элементами. Во время выполнения выражения к переменным `name` и `age` привязываются соответствующие значения кортежа. Проверим, так ли это:

```
iex(2)> name
"Bob"

iex(3)> age
25
```

Это может пригодиться в том случае, когда функция возвращает кортеж и вам необходимо привязать значение каждого элемента этого кортежа к отдельным переменным. В следующем примере для получения текущей даты и времени вызывается функция Erlang `:calendar.local_time/0`:

```
iex(4)> {date, time} = :calendar.local_time()
```

Дата и время, в свою очередь, тоже являются кортежами:

```
iex(5)> {year, month, day} = date
iex(6)> {hour, minute, second} = time
```

А что, если выражение с правой стороны не соответствует образцу? Сопоставление не удастся, и будет выведена ошибка:

```
iex(7)> {name, age} = "can't match"
** (MatchError) no match of right hand side value: "can't match"
```

> **ПРИМЕЧАНИЕ** Механизм обработки ошибок будет рассмотрен далее в восьмой главе. Пока что достаточно сказать, что происходит нечто, подобное обработке исключений в популярных языках: при возникновении ошибки управление передается вверх по цепочке вызовов заранее определенному обработчику (если он присутствует), который перехватывает ошибку.

Напоследок стоит отметить, что выражение сопоставления, как и любое другое выражение, всегда возвращает значение. Результатом является терм, находящийся справа:

```
iex(8)> {name, age} = {"Bob", 25}  <-- Выражение сопоставления
{"Bob", 25}  <-- Результат сопоставления
```

Сопоставление не ограничивается представлением элементов кортежа отдельными переменными. Как ни странно, с левой стороны выражения сопоставления могут стоять и константы.

3.1.3. Сопоставление с константой

Образец слева также может содержать константы:

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

Как вы уже знаете, оператор сопоставления = соотносит терм справа и образец слева. В примере выше терм 1 сопоставляется с образцом 1. Очевидно, ошибки здесь быть не может, и результатом всего выражения является терм справа.

Данный пример не имеет практической ценности, но он наглядно показывает то, что слева от оператора = может стоять константа, что доказывает, что оператор = не является оператором присваивания.

Константы представляют особый интерес при реализации сложных сопоставлений. Как вы уже знаете, кортежи принято использовать для группировки различных полей одной записи. В следующем примере создается кортеж, содержащий имя и возраст человека:

```
iex(2)> person = {:person, "Bob", 25}
```

Первый элемент – это атом-константа `:person`, обозначающий, что кортеж содержит информацию о человеке. Зная это, впоследствии вы сможете вывести его персональные данные:

```
iex(3)> {:person, name, age} = person
{:person, "Bob", 25}
```

Терм справа представляет собой кортеж с тремя элементами, первый из которых имеет значение `:person`. После сопоставления остальные два элемента кортежа привязываются к переменным `name` и `age`, что можно легко проверить:

```
iex(4)> name
"Bob"

iex(5)> age
25
```

Эта идиома довольно часто встречается в Elixir.

Многие функции Elixir и Erlang возвращают либо `{:ok, result}`, либо `{:error, reason}`. Представьте, что ваша система зависит от файла конфигурации, соответственно, он должен всегда находиться в доступе. Содержимое файла можно считать с помощью функции `File.read/1`:

```
{:ok, contents} = File.read("my_app.config")
```

В этой строке с кодом выполняются целых три действия:

- попытка открыть и считать файл my_app.config;
- при успешной попытке – извлечение содержимого файла в переменную `contents`;
- при неудачной попытке – вывод ошибки, поскольку результат выполнения функции – это кортеж вида `{:error, reason}`, и его успешное сопоставление с кортежем `{:ok, contents}` невозможно.

Используя константы в качестве образцов, вы делаете выражение сопоставления более компактным и назначаете правой части определенное значение.

3.1.4. Переменные в качестве образцов

Если в образце присутствует переменная, она всегда приводится в соответствие с термом, стоящим справа, и связывается с его значением.

Бывает, что само значение терма вас не интересует, но все равно требуется провести сопоставление. Например, чтобы получить текущее время, можно использовать функцию `:calendar.local_time/0`, возвращающую кортеж `{date, time}`. Допустим, дата не представляет для вас интереса, и хранить ее в отдельной переменной незачем. В подобных случаях очень выручает *анонимная переменная* (_):

```
iex(1)> {_, time} = :calendar.local_time()

iex(2)> time
{20, 44, 18}
```

Когда дело доходит до сопоставления, анонимная переменная ведет себя точно так же, как и именованная: она сопоставляется с термом. Однако значение терма ни к чему не привязывается.

Анонимной переменной можно добавить описательное имя после знака подчеркивания:

```
iex(1)> {_date, time} = :calendar.local_time()
```

Переменная `_date` считается анонимной, потому что ее имя начинается со знака подчеркивания. С технической точки зрения, такие переменные можно использовать и в последующем коде программы, но компилятор выведет предупреждение.

Образцы могут иметь произвольные вложения. Возьмем тот же пример, но в этот раз, скажем, необходимо узнать, который час:

```
iex(3)> {_, {hour, _, _}} = :calendar.local_time()

iex(4)> hour
20
```

В пределах образца к переменной можно обращаться несколько раз. В выражениях ниже на выходе ожидаем получить триплет RGB с одинаковыми значениями компонентов:

```
iex(5)> {amount, amount, amount} = {127, 127, 127}
{127, 127, 127}

iex(6)> {amount, amount, amount} = {127, 127, 1}
** (MatchError) no match of right hand side value: {127, 127, 1}
```

Сопоставление с кортежем с тремя одинаковыми элементами

Элементы кортежа различаются, и возникает ошибка

В некоторых случаях вам может понадобиться выполнить сопоставление с содержимым какой-либо переменной. Для этого в Elixir есть специальный *фиксирующий оператор* (^), использование которого лучше всего рассмотреть на примере:

```
iex(7)> expected_name = "Bob"
"Bob"
iex(8)> {^expected_name, _} = {"Bob", 25}
{"Bob", 25}
iex(9)> {^expected_name, _} = {"Alice", 30}
** (MatchError) no match of right hand side value: {"Alice", 30}
```

Сопоставление с правой частью и привязка ее значения к переменной expected_name

Сопоставление с содержимым переменной expected_name

Использование переменной `^expected_name` в составе образца предполагает, что ее *значение* должно находиться в соответствующей позиции в терме справа. Вы бы получили тот же результат, если бы в образце стоял кортеж с уже заданным значением первого элемента `{"Bob", _}`. Во втором сопоставлении кортеж справа не соответствует образцу, поэтому возникает ошибка.

Обратите внимание, фиксирующий оператор не привязывает значение к переменной. Ожидается, что переменная уже связана с каким-либо значением, и вы выполняете сопоставление с ним. Данный подход чаще всего используется, когда необходимо прописать образец во время выполнения программы.

3.1.5. Сопоставление списков

Механизм сопоставления списков работает похожим образом. Сопоставим два списка из трех элементов:

```
iex(1)> [first, second, third] = [1, 2, 3]
[1, 2, 3]
```

Ранее упомянутые способы работы с образцами применимы и для сопоставления списков:

```
[1,       second,   third] = [1, 2, 3]
[first,   first,    first] = [1, 1, 1]
[first,   second,   _    ] = [1, 2, 3]
[^first,  second,   _    ] = [1, 2, 3]
```

Первым элементом должна быть единица

Все элементы должны быть одинаковыми

Кортеж должен включать третий элемент, значение которого может быть любым

Первый элемент должен иметь то же значение, что и переменная first

Сопоставление списков чаще всего основывается на присущей им рекурсивности. Как уже говорилось во второй главе, любой непустой список – это не что иное, как рекурсивная структура, которую можно представить в виде пары «голова–хвост» (`[head | tail]`). Выполняя сопоставление с образцом, можно поместить содержимое каждого из этих двух элементов в отдельные переменные:

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

iex(4)> head
1

iex(5)> tail
[2, 3]
```

Если вам необходимо получить только один элемент этой пары, используйте анонимную переменную. Вот пример вычисления наименьшего элемента списка:

```
iex(6)> [min | _] = Enum.sort([3,2,1])
iex(7)> min
1
```

Сначала список сортируется, затем образец `[min | _]` вытаскивает только голову отсортированного списка. Более элегантно этот пример выглядел бы с использованием функции `hd`, о которой говорилось во второй главе. Образец `[head | _]` больше подходит для сопоставления с аргументами функции, и вы убедитесь в этом в разделе 3.2.

3.1.6. Сопоставление словарей

Для сопоставления словарей используйте следующий синтаксис:

```
iex(1)> %{name: name, age: age} = %{name: "Bob", age: 25}
%{age: 25, name: "Bob"}

iex(2)> name
"Bob"

iex(3)> age
25
```

При этом образец не обязательно должен содержать все ключи правого терма:


```
iex(4)> %{age: age} =  %{name: "Bob", age: 25}

iex(5)> age
25
```

Это правило частичного соответствия может вызвать у вас диссонанс. Словари нередко используются для представления структурированных данных, и тогда могут быть особо интересны лишь отдельные поля. В предыдущем примере было необходимо извлечь только поле `age`, игнорируя все остальное, и правило частичного соответствия позволяет это сделать.

Разумеется, если образец содержит ключи, отличные от ключей терма справа, то сопоставление не удастся:

```
iex(6)> %{age: age, works_at: works_at} = %{name: "Bob", age: 25}
** (MatchError) no match of right hand side value
```

3.1.7. Сопоставление с битовыми строками и бинарными данными

Битовым строкам и бинарным данным в книге не уделено много внимания, однако стоит привести простой пример синтаксиса их сопоставления. Как вы уже знаете, *битовая строка* – это набор битов, а *бинарные данные* – это частный случай бинарной строки, размер которой кратен размеру байта.

Чтобы выполнить сопоставление с бинарной последовательностью, используйте подобный синтаксис:

```
iex(1)> binary = <<1, 2, 3>>
<<1, 2, 3>>

iex(2)> <<b1, b2, b3>> = binary    ← Сопоставление с бинарной последовательностью
<<1, 2, 3>>

iex(3)> b1
1

iex(4)> b2
2

iex(5)> b3
3
```

В данном примере приводится сопоставление с бинарной последовательностью из трех байтов, при этом содержимое каждого байта помещается в отдельную переменную.

Теперь попробуем разбить последовательность на две части, поместив содержимое первого байта в одну переменную, а оставшуюся часть последовательности – в другую:

```
iex(6)> <<b1, rest :: binary>> = binary
<<1, 2, 3>>

iex(7)> b1
1

iex(8)> rest
<<2, 3>>
```

Запись `rest::binary` означает, что последовательность может быть произвольного размера. Извлечь можно даже отдельные биты или группы битов. В следующем примере один байт разбивается на два 4-битных значения:

```
iex(9)> <<a :: 4, b :: 4>> = << 155 >>
<< 155 >>

iex(10)> a
9

iex(11)> b
11
```

Образец `a::4` показывает, что значение переменной должно быть 4-битным. В этом примере первые четыре бита бинарного представления помещаются в переменную `a`, а вторые четыре бита – в переменную `b`. Так как в двоичном виде число 155 записывается как 10011011, значения равны 9 (1001) и 11 (1011) соответственно.

Сопоставление с битовыми строками и бинарными данными имеет огромную практическую ценность при парсинге упакованного двоичного кода, полученного из файла, со внешнего устройства или переданного по сети. Сопоставление с образцом поможет аккуратно выделить из таких данных отдельные биты и байты.

Как отмечалось ранее, данная возможность в рамках книги использована не будет. Однако она может помочь вам решить определенные задачи в своих собственных разработках.

Сопоставление с битовыми строками

Надеюсь, вы помните, что строки являются бинарными данными, а значит, путем сопоставления можно извлечь отдельные биты и байты из строки:

```
iex(13)> <<b1, b2, b3>> = "ABC"
"ABC"

iex(13)> b1
65

iex(14)> b2
66

iex(15)> b3
67
```

Переменные `b1`, `b2`, и `b3` содержат соответствующие байты строки, с которой производится сопоставление. Это решение не самое удачное, особенно для работы со строками Unicode. Извлекать отдельные символы лучше всего с помощью функций модуля `String`.

Более привлекательный способ – сопоставление с началом строки:

```
iex(16)> command = "ping www.example.com"
"ping www.example.com"

iex(17)> "ping " <> url = command    <-- Сопоставление со строкой
"ping www.example.com"

iex(18)> url
"www.example.com"
```

В данном примере создается строка, содержащая команду `ping`. Запись `"ping " <> url = command` означает, что переменная `command` – это бинарная строка, начинающаяся с `"ping "`. В случае успешного сопоставления оставшаяся часть строки привязывается к переменной `url`.

3.1.8. Сложные сопоставления

Сложные сопоставления уже встречались вам ранее, а теперь самое время рассмотреть их поближе. Приведу пример такого сопоставления с образцом, имеющим произвольное вложение:

```
iex(1)> [_, {name, _}, _] = [{"Bob", 25}, {"Alice", 30}, {"John", 35}]
```

Терм, с которым производится сопоставление, представляет собой список из трех элементов, каждый из которых является кортежем с данными о человеке и содержит два поля – имя и возраст человека. Сопоставление выше позволяет извлечь имя второго по списку человека.

Еще одна любопытная функциональная возможность – это объединение сопоставлений в цепочки. Прежде чем посмотреть на пример, немного углубимся в понятие выражения сопоставления.

В общем виде выражение сопоставления выглядит следующим образом:

```
pattern = expression
```

Как вы уже поняли из примеров, справа может находиться абсолютно любое выражение:

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

Проанализируем, что именно происходит за кулисами.

1. Выполняется выражение справа.
2. Результирующее значение сопоставляется с образцом, стоящим слева.
3. К переменной справа привязывается результирующее значение.
4. Результат выражения сопоставления (результат терма справа) выводится на экран.

Добавим в продолжение этого списка то, что выражения сопоставления можно объединять в цепочки:

```
iex(3)> a = (b = 1 + 3)
4
```

Теперь разложим по действиям этот простой пример.

1. Выполняется выражение `1 + 3`.
2. Результат (4) сопоставляется с образцом `b`.
3. Результат вложенного сопоставления (4) сопоставляется с образцом `a`.

Следовательно, обе переменные (`a` и `b`) имеют значение 4.

Скобки в записи сложных сопоставлений не обязательны, и многие разработчики опускают их:

```
iex(4)> a = b = 1 + 3
4
```

Как видите, результат получился такой же, так как оператор = является правоассоциативным.

Теперь разберем более полезный пример. Вспомним функцию `:calendar.local_time/0`:

```
iex(5)> :calendar.local_time()
{{2018, 11, 11}, {21, 28, 41}}
```

Допустим, необходимо получить результат функции (типа datetime), а также значение текущего часа. Вот как можно сделать это в одном сложном сопоставлении:

```
iex(6)> date_time = {_, {hour, _, _}} = :calendar.local_time()
```

Даже если поменять порядок сопоставляемых элементов (при условии выполнения выражения в одно и то же время), результат останется тем же:

```
iex(7)> {_, {hour, _, _}} = date_time = :calendar.local_time()
```

В любом случае, задача будет решена:

```
iex(8)> date_time
{{2018, 11, 11}, {21, 32, 34}}

iex(9)> hour
21
```

Это возможно благодаря тому, что результатом сопоставления всегда является результат терма справа. При удачном сопоставлении вы сможете запросто извлечь любые необходимые вам части этого терма.

3.1.9. Обобщенное поведение

Теперь вам должна быть понятна основная механика сопоставления с образцом. Попробуем обобщить все то, что было рассмотрено на многочисленных примерах.

Выражение сопоставления состоит из двух частей – *образца* (с левой стороны) и *терма* (с правой стороны). В таком выражении терм сопоставляется с образцом.

Если сопоставление прошло успешно, ко всем переменным образца привязываются соответствующие значения терма, а результатом всего выражения является сам терм. Если сопоставление неудачно, возникает ошибка.

Соответственно, в выражении сопоставления выполняются два основных действия:

- проверка соответствия образца и терма, возникновение ошибки при неудаче;
- привязка значений терма к переменным образца.

Выражения с оператором сопоставления – это всего лишь один из примеров использования сопоставления с образцом. На самом деле оно может присутствовать в различного рода выражениях и особенно эффективно используется в функциях.

3.2. Сопоставление с образцом в функциях

Сопоставление с образцом можно использовать при задании аргументов функции. Давайте вспомним, как выглядит простейшее объявление функции:

```
def my_fun(arg1, arg2) do
...
end
```

Параметры `arg1` и `arg2` – это образцы, с которыми можно выполнять сопоставление по обычным правилам.

Рассмотрим это на конкретном примере. Как уже говорилось во второй главе, кортежи часто используются для объединения связанных между собой полей. Например, при подсчете геометрических величин прямоугольник можно представить в виде кортежа `{a, b}`, содержащего длины его сторон в качестве элементов. В листинге 3.1 показано объявление функции, которая вычисляет площадь прямоугольника.

Листинг 3.1 ❖ Сопоставление с аргументами функции (rect.ex)

```
defmodule Rectangle do
  def area({a, b}) do    <-- Сопоставление с аргументом
    a * b
  end
end
```

Обратите внимание на то, как производится сопоставление с аргументом. Функция `Rectangle.area/1` ожидает получить на вход кортеж с двумя элементами

в качестве аргумента, затем она привязывает соответствующие элементы кортежа к переменным и возвращает результат.

Давайте проверим, сработает ли это в интерактивной оболочке. Запустите iex и подгрузите модуль, выполнив следующую команду:

```
$ iex rect.ex
```

Теперь вызовите функцию:


```
iex(1)> Rectangle.area({2, 3})
6
```

Разберемся, что только что произошло. Когда вы вызываете функцию, введенные вами аргументы сопоставляются с образцами, указанными в определении функции. Функция получает на вход кортеж с двумя элементами и привязывает их значения к переменным a и b.

В случае с функциями термом является аргумент, переданный функции во время вызова, а образцом – параметр, обозначенный в ее определении, то есть кортеж {a, b} для данного примера.

Само собой, если передать функции что-либо, отличное от кортежа с двумя элементами, возникнет ошибка:


```
iex(2)> Rectangle.area(2)
** (FunctionClauseError) no function clause matching in Rectangle.area/1
    iex:2: Rectangle.area(2)
```

Сопоставление с аргументами функции – очень эффективный прием. Именно он лежит в основе одной из самых главных возможностей Elixir – функций с несколькими предложениями (multiclause functions).

3.2.1. Функции с несколькими предложениями

Elixir позволяет осуществлять перегрузку функций при помощи указания нескольких предложений. *Предложение* – это определение функции, следующее после конструкции def. Если одна и та же функция с определенной арностью имеет несколько определений, то такая функция называется *функцией с несколькими предложениями*.

Рассмотрим пример. Скажем, необходимо разработать модуль Geometry для вычисления величин различных геометрических фигур. Представим их в виде кортежей, первый элемент которых будет содержать название фигуры:

```
rectangle = {:rectangle, 4, 5}
square = {:square, 5}
circle = {:circle, 4}
```

Имея представления форм в таком виде, создадим функцию для подсчета площади фигуры.

Листинг 3.2 ❖ Функции с несколькими предложениями (geometry.ex)

```
defmodule Geometry do
  def area({:rectangle, a, b}) do ◄——— Первое предложение функции area/1
    a * b
  end
```

```
  def area({:square, a}) do          <-- Второе предложение функции area/1
    a * a
  end

  def area({:circle, r}) do          <-- Третье предложение функции area/1
    r * r * 3.14
  end
end
```

Как видите, одна и та же функция имеет три предложения. От передаваемого аргумента будет зависеть, какое из предложений будет вызвано. Посмотрим, как это работает в оболочке:

```
iex(1)> Geometry.area({:rectangle, 4, 5})
20

iex(2)> Geometry.area({:square, 5})
25

iex(3)> Geometry.area({:circle, 4})
50.24
```

Когда вы вызываете функцию, среда выполнения проходит каждое из ее предложений в том порядке, в котором они указаны в исходном коде, а затем пытается сопоставить их с переданными в нее аргументами. Выполняется первое предложение, соответствующее всем аргументам.

Если ни одно предложение не соответствует образцу, возникает ошибка:

```
iex(4)> Geometry.area({:triangle, 1, 2, 3})
** (FunctionClauseError) no function clause matching in Geometry.area/1
    geometry.ex:2: Geometry.area({:triangle, 1, 2, 3})
```

Важно понимать, что с точки зрения вызывающей стороны функция с несколькими предложениями является обычной. Нельзя обратиться к определенному предложению, работать можно только со всей функцией целиком.

Это распространяется не только на вызовы функций. Ранее во второй главе приводился пример создания значения функции с помощью оператора захвата `&`:

```
&Module.fun/arity
```

Если таким же образом обратиться к функции `Geometry.area/1`, то захват будет распространяться на все ее предложения:

```
iex(4)> fun = &Geometry.area/1     <-- Захват всей функции

iex(5)> fun.({:circle, 4})
50.24

iex(6)> fun.({:square, 5})
25
```

Это доказывает, что функция воспринимается как единое целое, даже если она содержит несколько предложений.

Иногда вместо вывода ошибки бывает необходимо получить ошибочный терм. В таком случае можно указать *стандартное предложение*, которое всегда будет давать удачный результат при сопоставлении. Сделаем это для функции `area`. В сле-

дующем листинге функция area содержит еще одно предложение, срабатывающее при неверно введенных данных.

Листинг 3.3 ❖ Функция с несколькими предложениями (geometry_invalid_input.ex)

```
defmodule Geometry do
  def area({:rectangle, a, b}) do
    a * b
  end

  def area({:square, a}) do
    a * a
  end

  def area({:circle, r}) do
    r * r * 3.14
  end

  def area(unknown) do
    {:error, {:unknown_shape, unknown}}
  end
 end
```

Дополнительное предложение, срабатывающее при неверно введенных данных

Если ни одно из первых трех предложений не соответствует образцу, вызывается последнее, так как в нем образец с переменными всегда будет соответствовать указанному терму. В данном примере в случае ошибки возвращаться будет кортеж с двумя элементами {:error, reason}.

Попробуем вызвать функцию в оболочке:

```
iex(1)> Geometry.area({:square, 5})
25

iex(2)> Geometry.area({:triangle, 1, 2, 3})
{:error, {:unknown_shape, {:triangle, 1, 2, 3}}}
```

> **ПРИМЕЧАНИЕ** Чтобы все сработало как нужно, не забудьте поместить предложения в определенном порядке. Среда выполнения всегда проходит предложения в том порядке, в котором они заданы в исходном коде. Если предложение area(unknown) будет стоять первым, вы всегда будете получать ошибку.

Обратите внимание, что предложение area(unknown) справедливо только для функции area/1. Если передать в функцию больше одного аргумента, то это предложение не будет вызвано. Как вы уже знаете, функции отличаются по имени и арности. Так как функции с одним и тем же именем, но разной арностью – на самом деле две разные функции, не существует такого способа задать предложение area, чтобы оно выполнялось независимо от количества переданных аргументов.

И последнее. Всегда помещайте предложения одной и той же функции друг за другом, вместо того чтобы разбрасывать их по разным частям модуля. Если следы функции с несколькими предложениями проскальзывают на протяжении всего файла, то проанализировать поведение этой функции будет крайне затруднительно. Компилятор тоже не будет от этого в восторге и выведет предупреждение.

3.2.2. Ограничители

Предположим, вам нужно написать функцию, принимающую число и возвращающую атом :negative, :zero или :positive в зависимости от значения числа. К сожалению, обычное сопоставление с образцом здесь бессильно. Однако Elixir сможет решить эту задачу при помощи ограничителей.

Ограничители – это своеобразное расширение возможностей стандартного механизма сопоставления с образцом. Они позволяют задавать дополнительные условия удачного сопоставления.

Чтобы указать ограничитель, добавьте предложение when после списка аргументов. Разберем это на примере. Код ниже определяет, какое число было передано – положительное, отрицательное или ноль.

Листинг 3.4 ❖ Использование ограничителей (test_num.ex)

```
defmodule TestNum do
  def test(x) when x < 0 do
    :negative
  end

  def test(0), do: :zero

  def test(x) when x > 0 do
    :positive
  end
end
```

Ограничитель – это логическое выражение, накладывающее на предложение определенные условия. Первое предложение будет вызвано, только если передать функции отрицательное число, а последнее – если передать положительное, как показано ниже:

```
iex(1)> TestNum.test(-1)
:negative

iex(2)> TestNum.test(0)
:zero

iex(3)> TestNum.test(1)
:positive
```

Интересно, что если вызвать эту функцию с аргументом, не являющимся числом, можно получить довольно странный результат:

```
iex(4)> TestNum.test(:not_a_number)
:positive
```

В чем тут дело? Объяснить это можно тем, что термы в Elixir можно сравнивать посредством операторов < и >, даже если их типы не одинаковы. Поэтому результат зависит от следующего порядка типов:

```
number < atom < reference < fun < port < pid <
  tuple < map < list < bitstring (binary)
```

Число всегда меньше любого другого типа, именно поэтому функция TestNum.test/1 возвращает :positive, если передать в нее значение любого другого типа.

Чтобы этого избежать, добавьте в ограничитель проверку типа аргумента, как показано ниже.

Листинг 3.5 ❖ Использование ограничителей (test_num2.ex)

```
defmodule TestNum do
  def test(x) when is_number(x) and x < 0 do
    :negative
  end

  def test(0), do: :zero

  def test(x) when is_number(x) and x > 0 do
    :positive
  end
end
```


В данном примере для проверки того, является ли переданный аргумент числом, используется функция `Kernel.is_number/1`. Она выводит ошибку, если передать аргумент другого типа:

```
iex(1)> TestNum.test(-1)
:negative

iex(2)> TestNum.test(:not_a_number)
** (FunctionClauseError) no function clause matching in TestNum.test/1
```

Набор операторов и функций, которые могут присутствовать в ограничителях, строго определен. Функции, созданные вами, и большинство существующих функций не сработают. Вот некоторые операторы и функции, допустимые к использованию в ограничителях:

- операторы сравнения (`==`, `!=`, `===`, `!==`, `>`, `<`, `<=`, `>=`);
- логические операторы (`and`, `or`) и операторы отрицания (`not`, `!`);
- арифметические операторы (`+`, `-`, `*`, `/`);
- функции проверки типов из модуля `Kernel` (`is_number/1`, `is_atom/1` и т. д.).

Полный список операторов и функций ищите по адресу: **https://hexdocs.pm/elixir/guards.html**.

В каких-то случаях используемая в ограничителе функция может вызывать ошибку. Например, функция `length/1` работает только со списками. Предположим, вы использовали эту функцию для вычисления наименьшего элемента непустого списка:

```
defmodule ListHelper do
  def smallest(list) when length(list) > 0 do
    Enum.min(list)
  end

  def smallest(_), do: {:error, :invalid_argument}
end
```

Можно подумать, что вызов `ListHelper.smallest/1` с аргументом, отличным по типу от списка, вызовет ошибку, но этого не произойдет. Если ошибка сгенерируется внутри ограничителя, то она внутри него и останется, а выражение с ограничителем возвратит `false`. Такое предложение не будет соответствовать образцу.

Если вызвать функцию `ListHelper.smallest(123)` в предыдущем примере, она возвратит `{:error, :invalid_argument}`. Это доказывает, что ошибка обрабатывается внутри выражения с ограничителем.

3.2.3. Анонимные функции с несколькими предложениями

Анонимные функции (лямбда-функции) также могут состоять из нескольких предложений. Для начала вспомним простейший способ объявления и использования таких функций:

```
iex(1)> double = fn x -> x*2 end
```
Определение лямбда-функции

```
iex(2)> double.(3)
```
Вызов лямбда-функции

```
6
```

В общем виде синтаксис анонимной функции с несколькими предложениями такой:

```
fn
  pattern_1, pattern_2 ->
    ...

  pattern_3, pattern_4 ->
    ...

  ...
end
```

Выполняется, если сопоставление с образцом pattern_1 удачно

Выполняется, если сопоставление с образцом pattern_2 удачно

Разберем это на примере, переписав функцию `test/1`, проверяющую, какое число было передано – положительное, отрицательное или ноль:

```
iex(3)> test_num =
          fn
            x when is_number(x) and x < 0 ->
              :negative

            0 -> :zero

            x when is_number(x) and x > 0 ->
              :positive
          end
```

Обратите внимание, что предложение лямбда-функции заканчивается там, где начинается новое предложение (в виде `pattern →`), или там, где ключевым словом `end` заканчивается определение функции.

Поскольку все предложения анонимной функции перечисляются в одном выражении `fn`, то принято опускать скобки в каждом предложении. Напротив, каждое предложение именованной функции указывается в отдельном выражении `def` (или `defp`), поэтому в них скобки все же рекомендуется ставить.

Проверим, как работает определенная ранее функция:

```
iex(4)> test_num.(-1)
:negative

iex(5)> test_num.(0)
:zero
```

```
iex(6)> test_num.(1)
:positive
```

Анонимные функции с несколькими предложениями находят свое применение при реализации функций высшего порядка. Вы увидите это сами чуть позже, а на этом мы закончим изучение основной теории по функциям с несколькими предложениями. Они играют важную роль в реализации условного ветвления во время исполнения.

3.3. Условные конструкции

Elixir предоставляет несколько базовых способов реализации условных переходов, включая конструкции `if` и `case`. Функции с несколькими предложениями тоже могут быть использованы для этих целей. В данном разделе будут рассмотрены все возможные в Elixir способы реализации ветвления.

3.3.1. Ветвление с помощью функций с несколькими предложениями

В предыдущих примерах вы уже научились использовать функции с несколькими предложениями для реализации условной логики:

```
defmodule TestNum do
  def test(x) when x < 0, do: :negative
  def test(0), do: :zero
  def test(x), do: :positive
end
```

Три предложения представляют собой три условных перехода. В таком типовом императивном языке программирования, как JavaScript, вы бы написали что-то подобное:

```
function test(x){
  if (x < 0) return "negative";
  if (x == 0) return "zero";
  return "positive";
}
```

Можно сказать, что обе версии одинаково читабельны. Однако функции с несколькими предложениями помогут извлечь все преимущества сопоставления с образцом. В примере ниже функция с несколькими предложениями используется для проверки списка на пустоту:

```
defmodule TestList do
  def empty?([]), do: true
  def empty?([_|_]), do: false
end
```

Первое предложение выполняется, если список пустой, а второе – когда список содержит значения и представлен структурой `[head | tail]`.

Используя сопоставление с образцом, вы можете реализовывать полиморфные функции, выполняющие различные действия в зависимости от типа передаваемых в них аргументов. В следующем примере приведена реализация функции, удваивающей переменную. Поведение функции при передаче в нее числа и бинарных данных (строки) различно:

```
iex(1)> defmodule Polymorphic do
          def double(x) when is_number(x), do: 2 * x
          def double(x) when is_binary(x), do: x <> x
        end

iex(2)> Polymorphic.double(3)
6

iex(3)> Polymorphic.double("Jar")
"JarJar"
```

Потенциал функций с несколькими предложениями становится заметен при использовании их в рекурсиях. Код становится декларативным и избавлен от лишних операторов `if` и `return`. Вот так выглядит рекурсивная реализация факториала на основе функции с несколькими предложениями:

```
iex(4)> defmodule Fact do
          def fact(0), do: 1
          def fact(n), do: n * fact(n - 1)
        end

iex(5)> Fact.fact(1)
1

iex(6)> Fact.fact(3)
6
```

Рекурсия на основе функций с несколькими предложениями часто выступает в качестве ключевого элемента в организации работы циклов. Это подробным образом будет рассмотрено в следующем разделе, а сейчас приведу простой пример функции, суммирующей все элементы списка:

```
iex(7)> defmodule ListHelper do
          def sum([]), do: 0
          def sum([head | tail]), do: head + sum(tail)
        end

iex(8)> ListHelper.sum([])
0

iex(9)> ListHelper.sum([1, 2, 3])
6
```

Суммирование реализовано с учетом рекурсивного определения списка. Сумма элементов пустого списка всегда равна 0, а непустого – сумме значений его головы и хвоста.

Возможности функций с несколькими предложениями не выходят за рамки возможностей стандартных конструкций ветвления. Однако их использование предполагает разбиение кода на несколько несложных функций и помещение условной логики вглубь последних из них. Лежащий в основе механизм сопостав-

ления с образцом дает возможность реализовать любые виды ветвления на базе значений или типов аргументов функции. Функции высшего порядка остаются сосредоточенными на основной логике программы, а код становится гораздо более наглядным.

Нельзя не отметить, что иногда код выглядит лучше с использованием ветвления в классическом, то есть императивном, виде. Рассмотрим еще несколько конструкций ветвления, возможных в Elixir.

3.3.2. Классические конструкции ветвления

Решения с функциями с несколькими предложениями не всегда уместны. Они предполагают создание отдельной функции и передачу ей необходимых аргументов. В некоторых случаях проще использовать в функции классические условные конструкции, и для этого в Elixir существуют макросы `if`, `unless`, `cond` и `case`. Грубо говоря, они работают так же, как и соответствующие конструкции в других языках, но все же кое-чем отличаются. Рассмотрим каждый макрос подробнее.

If u unless

Синтаксис макроса `if` должен быть вам знаком:

```
if condition do
  ...
else
  ...
end
```

Если условие истинно, то выполняется главная ветвь, если же условие возвращает `false` или `nil`, выполняется блок `else`.

Можно записать все выражение в одну строку, чтобы оно выглядело примерно как выражение с конструкцией def:

```
if condition, do: something, else: another_thing
```

Любое выражение в Elixir возвращает какое-либо значение. Выражение с `if` возвращает результат выполненного блока (то есть результат последнего выражения этого блока). Если условие ложно, а блок `else` отсутствует, то результатом всего выражения будет атом `nil`:

```
iex(1)> if 5 > 3, do: :one
:one

iex(2)> if 5 < 3, do: :one
nil

iex(3)> if 5 < 3, do: :one, else: :two
:two
```

Рассмотрим более конкретный пример. Реализуем функцию `max`, возвращающую больший элемент из двух (согласно семантике оператора `>`):

```
def max(a, b) do
  if a >= b, do: a, else: b
end
```

Также доступен и макрос `unless`, являющийся эквивалентом конструкции `if (not …)`:

```
def max(a, b) do
  unless a >= b, do: b, else: a
end
```

Cond

Макрос `cond` выступает в качестве эквивалента шаблона `if-else-if`. Он принимает некоторое число условий и выполняет блок первого истинного из них:

```
cond do
  expression_1 ->
    ...

  expression_2 ->
    ...
 ...
end
```

Результатом `cond` является результат выполненного блока. Если все условия ложны, возникает ошибка.

Функция `max/2` при использовании конструкции `cond` будет выглядеть следующим образом:

```
def max(a, b) do
  cond do
    a >= b -> a
    true -> b      <-- Эквивалент стандартного условия
  end
end
```

Этот пример должен быть вам предельно понятен, за исключением странной части `true → b`. `true` здесь означает, что это условие всегда будет истинным, и этот блок выполняется, только если ни одно из стоящих выше условий конструкции `cond` не было удовлетворено.

Case

Синтаксис конструкции `case` имеет следующий вид:

```
case expression do
  pattern_1 ->
    ...

  pattern_2 ->
    ...

  ...
end
```

Слово pattern здесь употреблено потому, что в этих строках происходит сопоставление с образцом. Сначала вычисляется указанное выражение, а затем его результат сопоставляется с последующими образцами. Выполняется тот блок, где сопоставление было удачным, а результат этого блока и будет результатом всего

выражения case. Если сопоставление не удалось ни с одним из образцов, выводится ошибка.

Функция max с конструкцией case в теле будет выглядеть следующим образом:

```
def max(a,b) do
  case a >= b do
    true -> a
    false -> b
  end
end
```

Конструкция case отлично подойдет в качестве альтернативы функции с несколькими предложениями. Эти два подхода совершенно равносильны, а синтаксис case можно спроецировать на синтаксис функции с несколькими предложениями:

```
defp fun(pattern_1), do: ...
defp fun(pattern_2), do: ...
...
```

Достаточно лишь прописать fun(expression) при вызове.

Чтобы стандартное условие всегда было истинным, можно использовать в его записи анонимную переменную:

```
case expression do
  pattern_1 -> ...
  pattern_2 -> ...
  ...
  _ -> ...    <-- Стандартное условие, которое всегда истинно
end
```

Как вы могли заметить, в Elixir существуют различные способы реализации условной логики. Функции с несколькими предложениями позволяют сделать это в более явном виде, но для этого понадобится определить отдельную функцию и передать в нее все необходимые аргументы. Классические конструкции типа if и case придают коду более императивный вид и чаще всего выглядят проще. Выбор подходящего решения зависит от конкретной задачи и предпочтений разработчика.

3.3.3. Специальная форма with

Рассмотрим последнюю конструкцию ветвления – *специальную форму* with, позволяющую объединить несколько выражений и возвратить ошибку при обнаружении первого ложного из них. Разберемся с этим на простом примере.

Представим, что необходимо обработать регистрационные данные, подтвержденные пользователем. В качестве исходных данных имеется словарь с ключами-строками ("login", "email" и "password"). Он выглядит следующим образом:

```
%{
  "login" => "alice",
  "email" => "some_email",
  "password" => "password",
```

```
  "other_field" => "some_value",
  "yet_another_field" => "...",
  ...
}
```

Наша задача – привести данный словарь к нормальному виду так, чтобы он содержал только поля логина, почты и пароля. Как правило, если набор полей четко определен и заранее известен, ключи можно представить в виде атомов. Поэтому для данного словаря можно возвратить следующую структуру:

```
%{login: "alice", email: "some_email", password: "password"}
```

Однако запрашиваемые поля не всегда могут присутствовать в исходном словаре, и в таком случае было бы неплохо возвратить ошибку. То есть нужно сделать так, чтобы функция возвращала либо стандартизированный словарь, либо ошибку, а в характерном для Elixir виде `{:ok, some_result}` или `{:error, error_reason}`. Для данного примера `some_result` – это стандартизированный словарь данных пользователя, а `error_reason` – текст с описанием ошибки.

Начнем с создания вспомогательных функций для извлечения значений каждого поля:

```
defp extract_login(%{"login" => login}), do: {:ok, login}
defp extract_login(_), do: {:error, "login missing"}

defp extract_email(%{"email" => email}), do: {:ok, email}
defp extract_email(_), do: {:error, "email missing"}

defp extract_password(%{"password" => password}), do: {:ok, password}
defp extract_password(_), do: {:error, "password missing"}
```

Наличие поля здесь проверяется путем сопоставления с образцом.

Теперь займемся функцией высшего порядка `extract_user/1`, объединяющей три предыдущие функции. Можно реализовать ее, например, с помощью конструкции `case`:

```
def extract_user(user) do
  case extract_login(user) do
    {:error, reason} -> {:error, reason}

    {:ok, login} ->
      case extract_email(user) do
        {:error, reason} -> {:error, reason}

        {:ok, email} ->
          case extract_password(user) do
            {:error, reason} -> {:error, reason}

            {:ok, password} ->
              %{login: login, email: email, password: password}
          end
      end
  end
end
```

Многовато кода, учитывая, что в нем всего три функции. Каждый запрос дает по два ответвления в зависимости от результата, и в итоге имеется три вложенных

друг в друга выражения case. На практике чаще всего подобных проверок гораздо больше, и при таком способе код достаточно сильно засоряется.

Именно для таких случаев существует специальная форма with. Она дает возможность использовать сопоставление с образцом для сцепления нескольких выражений, а также проверки соответствия результата указанному образцу и возвращает первый из неудовлетворительных результатов.

Простейшая вариация конструкции with выглядит следующим образом:

```
with pattern_1 <- expression_1,
     pattern_2 <- expression_2,
     ...
do
  ...
end
```

Сначала выполняется первое выражение, и его результат сопоставляется с соответствующим образцом. Если сопоставление удалось, выполняется следующее выражение. Если все выражения соответствуют образцам, то выполняется блок do, а результатом всей конструкции with является результат последнего выражения блока do.

Однако если какое-то выражение не соответствует образцу, следующие за ним выражения выполняться не будут, а конструкция with возвратит результат неудачного выражения.

Рассмотрим пример:

```
iex(1)> with {:ok, login} <- {:ok, "alice"},
             {:ok, email} <- {:ok, "some_email"} do
          %{login: login, email: email}
        end

%{email: "some_email", login: "alice"}
```

В нем для извлечения логина и пароля проводятся два сопоставления с образцом, после чего выполняется блок do. Результатом всего выражения with является результат последнего выражения блока do. На первый взгляд, то же самое можно записать в виде:

```
{:ok, login} = {:ok, "alice"}
{:ok, email} = {:ok, "email"}
%{login: login, email: email}
```

Преимуществом реализации этого функционала с with является возможность возвратить результат терма, который не удалось сопоставить с образцом:

```
iex(2)> with {:ok, login} <- {:error, "login missing"},
             {:ok, email} <- {:ok, "email"} do
          %{login: login, email: email}
        end
{:error, "login missing"}
```

Это ровно то, что было нужно. Теперь перепишите функцию высшего уровня extract_user с использованием with, как показано в листинге 3.6.

Листинг 3.6 ❖ Извлечение данных о пользователе с помощью with (user_extraction.ex)

```
def extract_user(user) do
  with {:ok, login} <- extract_login(user),
       {:ok, email} <- extract_email(user),
       {:ok, password} <- extract_password(user) do
    {:ok, %{login: login, email: email, password: password}}
  end
end
```

Как видите, этот вариант намного короче и понятнее. Вы извлекаете необходимый набор данных и двигаетесь дальше только в случае успеха. Вы получаете на выходе приведенную в нормальный вид структуру или первую же возникшую ошибку. Полная реализация находится в файле user_extraction.ex. Попробуйте сами:

```
$ iex user_extraction.ex

iex(1)> UserExtraction.extract_user(%{})
{:error, "login missing"}

iex(2)> UserExtraction.extract_user(%{"login" => "some_login"})
{:error, "email missing"}

iex(3)> UserExtraction.extract_user(%{"login" => "some_login",
                                      "email" => "some_email"})
{:error, "password missing"}

iex(4)> UserExtraction.extract_user(%{"login" => "some_login",
                                      "email" => "some_email",
                                      "password" => "some_password"})
{:ok, %{email: "some_email", login: "some_login",
  password: "some_password"}}
```

Специальная форма with имеет еще несколько интересных возможностей, которые не будут рассмотрены в книге. Вы можете почитать о них здесь: **https://hexdocs.pm/elixir/Kernel.SpecialForms.html#with/1**.

На этом закончим с конструкциями ветвления. Самое время научиться реализовывать циклы и итерации на Elixir.

3.4. Циклы и итерации

По сравнению с циклами в других известных языках программирования, циклы в Elixir работают совершенно иначе. Конструкции вроде while и do…while не предоставляются. Как бы то ни было, практически в любой программе существует необходимость реализации своего рода динамических циклов. Так как же тогда сделать это на Elixir?

Основной способ организации циклов в Elixir – это рекурсия, и далее вы узнаете, как ей пользоваться.

> **ПРИМЕЧАНИЕ** Несмотря на то что рекурсия лежит в основе любых циклических конструкций, в промышленном коде на Elixir она встречается относительно редко. Все потому, что в нем присутствует большое количество абстракций высшего порядка,

делающих рекурсию неочевидной. Далее по ходу книги вы изучите многие из них, однако крайне важно понимать, как устроен механизм рекурсии в Elixir, поскольку он является ключевым элементом практически всех сложных программ.

ПРИМЕЧАНИЕ Большинство примеров данного раздела посвящено простым задачам (вроде вычисления суммы всех элементов списка), то есть задачам, решение которых Elixir может красиво и эффективно упаковать в одну строку. Такие несложные примеры помогут вам лучше понять различные аспекты вычислений на основе рекурсии.

3.4.1. Итерация на основе рекурсии

Пусть необходимо реализовать функцию, выводящую первые *n* натуральных чисел (целых положительных чисел). Так как циклов в Elixir нет, попробуем сделать это через рекурсию. В листинге ниже показано стандартное решение этой задачи.

Листинг 3.7 ❖ Вывод первых *n* натуральных чисел (natural_nums.ex)

```
defmodule NaturalNums do
  def print(1), do: IO.puts(1)
  def print(n) do
    print(n - 1)
    IO.puts(n)
  end
end
```

В основе этого кода лежит рекурсия, сопоставление с образцом и функции с несколькими предложениями. Он выглядит довольно декларативно: если n = 1, то число выводится на экран, а в обратном случае выводятся первые n – 1 чисел и затем n-е число.

Выполнив этот код в оболочке, можно убедиться, что он работает верно:

```
iex(1)> NaturalNums.print(3)
1
2
3
```

Вы не могли не заметить, что функция сработает некорректно, если передать в нее отрицательное или вещественное число. Чтобы это исправить, нужно добавить дополнительные ограничители, что и предлагаю вам сделать в качестве упражнения.

Код из листинга 3.7 иллюстрирует стандартный способ организации условного цикла. Объявляется функция с несколькими предложениями, в первых предложениях которой прописывается выход из рекурсии. Далее следуют основные предложения, частично реализующие вывод результата и рекурсивный вызов функции.

Теперь попробуем вычислить что-то с помощью цикла и возвратить результат. Этот пример вы уже изучили в разделе конструкций ветвления, но обратимся к нему еще раз. В листинге ниже приведена реализация функции, суммирующей все элементы списка.

Листинг 3.8 ❖ Вычисление суммы всех элементов списка (sum_list.ex)

```
defmodule ListHelper do
  def sum([]), do: 0
```

```
  def sum([head | tail]) do
    head + sum(tail)
  end
end
```

Этот код говорит сам за себя:

- сумма всех элементов пустого списка равна 0;
- сумма всех элементов непустого списка равна сумме сумм головы и хвоста списка.

Посмотрим на это в действии:

```
iex(1)> ListHelper.sum([1, 2, 3])
6

iex(2)> ListHelper.sum([])
0
```

На примере других языков вам должно быть известно, что вызов функции помещает ее параметры в стек, а значит, потребляет некоторое количество памяти. Слишком глубокая рекурсия может привести к переполнению стека и аварийному завершению всей программы. В Elixir эта ситуация разрешается благодаря оптимизации хвостовой рекурсии.

3.4.2. Хвостовые вызовы функций

Если последним действием функции является вызов другой функции (или самой себя), это называется *хвостовым вызовом*:

```
def original_fun(...) do
  ...
  another_fun(...)    ←—— Хвостовой вызов
end
```

Elixir (а если быть точнее, то Erlang) воспринимает хвостовые вызовы как некий способ выполнения *оптимизации хвостовой рекурсии*. Во время вызова функции вместо стандартной операции помещения ее параметров в стек происходит что-то вроде безусловного перехода. Перед хвостовым вызовом функции дополнительное пространство в стеке не выделяется, а значит, и не расходуется лишняя память.

Как это возможно? В предыдущем примере последним действием функции `original_fun` является вызов функции `another_fun`. Конечный результат функции `original_fun` – это результат `another_fun`. Именно поэтому компилятор может безопасно перейти к началу `another_fun`, не задействовав при этом дополнительную память. Когда `another_fun` завершает свои действия, он возвращается к тому месту, где вызывается функция `original_fun`.

Хвостовые вызовы находят свое применение в рекурсивных функциях. Функция с хвостовой рекурсией – это функция, последней операцией вызывающая саму себя и работающая практически бесконечно без потребления дополнительной памяти.

Вот так выглядит на Elixir функция – эквивалент бесконечного цикла:

```
def loop_forever(...) do
  ...
```

```
  loop_forever(...)
end
```

Так как хвостовая рекурсия не потребляет дополнительную память, это решение идеально подходит для реализации достаточно длинных итераций. Тем не менее есть у него один недостаток – по сравнению с классической рекурсией оно выглядит менее декларативно.

В следующем листинге показана реализация функции `ListHelper.sum/1` с хвостовой рекурсией.

Листинг 3.9 ❖ Вычисление суммы первых n натуральных чисел с помощью хвостовой рекурсии (sum_list_tc.ex)

```
defmodule ListHelper do
  def sum(list) do
    do_sum(0, list)
  end

  defp do_sum(current_sum, []) do
    current_sum
  end

  defp do_sum(current_sum, [head | tail]) do
    new_sum = head + current_sum
    do_sum(new_sum, tail)
  end
end
```

Первое, что стоит отметить, – это то, что функций здесь две. Экспортированная функция `sum/1` вызывается клиентами модуля и на первый взгляд работает точно так же, как и раньше.

В приватной функции `do_sum/2` реализуется хвостовая рекурсия. Это функция с двумя предложениями, каждое из которых мы сейчас рассмотрим подробно. Второе представляет наибольший интерес, поэтому начнем с него:

```
defp do_sum(current_sum, [head | tail]) do
  new_sum = head + current_sum   <-- Подсчет нового значения суммы
  do_sum(new_sum, tail)          <-- Хвостовой вызов
end
```

В это предложение передаются два аргумента – список с элементами и сумма, подсчитанная ранее (`current_sum`). Затем вычисляется новое значение суммы и функция вызывает саму себя, передавая в вызов оставшуюся часть списка и значение новой суммы. Поскольку этот вызов происходит последним, функция считается функцией с хвостовым вызовом, и дополнительная память не расходуется.

Переменная `new_sum` введена в примере для наглядности. Можно представить вычисления и во встроенном виде:

```
defp do_sum(current_sum, [head | tail]) do
  do_sum(head + current_sum, tail)
end
```

Эта функция тоже рекурсивная, так как последним действием вызывает саму себя.

Теперь осталось разобраться с тем, что делает первое предложение функции `do_sum/2`:

```
defp do_sum(current_sum, []) do
  current_sum
end
```

Это предложение останавливает рекурсию. В нем осуществляется сопоставление с пустым списком, это и есть последний шаг итерации. Как только до него доходит дело, суммировать становится больше нечего, и возвращается полученный результат.

Наконец, функция `sum/1`:

```
def sum(list) do
  do_sum(0, list)
end
```

Она используется клиентами модуля и отвечает за инициализацию значения параметра `current_sum`, рекурсивно передаваемого в функцию `do_sum`.

Хвостовую рекурсию можно рассматривать в качестве аналога классического цикла в императивных языках. Параметр `current_sum` – обычный *аккумулятор*, то есть значение, к которому пошагово добавляется результат каждой итерации. Функция `do_sum/2` выполняет шаг итерации и передает аккумулятор следующему шагу. Как вы помните, в Elixir существует понятие иммутабельности, и поэтому для вычисления значения суммы на протяжении всего цикла необходимы такие действия. Первое предложение `do_sum/2` описывает конечную точку итерации и возвращает значение аккумулятора.

Переписанная с использованием хвостовой рекурсии версия функции для подсчета суммы элементов списка работает корректно, вы можете проверить это в оболочке:

```
iex(1)> ListHelper.sum([1, 2, 3])
6

iex(2)> ListHelper.sum([])
0
```

Как видите, с точки зрения вызывающей стороны функция работает точно так же. Изнутри же используется хвостовая рекурсия, что позволяет обрабатывать сколь угодно большие списки, не требуя дополнительного объема памяти.

Хвостовая или обычная?

Учитывая достоинства хвостовой рекурсии, вы могли подумать, что ее использование для создания циклов является предпочтительным. На самом деле все не так просто. Обычная рекурсия чаще всего выглядит более емко и элегантно и при определенных обстоятельствах положительно влияет на производительность. Работая с рекурсией, выбирайте то решение, которое, на ваш взгляд, подходит больше. Если необходимо реализовать бесконечный цикл, хвостовая рекурсия – единственное, что поможет это сделать. В любом другом случае делайте выбор в пользу наиболее емкого и производительного решения.

Как распознать хвостовой вызов

Хвостовые вызовы могут выглядеть по-разному. Предыдущий пример иллюстрирует самую очевидную его форму. Хвостовой вызов может присутствовать и в условном выражении:

```
def fun(...) do
  ...
  if something do
    ...
    another_fun(...)     ◄——— Хвостовой вызов
  end
end
```

Так как вызов `another_fun` является последней операцией функции, данный вызов – хвостовой. Это актуально и для выражений `unless`, `cond`, `case` и `with`.

А вот в следующем коде хвостового вызова нет:

```
def fun(...) do
  1 + another_fun(...)   ◄——— Этот вызов не хвостовой
end
```

Вызов `another_fun` – не последняя операция функции `fun`: чтобы получить результат внешней функции, необходимо увеличить результат функции `another_fun` на единицу.

Идеи для практики

Вся эта теория выглядит замысловатой, но все гораздо проще, чем кажется. Если вы перешли на Elixir с императивного языка, то вам понадобится какое-то время, чтобы приобрести навык рекурсивного мышления в комбинации с механизмом сопоставления с образцом. Если вы решите взять небольшой перерыв и потренироваться с рекурсией самостоятельно, то ниже вы найдете список функций, которые стоит попробовать реализовать:

- функция `list_len/1`, вычисляющая длину списка;
- функция `range/2`, принимающая два целых числа `from` и `to` и возвращающая список всех чисел, находящихся в заданном интервале;
- функция `positive/1`, принимающая список и возвращающая другой список, содержащий только положительные числа исходного списка.

Попробуйте реализовать эти функции сначала с обычной рекурсией, а затем перепишите их с использованием хвостовой рекурсии. При возникновении неразрешимых вопросов вы можете обратиться к файлам recursion_practice.ex и recursion_practice_tc.ex с готовыми решениями.

Рекурсия – стандартная техника организации циклов. Несмотря на то что без нее не обходится ни один цикл, нет необходимости так часто использовать ее в явном виде. Многие типовые задачи можно решить при помощи функций высшего порядка.

3.4.3. Функции высшего порядка

Функция высшего порядка – это функция, принимающая в качестве аргументов другие функции и/или возвращающая значение функции в качестве результата.

Во второй главе вам уже встречалась такая функция, когда вы использовали `Enum.each/2` для прохода списка и вывода всех его элементов. Вспомним, как это было:

```
iex(1)> Enum.each(
          [1, 2, 3],
          fn x -> IO.puts(x) end      ← Передача значения функции другой функции
        )
1
2
3
```

Функция `Enum.each/2` принимает перечисление (в данном случае список) и анонимную функцию. Она проходит через перечисление, вызывая лямбду для каждого его элемента. Так как `Enum.each/2` принимает другую функцию в качестве аргумента, она является функцией высшего порядка.

Функцию `Enum.each/2` можно использовать для перебора элементов перечисляемых структур, не прибегая к рекурсии. С точки зрения внутренней структуры `Enum.each/2` опирается на рекурсию, ведь без нее циклы в Elixir невозможны. Но сложность реализации рекурсии, повторяемость кода и премудрости хвостовой рекурсии скрыты от ваших глаз.

`Enum.each/2` – это лишь один пример итерации, выполненной с помощью функции высшего порядка. Стандартная библиотека Elixir предоставляет множество таких вспомогательных функций в модуле `Enum`. Модуль `Enum` вообще содержит большое количество полезных функций, для циклов и итераций это своеобразная палочка-выручалочка. Уделите немного времени на изучение документации модуля (**https://hexdocs.pm/elixir/Enum.html**), так как в книге будет рассмотрена только одна из наиболее часто используемых его функций.

Перечисления

Большинство функций модуля `Enum` работает с перечислениями. Об этом будет подробно говориться в четвертой главе. А пока достаточно понимать, что *перечисление* – это структура данных, реализующая определенный контракт, благодаря чему может быть использована функциями из модуля `Enum`.

Перечислениями являются, например, списки, диапазоны, словари и структура `MapSet`. Можно также превратить вашу собственную структуру данных в перечисление и тем самым извлечь максимум пользы из возможностей модуля `Enum`.

Вам часто понадобится проводить поэлементное преобразование элементов списка для получения нового списка. Функция `Enum.map/2` существует именно для этого. Она принимает перечисление и лямбда-функцию, преобразующую исходные элементы, в качестве аргументов. В следующем примере каждый элемент списка удваивается:

```
iex(1)> Enum.map(
          [1, 2, 3],
          fn x -> 2 * x end
        )
[2, 4, 6]
```

Как вы уже знаете, для того чтобы сделать определение анонимной функции более компактным, можно использовать оператор захвата &:

```
iex(2)> Enum.map(
          [1, 2, 3],
          &(2 * &1)
        )
```

Конструкция &(…) обозначает упрощенное определение анонимной функции, где &n используется в качестве заполнителя для n-го аргумента функции.

Enum.filter/2 – еще одна полезная функция, используемая для извлечения нескольких элементов списка, выбранных по определенному критерию. Код ниже возвращает все нечетные числа списка:

```
iex(3)> Enum.filter(
          [1, 2, 3],
          fn x -> rem(x, 2) == 1 end
        )
[1, 3]
```

Enum.filter/2 принимает в качестве аргументов перечисление и лямбду, а возвращает только те элементы списка, для которых лямбда возвращает true.

Для этой функции также можно использовать синтаксис с оператором захвата:

```
iex(3)> Enum.filter(
          [1, 2, 3],
          &(rem(&1, 2) == 1)
        )
[1, 3]
```

Поиграем с Enum еще немного. Вспомним пример из раздела 3.3.3, в котором использовалась конструкция with для проверки подтверждения логина, почты и пароля и возвращалась первая из возникших ошибок. Вооружившись новыми знаниями, перепишем пример так, чтобы он сразу же возвращал все отсутствующие поля.

Вы помните, что на входе имеется словарь, и необходимо получить ключи "login", "email", и "password" и записать их в словарь, где ключи представлены атомами. Если запрашиваемое поле отсутствует, должна возбуждаться ошибка. Ранее вы добились того, чтобы возвращалась ошибка при обнаружении первого из ненайденных полей, но гораздо эффективнее было бы сделать так, чтобы возвращался сразу список всех отсутствующих полей.

Функция Enum.filter/2 позволяет сделать это без лишних усилий. Алгоритм таков, что при переборе элементов списка запрашиваемых полей необходимо извлечь только те поля, которые отсутствуют в словаре. Проверить наличие ключа можно с помощью функции Map.has_key?/2. Готовое решение будет выглядеть следующим образом.

Листинг 3.10 ❖ Вывод всех отсутствующих полей (user_extraction_2.ex)

```
case Enum.filter(
       ["login", "email", "password"],
       &(not Map.has_key?(user, &1))
     ) do
```

Выбор запрашиваемых полей

Получение только отсутствующих полей

```
  [] ->    <------ Все поля присутствуют
    ...

  missing_fields ->    <------ Некоторые поля отсутствуют
    ...
end
```

Функция `Enum.filter/2` может возвратить один из двух возможных результатов. Если на выходе вы получили пустой список, то все необходимые данные присутствуют, и можно их извлечь. В противном случае отсутствует несколько полей, и необходимо вывести ошибку. Код для каждой условной ветви в целях лаконичности в данном примере опущен, но вы можете найти его целиком в файле user_extraction_2.ex.

Reduce

Пожалуй, наиболее универсальной функцией модуля `Enum` можно назвать `Enum.reduce/3`. Ее можно использовать для преобразования перечисления во что угодно. Если в вашем языке поддерживаются функции первого класса, то вам функция `reduce` может быть известна под названием *inject* или *fold*.

Лучше всего рассмотреть, как работает функция `reduce`, на примере. Воспользуемся ей для подсчета суммы всех элементов списка. Прежде чем реализовать это на Elixir, попробуем представить решение в императивном виде, например на JavaScript:

```
var sum = 0;        <------ Инициализация суммы
[1, 2, 3].forEach(function(element) {
  sum += element;   <------ Суммирование с результатом
})
```

Так выглядит стандартное императивное решение. Сначала инициализируется аккумулятор (переменная `sum`), а затем начинается цикл, каждый шаг которого прибавляет к переменной значение нового элемента списка. Когда цикл подходит к концу, переменная хранит конечное значение.

В функциональных языках программирования переменная-аккумулятор не может быть изменена, но результат можно вычислить пошагово с помощью функции `Enum.reduce/3`. Она имеет следующую запись:

```
Enum.reduce(
  enumerable,
  initial_acc,
  fn element, acc ->
    ...
  end
)
```

`Enum.reduce/3` принимает перечисление в качестве первого аргумента, а вторым аргументом является изначальное значение аккумулятора, которое будет вычисляться пошагово. Последний аргумент – это лямбда-функция, которая вызывается для каждого элемента. Она принимает элемент перечисления и текущее значение аккумулятора. Задача этой функции – вычислить и возвратить новое значение аккумулятора. По завершении итерации `Enum.reduce/3` возвращает конечное значение аккумулятора.

Попробуем вычислить сумму элементов списка с помощью `Enum.reduce/3`:

```
iex(4)> Enum.reduce(
          [1, 2, 3],
          0,  <-- Задание начального значения аккумулятора
          fn element, sum -> sum + element end  <-- Пошаговое обновление значения аккумулятора
        )
6
```

Вот и все! Я сам перешел на Elixir с императивного языка, и, чтобы лучше понимать, что происходит, я представляю лямбду как функцию, которая вызывается в каждом шаге итерации и добавляет к результату часть информации.

Как отмечалось ранее, многие операторы сами по себе являются функциями, и вы можете превратить оператор в лямбду, вызвав `&+/2`, `&*/2` и т. д. Не забывайте об этом при работе с функциями высшего порядка. К примеру, предыдущий код можно записать в более сжатом виде:

```
iex(5)> Enum.reduce([1,2,3], 0, &+/2)
6
```

Стоит заметить, что существует также функция `Enum.sum/1`, которая делает ровно то же самое. Смысл примера был в том, чтобы показать реализацию перебора элементов коллекции и пошаговое увеличение результата.

Поработаем еще немного с `reduce`. Предыдущий пример сработает, только если передать список, содержащий исключительно числа. Если тип элементов будет другим, то возникнет ошибка (т. к. оператор + определен только для чисел). Следующий пример будет работать со списком, содержащим элементы любого типа, а суммироваться будут только его элементы-числа:

```
iex(6)> Enum.reduce(
          [1, "not a number", 2, :x, 3],
          0,
          fn  <-- Лямбда с несколькими предложениями
            element, sum when is_number(element) ->  <-- Действия с элементами-числами
              sum + element

            _, sum -> sum  <-- Действия с элементами других типов
          end
        )
```

В данном примере для достижения желаемого результата используется анонимная функция с несколькими предложениями. Если элемент является числом, его значение добавляется к уже имеющейся сумме. Если тип элемента другой, функция возвращает текущее значение суммы и передает его без изменений в следующий шаг итерации.

Лично я стараюсь избегать создания слишком объемных анонимных функций. Если в анонимной функции присутствует более или менее сложная логика, то есть вероятность, что код будет выглядеть лучше, если его вынести в отдельную функцию. Во фрагменте кода, представленном ниже, код анонимной функции перемещен в отдельную приватную функцию:

```
defmodule NumHelper do
  def sum_nums(enumerable) do
```

```
    Enum.reduce(enumerable, 0, &add_num/2)      ← Захват add_num/2 для лямбды
  end

  defp add_num(num, sum) when is_number(num), do: sum + num      ← Выполнение шагов итерации
  defp add_num(_, sum), do: sum
end
```

Такой подход не сильно отличается от предыдущего. Шаги итерации вынесены в отдельную приватную функцию `add_num/2`. Вызывая `Enum.reduce`, с помощью оператора захвата `&` вы передаете лямбду, которая делегирует этой функции.

Обратите внимание, что при захвате функции имя модуля не указывается: функция `add_num/2` находится в том же модуле, поэтому его префикс можно опустить. По факту `add_num/2` – приватная функция, и ее не получится захватить с префиксом модуля.

На этом обзор функций модуля `Enum` подходит к концу. Не забудьте уделить время не рассмотренным здесь функциям, они помогут упростить работу с циклами, итерациями и операциями с перечислениями.

3.4.4. Генераторы

Под этим таинственным названием скрывается еще одна конструкция, призванная помочь с организацией перебора и преобразования перечислений. В примере ниже для возведения элемента в квадрат используется *генератор*:

```
iex(1)> for x <- [1, 2, 3] do
          x*x
        end
```


Генератор запускает блок `do/end` для каждого элемента списка. В результате получается список, содержащий возвращенные блоком `do/end` значения. В таком простом виде `for` ничем не отличается от `Enum.map/2`.

Генераторы обладают и другими особенностями, благодаря которым решения с ними выглядят более элегантно, чем итерации с использованием функций `Enum`. Например, можно выполнить вложенные итерации по нескольким коллекциям. В следующем примере это делается для получения небольшой таблицы умножения:

```
iex(2)> for x <- [1, 2, 3], y <- [1, 2, 3], do: {x, y, x*y}
[
  {1, 1, 1}, {1, 2, 2}, {1, 3, 3},
  {2, 1, 2}, {2, 2, 4}, {2, 3, 6},
  {3, 1, 3}, {3, 2, 6}, {3, 3, 9}
]
```

В данном примере генератор выполняет вложенную итерацию, вызывая указанный выше блок для каждой комбинации исходных коллекций.

Как и функции модуля `Enum`, генераторы могут осуществлять перебор любой перечисляемой структуры данных. Например, составим таблицу умножения однозначных чисел с помощью диапазонов:

```
iex(3)> for x <- 1..9, y <- 1..9, do: {x, y, x*y}
```

Во всех рассмотренных примерах результатом генерации является список. Но генераторы могут возвращать любые данные, которые могут быть объединены в коллекцию. *Коллекция* – это абстрактный термин для обозначения характерного для функциональных языков типа данных, содержащего набор значений. Примерами коллекций могут послужить списки, словари, структуры MapSet и потоки файлов. Вы также можете создать свой собственный тип для коллекции (подробнее об этом в четвертой главе).

В общем случае генератор перебирает элементы перечисления, вызывая указанный блок для каждого значения и помещая результаты в структуру-коллекцию. Посмотрим, как это работает.

Код ниже возвращает словарь, содержащий таблицу умножения. Его ключами являются множители {x,y}, а значениями – их произведения:

```
iex(4)> multiplication_table =
          for x <- 1..9, y <- 1..9,
              into: %{} do      <-------- Определение коллекции
            {{x, y}, x*y}
          end

iex(5)> multiplication_table[{7, 6}]
42
```

Опция into определяет тип коллекции. В данном случае это пустой словарь %{}, в который будут записываться значения, возвращаемые блоком do. Обратите внимание, что возвращаемым результатом блока do является кортеж вида {множители, произведение}, потому что словарь «знает», как его обработать. Первый элемент будет ключом, а второй – соответствующим ему значением.

Еще одна примечательная особенность генераторов – это возможность определения фильтров, что позволяет опустить некоторые из передаваемых на вход элементов. В примере ниже генерируется несимметричная таблица умножения для чисел x и y, где x всегда меньше y:

```
iex(6)> multiplication_table =
          for x <- 1..9, y <- 1..9,
              x <= y,    <-------- Фильтр генератора
              into: %{} do
            {{x, y}, x*y}
          end

iex(7)> multiplication_table[{6, 7}]
42

iex(8)> multiplication_table[{7, 6}]
nil
```

Фильтр генератора применяется для каждого элемента перечисления до выполнения блока. Если фильтр возвращает true, то вызывается блок, и результат помещается в коллекцию, а в противном случае генератор переходит к следующему элементу.

Как видите, генераторы – очень занимательная вещь, позволяющая производить преобразования исходных перечисляемых структур. Да, функции модуля Enum, а именно Enum.reduce/3, тоже прекрасно с этим справляются, но нельзя не от-

метить, что с генераторами код выглядит изящнее. Это становится особенно заметным, когда необходимо вычислить декартово произведение нескольких перечислений, как в примере с таблицей умножения.

> **ПРИМЕЧАНИЕ** Генераторы работают и с бинарными данными, но синтаксис при этом немного отличается от рассмотренного выше. Ознакомиться с ним более подробно вы можете в документации генераторов по адресу: **https://hexdocs.pm/elixir/Kernel.SpecialForms.html#for/1**.

3.4.5. Потоки

Потоки – это особый вид перечислений, который может пригодиться для реализации составных операций над перечислениями с отложенным выполнением. Для наглядности разберем один из недостатков стандартных функций `Enum`.

Пусть имеется список сотрудников, и необходимо вывести каждый элемент с его порядковым номером по списку:

```
1. Alice
2. Bob
3. John
...
```

Задачу довольно легко решить, используя комбинацию нескольких функций `Enum`. Например, функция `Enum.with_index/1` принимает в качестве аргумента перечисление и возвращает список кортежей, первый элемент которых – это имя сотрудника, взятое из перечисления, а второй – индекс элемента перечисления (начиная с нуля):

```
iex(1)> employees = ["Alice", "Bob", "John"]
["Alice", "Bob", "John"]
iex(2)> Enum.with_index(employees)
[{"Alice", 0}, {"Bob", 1}, {"John", 2}]
```

Теперь можно передать результат `Enum.with_index/1` в `Enum.each/2`, чтобы получить желаемый результат:

```
iex(3)> employees                                  |>
        Enum.with_index                            |>
        Enum.each(
          fn {employee, index} ->
            IO.puts("#{index + 1}. #{employee}")
          end)
1. Alice
2. Bob
3. John
```

Здесь для объединения в цепочку нескольких вызовов функций используется оператор конвейера. Это позволяет отказаться от применения промежуточных переменных и делает код немного чище.

Оператор конвейера в оболочке

Как вы могли заметить, оператор конвейера помещен в конец строки. Это сделано потому, что в оболочке он должен стоять в той же строке, что и предыдущее выражение, иначе выражение будет сразу же выполнено оболочкой (как уже объяснялось во второй главе).

Оператор конвейера в конце строки сигнализирует компилятору, что выражение не может быть выполнено, пока не будет предоставлено больше входных данных. Однако в файлах с исходным кодом оператор |> следует помещать в начало следующей строки.

Что не так в этом коде? Очевидно, слишком много итераций. Функция `Enum.with_index/1` создает список кортежей, перебирая весь исходный список, а `Enum.each` выполняет еще одну итерацию уже по новому списку. Было бы гораздо эффективнее объединить обе операции в один шаг, с чем вам и помогут потоки.

Реализация потоков представлена в модуле `Stream` (**https://hexdocs.pm/elixir/Stream.html**), который на первый взгляд очень напоминает модуль `Enum` тем, что содержит функции `map`, `filter` и `take`. Эти функции принимают на вход любую перечисляемую структуру данных и возвращают поток – перечисление с особыми возможностями.

Поток – это перечисление с отложенным исполнением, то есть он возвращает результат только по запросу. Давайте посмотрим на это в действии.

В примере ниже каждый элемент списка удваивается с помощью потока:

```
iex(4)> stream = [1, 2, 3] |>
          Stream.map(fn x -> 2 * x end
#Stream<[enum: [1, 2, 3],
  funs: [#Function<44.45151713/1 in Stream.map/2>]]>
```

Создание потока

Результатом функции Stream.map/2 является поток

Поскольку поток выполняется по требованию, то итерация по исходному списку (`[1, 2, 3]`) и соответствующее преобразование (умножение на 2) еще не запущены. Вместо этого результатом является описывающая вычисление структура.

Чтобы запустить итерацию, необходимо передать поток в функцию `Enum` (each, map, filter и др.). Можно также использовать функцию `Enum.to_list/1`, преобразующую любое перечисление в список:

```
iex(5)> Enum.to_list(stream)
[2, 4, 6]
```

Итерация потока

`Enum.to_list/1` (и любая другая функция `Enum`) выполняется незамедлительно, она запускает итерацию по исходному списку и выводит результат. Таким образом, для работы `Enum.to_list/1` требуется, чтобы передаваемая в нее перечисляемая структура начала производить вычисления. Поэтому результат, возвращаемый потоком, создается в момент его передачи в функцию `Enum`.

Отложенные итерации по списку – не единственное действие, выполняющееся по запросу при использовании потоков. Значения в потоке вычисляются по одному и тогда, когда функция `Enum.to_list` запрашивает следующий элемент. Например, с помощью `Enum.take/2` можно вывести только один элемент потока:

```
iex(6)> Enum.take(stream, 1)
[2]
```

Так как `Enum.take/2` выполняет итерацию только по указанному числу элементов, переданный поток удвоил лишь один элемент списка, а остальные остались нетронутыми.

Возвращаясь к примеру с сотрудниками, теперь можно вывести список в один шаг при помощи потока. Это очень быстро, просто замените функцию `Enum.with_index/1` на ее эквивалент с отложенным выполнением `Stream.with_index/1`:

```
iex(7)> employees                                    |>
        Stream.with_index                            |>
            Enum.each(
              fn {employee, index} ->
                IO.puts("#{index + 1}. #{employee}")
              end)
1. Alice
2. Bob
3. John
```

Выполнение отложенного преобразования

Результат тот же, но итерация происходит только один раз. Это крайне удобно при выполнении нескольких преобразований одного и того же списка. Далее в примере вычисляется квадратный корень только для положительных элементов списка, и к результату в начало добавляется порядковый номер:

```
iex(1)> [9, -1, "foo", 25, 49]                                |>
        Stream.filter(&(is_number(&1) and &1 > 0))             |>
        Stream.map(&{&1, :math.sqrt(&1)})                      |>
        Stream.with_index                                      |>
        Enum.each(
            fn {{input, result}, index} ->
              IO.puts("#{index + 1}. sqrt(#{input}) = #{result}")
            end
          )
1. sqrt(9) = 3.0
2. sqrt(25) = 5.0
3. sqrt(49) = 7.0
```

Этот пример показывает, каким выразительным может быть решение при построении абстракций исключительно на основе функций. Для начала из исходного списка отбираются только положительные числа. Каждое из этих чисел преобразуется в кортеж `{input_number, square_root}`. Затем полученные кортежи нумеруются с помощью `Stream.with_index/1` и выводятся на экран.

После вызова `Enum.each` несколько преобразований выполняется за один шаг. Если бы везде использовались функции `Enum`, по каждому промежуточному списку необходимо было бы произвести несколько итераций, что непременно отразилось бы на производительности и повысило бы расход памяти.

Возможность отложенного выполнения потоков позволяет эффективно обрабатывать потенциально большие исходные перечисляемые структуры. Например, если нужно пропарсить файл построчно. При реализации этого через функции `Enum` придется считать весь файл в память и повторить одни и те же действия для каждой строки. В свою очередь, использование потоков позволяет считывать

и парсить каждую строку в один шаг. Следующая функция принимает имя файла в качестве аргумента и возвращает список всех строк файла длиной более 80 символов:

```
def large_lines!(path) do
  File.stream!(path)
  |> Stream.map(&String.replace(&1, "\n", ""))
  |> Enum.filter(&(String.length(&1) > 80))
end
```

В функцию `File.stream!/1` передается путь к файлу и возвращается поток с его строками. Поскольку результатом является поток, операции над файлом выполняются только по запросу. После вызова `File.stream!` файл остается непрочитанным. Затем опять же с отложенным выполнением из каждой строки удаляется символ переноса строки. И наконец, `Enum.filter/2` выбирает только длинные строки. Получается, что содержимое всего файла не помещается в память, а работа идет непосредственно с каждой строкой.

> **ПРИМЕЧАНИЕ** Компилятор Elixir не поддерживает перечисления с отложенным выполнением. Они имеют довольно сложную реализацию, хотя принцип работы потоков элементарен и основан на анонимных функциях. Чтобы реализовать отложенные вычисления в оболочке, необходимо возвратить лямбду, производящую эти вычисления. Таким образом, они будут выполняться не сразу, так как возвращается лишь описание вычислений, а не результат. В момент, когда вам понадобятся результаты вычислений, просто вызовите анонимную функцию.

Идеи для практики

Вам потребуется время, чтобы привыкнуть к такому стилю программирования. Приведенные в книге примеры – отличная практика, но все же вам следует попробовать написать пару итераций самим. Ниже вы увидите список упражнений, выполнив которые, вы сможете стать с Elixir еще ближе.

Реализуйте следующие функции, используя `large_lines!/1` в качестве модели:

- `lines_lengths!/1`, принимающую на вход путь к файлу и возвращающую список чисел, где каждое число – это длина соответствующей строки файла;
- `longest_line_length!/1`, возвращающую размер самой длинной строки;
- `longest_line!/1`, возвращающую содержимое самой длинной строки файла;
- `words_per_line!/1`, возвращающую список чисел, каждое из которых обозначает количество слов в строке (чтобы посчитать слова в строке, используйте функцию `length(String.split(line))`).

Готовые реализации вы сможете найти в файле enum_streams_practice.ex, но я настоятельно рекомендую вам поработать над решениями задач самостоятельно.

Выводы

- Сопоставление с образцом – это конструкция, проверяющая совпадение терма, стоящего справа, и образца, находящегося слева. В процессе сопоставления к переменным образца привязываются соответствующие значения терма, а если терм не соответствует образцу, возникает ошибка.

- Аргументы функций являются образцами. При вызове функции происходит сопоставление переданных в нее значений с указанными в ее определении образцами.
- Функции могут содержать несколько предложений, при этом выполняться будет первое соответствующее аргументам предложение.
- Для реализации условной логики можно использовать функции с несколькими предложениями и конструкции `if`, `unless`, `cond`, `case` и `with`.
- Рекурсия – основное средство построения циклов. Для реализации довольно продолжительных циклов используется хвостовая рекурсия.
- Функции высшего порядка значительно упрощают работу с циклами. Модуль `Enum` предоставляет множество функций для осуществления итераций, а модуль Stream поможет реализовать составные итерации с отложенным выполнением.
- Для преобразования, фильтрации, объединения различных перечислений, а также реализации итераций по ним можно использовать генераторы.

Глава 4

Абстракции данных

В главе рассматривается:
- создание абстракций с помощью модулей;
- работа с иерархическими данными;
- реализация полиморфизма через протоколы.

В данной главе речь пойдет о создании структур данных высокого уровня. Ни одна сложная система не обходится без классических примеров абстракций высокого уровня (`Money`, `Date`, `Employee`, `OrderItem` и т. д.), не поддерживаемых языком напрямую, а написанных на основе встроенных типов.

В Elixir подобные абстракции реализуются с помощью «чистых» модулей, не содержащих собственного состояния. В этой главе вы научитесь создавать свои собственные абстракции данных и работать с ними.

В объектно-ориентированных языках программирования основными элементами построения абстракции являются классы и объекты. Например, имеется класс `String`, реализующий различные операции со строками. Каждая строка считается экземпляром этого класса, а работать с ними можно посредством вызова методов, как показано ниже на примере Ruby:

```
"a string".upcase
```

В Elixir такой подход обычно не используется. Следуя принципам функционального программирования, Elixir старается отделить данные от кода. Вместо классов используются модули, то есть наборы функций. Вызов метода для объекта заменяется вызовом функций модуля в явном виде и передачей им исходных данных в качестве аргументов. Вот как выглядит та же операция со строкой в Elixir:

```
String.upcase("a string")
```

Еще одно существенное различие между Elixir и объектно-ориентированными языками – это иммутабельность данных. Чтобы изменить данные, придется вызвать функцию и привязать ее результат к переменной, при этом исходные данные останутся нетронутыми, как показано в примерах ниже:

```
iex(1)> list = []
[]

iex(2)> list = List.insert_at(list, -1, :a)
[:a]

iex(3)> list = List.insert_at(list, -1, :b)
```

```
[:a, :b]

iex(4)> list = List.insert_at(list, -1, :c)
[:a, :b, :c]
```


Здесь результат предыдущей операции сохраняется и передается следующей операции.

Оба примера с кодом на Elixir показывают, что модули используются в качестве абстракций поверх типов данных. При работе со строками вы обращаетесь к модулю `String`, а при работе со списками – к модулю `List`.

`String` и `List` – модули, представляющие определенные типы данных. Они реализованы на Elixir, а их функции работают с заранее определенным форматом исходных данных. Функции модуля `String` ожидают получить бинарную строку в качестве первого аргумента, а функции модуля `List` – список.

К тому же *функции-модификаторы* (те, что преобразуют данные) возвращают данные того же типа. Функция `String.upcase/1` возвращает бинарную строку, а функция `List.insert_at/3` – список.

Модуль также содержит *функции-запросы*, возвращающие какую-либо информацию по переданным данным (`String.length/1`, `List.first/1` и др.). Этим функциям в качестве первого аргумента необходимо передать экземпляр абстракции данных, но возвращают они данные другого типа.

Подведем итог и выделим основные правила абстрагирования данных в Elixir.

1. Модуль – единица абстракции данных.
2. Функции модуля обычно ожидают получить в качестве первого аргумента экземпляр абстракции данных.
3. Функции-модификаторы возвращают измененную абстракцию.
4. Функции-запросы возвращают данные другого типа.

Следуя этим правилам, вы без проблем сможете создать свою собственную абстракцию данных высокого уровня. В следующих разделах вы узнаете, как это сделать.

4.1. Создание абстракций с помощью модулей

Списки и строки – очевидно, простейшие типы данных, однако абстракции данных высокого уровня основаны на тех же принципах. Во второй главе вы уже могли наблюдать абстракцию данных высокого уровня. Например, модуль `MapSet` представляет собой реализацию набора значений. Он полностью написан на Elixir и служит отличным примером того, как должна выглядеть абстракция в Elixir.

Рассмотрим пример использования `MapSet`:

```
iex(1)> days =
          MapSet.new() |>          <-- Создание экземпляра абстракции
            MapSet.put(:monday) |>  |
            MapSet.put(:tuesday)    | Внесение изменений

iex(2)> MapSet.member?(days, :monday)     <-- Запрос к абстракции
true
```

Этот пример более или менее следует описанным выше принципам. Для упрощения кода используется оператор конвейера, объединяющий несколько дей-

ствий в одно. Это возможно благодаря тому, что все функции модуля MapSet принимают на вход набор значений. Такие функции можно спокойно объединять в цепочки при помощи оператора |>.

Обратите внимание на функцию new/0, создающую пустой экземпляр абстракции. В ней нет ничего интересного, и назвать ее можно как угодно. Ее основная задача – создать пустую структуру данных, с которой можно будет работать в дальнейшем.

Так как MapSet является абстракцией, клиентов этого модуля не интересует его внутреннее устройство или структура данных. Они просто вызывают его функции, получают определенный результат и передают его функциям того же модуля.

> **ПРИМЕЧАНИЕ** Можно предположить, что абстракции вроде MapSet не что иное, как определяемые пользователем типы данных. Несмотря на большое сходство, абстракции на основе модулей не являются такими же типами данных, как те, что были описаны во второй главе. Они строятся поверх стандартных типов данных. Например, экземпляр MapSet также является словарем, что можно легко проверить, вызвав is_map(MapSet.new()).

Взяв модуль MapSet в качестве шаблона, попробуем создать простую абстракцию данных.

4.1.1. Создание простой абстракции

В качестве примера в данном разделе создадим обычный список дел. В этой задаче нет ничего сверхъестественного, однако она достаточно сложная, для того чтобы всему научиться. При этом у вас будет возможность сконцентрироваться на особом методе ее решения, вместо того чтобы продумывать каждое свое действие.

Самая простая версия списка дел будет обладать следующими функциональными возможностями:

- создание новой абстракции данных;
- добавление новых записей;
- запрос к абстракции.

Вот так выглядит его желаемая реализация:

```
$ iex simple_todo.ex
iex(1)> todo_list =
          TodoList.new() |>
            TodoList.add_entry(~D[2018-12-19], "Dentist") |>
            TodoList.add_entry(~D[2018-12-20], "Shopping") |>
            TodoList.add_entry(~D[2018-12-19], "Movies")

iex(2)> TodoList.entries(todo_list, ~D[2018-12-19])
["Movies", "Dentist"]

iex(3)> TodoList.entries(todo_list, ~D[2018-12-18])
[]
```

Код говорит сам за себя. Функция TodoList.new/0 создает экземпляр структуры, затем в нее добавляются новые записи, после чего выполняются запросы. Выражение ~D[2018-12-19] создает дату (19 декабря 2018 г.) с помощью модуля Date (как уже объяснялось в разделе 2.4.11).

По ходу изучения книги вы будете постепенно расширять функционал и менять интерфейс данного примера. В самом конце вы получите полностью готовый к работе распределенный веб-сервер для управления большим количеством списков дел.

Начнем с создания простого интерфейса. Первым делом решим вопрос с внутренним представлением данных. Из предыдущего примера можно видеть, что в основном пользователю списка необходимо будет производить поиск записей по определенной дате. Похоже, для этих целей идеально подойдет словарь – даты будут ключами, а списки записей для этих дат – значениями. Таким образом, в реализации функции `new/0` не будет ничего сложного.

Листинг 4.1 ❖ Инициализация списка дел (simple_todo.ex)

```
defmodule TodoList do
  def new(), do: %{}
  ...
end
```

Теперь необходимо реализовать функцию `add_entry/3`. На вход ей передается список дел (то есть словарь), и она должна будет добавить запись в список под соответствующим ключом (датой). Необходимо также учесть случай, когда записей на ту или иную дату не существует. Выходит, все это можно сделать с помощью всего одного вызова функции `Map.update/4`.

Листинг 4.2 ❖ Добавление записи (simple_todo.ex)

```
defmodule TodoList do
  ...
  def add_entry(todo_list, date, title) do
    Map.update(
      todo_list,
      date,
      [title],    <-- Исходное значение
      fn titles -> [title | titles] end    <-- Анонимная функция обновления
    )
  end
  ...
end
```

Функции `Map.update/4` передается словарь, ключ, исходное значение и *анонимная функция обновления*. Если значения для данного ключа не существует, используется исходное значение, иначе вызывается корректирующая лямбда. Она принимает существующее значение и возвращает новое значение для данного ключа. В подобном случае новая запись помещается в начало списка, и, как вы помните из второй главы, это действие со списком является наиболее эффективным. Получается, порядок вставки записей игнорируется в целях выполнения операции вставки как можно быстрее, то есть последние добавленные записи будут находиться в самом начале списка.

И наконец, реализуем функцию `entries/2`, возвращающую все записи по заданной дате или пустой список, если таковых не найдено. Решение, приведенное в листинге 4.3, не должно вызывать затруднений.

Листинг 4.3 ❖ Запрос к списку дел (simple_todo.ex)

```
defmodule TodoList do
  ...
  def entries(todo_list, date) do
    Map.get(todo_list, date, [])
  end
end
```

Вызвав `Map.get/3`, вы получите значение из словаря `todo_list`, соответствующее заданной дате. Третий аргумент этой функции – значение, которое возвращается по умолчанию, если заданный ключ в словаре отсутствует.

4.1.2. Сложные абстракции

Никто не запрещает вам создавать абстракции на основе других абстракций. В примере с самой первой версией списка есть возможность вынести часть кода в отдельную абстракцию.

Обратите внимание на то, как выполняются действия со словарем, а именно на то, как возвращается несколько значений, хранящихся под одним и тем же ключом. Вынесем это в отдельную абстракцию и назовем ее `MultiDict`.

Листинг 4.4 ❖ Реализация абстракции MultiDict (todo-multi_dict.ex)

```
defmodule MultiDict do
  def new(), do: %{}

  def add(dict, key, value) do
    Map.update(dict, key, [value], &[value | &1])
  end

  def get(dict, key) do
    Map.get(dict, key, [])
  end
end
```

Эта реализация почти полностью скопирована с самой первой. Изменились только имена функций, и в определении анонимной функции обновления для более компактной записи появился оператор захвата (`&[value | &1]`).

С использованием этой абстракции модуль `TodoList` заметно упрощается.

Листинг 4.5 ❖ Реализация модуля TodoList с использованием MultiDict (todo_multi_dict.ex)

```
defmodule TodoList do
  def new(), do: MultiDict.new()

  def add_entry(todo_list, date, title) do
    MultiDict.add(todo_list, date, title)
  end

  def entries(todo_list, date) do
    MultiDict.get(todo_list, date)
  end
end
```

Вынесение части логики в отдельную абстракцию и создание еще одной абстракции поверх нее – классический пример разделения ответственности в коде. Абстракцию `MultiDict` при необходимости без проблем можно использовать в дальнейшей разработке. Кроме того, возможности модуля `TodoList` можно расширить, добавив в него дополнительные функции, не входящие в `MultiDict`. Например, функцию `due_today/2`, возвращающую все записи на сегодняшнюю дату.

С помощью этого небольшого преобразования я хотел показать, что организация кода в Elixir не сильно отличается от организации кода в объектно-ориентированных языках. Инструменты для создания абстракций (не хранящие состояний модули и чистые функции вместо классов и методов) другие, но основная идея одна и та же.

4.4.3. Структурирование данных с помощью словарей

`TodoList` теперь работает как настоящая абстракция. Вы можете добавлять записи в структуру и выводить все записи по заданной дате. Но его интерфейс оставляет желать лучшего: добавляя новую запись, необходимо каждое ее поле указывать в виде отдельного аргумента:

```
TodoList.add_entry(todo_list, ~D[2018-12-19], "Dentist")
```

А если добавить еще одно поле записи (например, время), то придется менять сигнатуру функции, что в результате нарушит работу всех клиентов модуля. Кроме того, необходимо будет внести изменения в код там, где используются эти данные.

Очевидно, чтобы решить данную проблему, нужно как-то объединить все поля записи в одной абстракции данных. Как упоминалось в разделе 2.4.6, в Elixir это обычно делается с помощью словаря с ключами-атомами, в роли которых будут выступать поля записи. В примере далее показано, как создавать и использовать экземпляр `entry`:

```
iex(1)> entry = %{date: ~D[2018-12-19], title: "Dentist"}

iex(2)> entry.date
~D[2018-12-19]

iex(3)> entry.title
"Dentist"
```

Переписать код и представить записи в виде словарей не составит труда. Все, что нужно сделать, – это изменить код функции `TodoList.add_entry` так, чтобы она принимала на вход два аргумента – экземпляр списка дел и словарь, описывающий запись. Обновленный код можно видеть в листинге 4.6.

Листинг 4.6 ❖ Представление записей в виде словарей (todo_entry_map.ex)

```
defmodule TodoList do
  ...  def add_entry(todo_list, entry) do
    MultiDict.add(todo_list, entry.date, entry)
  end
  ...
end
```

Запись в виде словаря, где поле с датой является ключом, добавляется в `MultiDict`. Проще простого!

Посмотрим, как это работает, на реальном примере. Теперь, чтобы добавить новую запись, необходимо передать в функцию словарь:

```
iex(1)> todo_list = TodoList.new() |>
          TodoList.add_entry(%{date: ~D[2018-12-19], title: "Dentist"})
```

Этот код выглядит более развернуто из-за указанных в нем имен полей. Но благодаря тому, что записи представлены в виде словарей, механизм извлечения данных работает иначе. Функция `TodoList.entries/2` теперь возвращает всю запись целиком, а не только ее заголовок:

```
iex(2)> TodoList.entries(todo_list, ~D[2018-12-19])
[%{date: ~D[2018-12-19], title: "Dentist"}]
```

Словарь – важная часть текущей реализации абстракции `TodoList`, и во время выполнения программы отличить словарь от экземпляра `TodoList` невозможно. В некоторых случаях вам может понадобиться более четкое определение структуры. С этой задачей поможет справиться еще одна функциональная возможность Elixir – *структуры*.

4.1.4. Абстракции на основе структур

Предположим, необходимо выполнить несколько различных операций с дробями (`a/b`). Наиболее эффективно это можно сделать с помощью соответствующей абстракции. Пример ниже показывает один из случаев использования такой абстракции:

```
$ iex fraction.ex
iex(1)> Fraction.add(Fraction.new(1, 2), Fraction.new(1, 4)) |>
          Fraction.value()
0.75
```

Здесь выполняется сложение двух дробей (1/2 и 1/4) и возвращается числовое значение полученной в результате дроби. Дроби создаются посредством функции `Fraction.new/2`, а затем передаются другим функциям, умеющим с ними работать.

Заметьте, что я не говорю о внутреннем представлении этих дробей, потому что для клиента это не важно: он создает экземпляр абстракции и передает его функции из соответствующего модуля.

Так как же это реализовать? Для этого существует множество способов вроде использования кортежей или словарей. Однако Elixir предоставляет *структуры* – особое средство, позволяющее определить абстракцию и привязать ее к модулю. В каждом модуле может быть определена только одна структура, которую затем можно использовать для создания новых экземпляров абстракции и в качестве образца для сопоставления.

В нашем случае дробь имеет четко определенный вид, поэтому можно использовать структуру для обеспечения корректности данных.

Посмотрим на это в действии. Объявим структуру с помощью макроса `defstruct`, как показано в следующем листинге.

Листинг 4.7 ❖ Определение структуры (fraction.ex)

```
defmodule Fraction do
  defstruct a: nil, b: nil
  ...
end
```

Указанный после макроса `defstruct` ключевой список определяет поля структуры и их начальные значения. Теперь вы можете создать экземпляр структуры, используя для этого следующий синтаксис:

```
iex(1)> one_half = %Fraction{a: 1, b: 2}
%Fraction{a: 1, b: 2}
```

Обратите внимание на то, что структура заимствует имя модуля, в котором она определена. Между модулем и структурой образуется тесная связь: структура может существовать только в модуле, а в одном модуле может быть определена только одна структура.

По сути, экземпляр структуры представляет собой словарь особого вида, поэтому обратиться к отдельным ее полям можно так же, как это делается со словарем:

```
iex(2)> one_half.a
1

iex(3)> one_half.b
2
```

Одна из приятных возможностей структур заключается в том, что с ними можно производить сопоставление:

```
iex(4)> %Fraction{a: a, b: b} = one_half
%Fraction{a: 1, b: 2}

iex(5)> a
1

iex(6)> b
2
```

С помощью сопоставления можно убедиться, что та или иная переменная является структурой:

```
iex(6)> %Fraction{} = one_half
%Fraction{a: 1, b: 2}
```

Успешное сопоставление

```
iex(7)> %Fraction{} = %{a: 1, b: 2}
** (MatchError) no match of right hand side value: %{a: 1, b: 2}
```

Словарь не соответствует структуре

Любая структура `Fraction` будет соответствовать образцу `%Fraction{}` независимо от ее содержимого. Сопоставление со структурой производится таким же образом, как и со словарем. Это означает, что при сопоставлении следует указывать только необходимые в данный момент поля и игнорировать все остальные.

Изменение структуры выполняется аналогично тому же действию со словарем:

```
iex(8)> one_quarter = %Fraction{one_half | b: 4}
%Fraction{a: 1, b: 4}
```

При создании нового экземпляра структуры за основу берется самый первый экземпляр (one_half), а значение поля b становится равным 4.

С учетом сказанного выше расширим функционал абстракции Fraction. Для начала опишем функцию создания нового экземпляра дроби.

Листинг 4.8 ❖ Создание экземпляра дроби (fraction.ex)

```
defmodule Fraction do
  ...
  def new(a, b) do
    %Fraction{a: a, b: b}
  end
  ...
end
```


Данный код – это простая обертка над синтаксисом %Fraction{}. Она позволяет сделать код более аккуратным и даже замаскировать следы использования структур.

Теперь реализуем функцию Fraction.value/1, возвращающую десятичное представление дроби.

Листинг 4.9 ❖ Вычисление значения дроби (fraction.ex)

```
defmodule Fraction do
  ...
  def value(%Fraction{a: a, b: b}) do      ← Сопоставление со структурой
    a / b
  end
  ...
end
```

Функция value/1 выполняет сопоставление со структурой, помещая значения ее полей в отдельные переменные и используя их для подсчета конечного результата. Преимуществом сопоставления с образцом является то, что тип передаваемых данных строго определен. Передать можно только экземпляр структуры Fraction, иначе сопоставление не удастся.

Вместо манипуляций с переменными можно также использовать запись с точкой:

```
def value(fraction) do
  fraction.a / fraction.b
end
```

Этот код выглядит более компактным, но чтение полей при сопоставлении будет выполняться чуть медленнее, чем в предыдущем его варианте. В большинстве случаев такое снижение производительности не сыграет большой роли, поэтому можно смело делать выбор в пользу более читабельного варианта.

И последнее, что нужно сделать, – это реализовать функцию add для сложения двух дробей.

Листинг 4.10 ❖ Сложение двух дробей (fraction.ex)

```
defmodule Fraction
  ...
```

```
  def add(%Fraction{a: a1, b: b1}, %Fraction{a: a2, b: b2}) do
    new(
      a1 * b2 + a2 * b1,
      b2 * b1
    )
  end
  ....
end
```

А теперь выполним сложение:

```
iex(1)> Fraction.add(Fraction.new(1, 2), Fraction.new(1, 4)) |>
          Fraction.value()
0.75
```

Сработало. Используя структуры для представления дробей, вы определили свой собственный тип с указанием всех полей и их значений по умолчанию. Более того, теперь экземпляры структуры легко отличить от данных другого типа, что позволяет при сопоставлении с образцом четко обозначить, что в качестве аргументов функции принимаются только экземпляры структуры.

Структуры и словари

Всегда помните, что структуры на самом деле являются словарями и обладают теми же характеристиками по части производительности и затрат памяти. Однако некоторые возможности словарей для структур недоступны. Например, передать структуру функциям `Enum` не получится:

```
iex(1)> one_half = Fraction.new(1, 2)

iex(2)> Enum.to_list(one_half)
** (Protocol.UndefinedError) protocol Enumerable not implemented for
  %Fraction{a: 1, b: 2}
```

Как вы уже знаете, структура – это функциональная абстракция, а значит, ее поведение должно соответствовать реализации модуля, в котором она определена. Что касается абстракции `Fraction`, необходимо заранее определить, будет ли она перечислением, и если да, то каким. В противном случае функции `Enum` не смогут с ней работать.

Обычный словарь является перечислением, поэтому его спокойно можно превратить в список с помощью функции `Enum.to_list`:

```
iex(3)> Enum.to_list(%{a: 1, b: 2})
[a: 1, b: 2]
```

В то же время функции модуля `Map` прекрасно работают со структурами:

```
iex(4)> Map.to_list(one_half)
[__struct__: Fraction, a: 1, b: 2]
```

Обратите внимание на запись `_struct_: Fraction`. Данная пара ключ/значение включена в каждую структуру по умолчанию. Это сделано для того, чтобы Elixir смог отличить структуры от обычных словарей и во время выполнения осуществить необходимую диспетчеризацию обобщенного кода. Это будет подробно рассмотрено далее в этой главе в разделе, посвященном протоколам.

Наличие поля `struct` накладывает определенные ограничения на сопоставление с образцом. Обычный словарь не может соответствовать образцу-структуре:

```
iex(5)> %Fraction{} = %{a: 1, b: 2}
** (MatchError) no match of right hand side value: %{a: 1, b: 2}
```

Но если поменять их местами, то все заработает:

```
iex(5)> %{a: a, b: b} = %Fraction{a: 1, b: 2}
%Fraction{a: 1, b: 2}

iex(6)> a
1

iex(7)> b
2
```

Давайте разберемся, почему так происходит. Как вы уже знаете, все поля образца должны присутствовать и в терме. При сопоставлении словаря с образцом-структурой это правило не соблюдается, так как `%Fraction{}` содержит поле `struct`, которого нет в словаре.

Напротив, когда структура сопоставляется с образцом `%{a: a, b: b}`, сопоставление проходит успешно, потому что указанные поля присутствуют в структуре `Fraction`.

Записи

Структурировать данные можно не только с помощью словарей и структур, но еще и с помощью *записей*. Они позволяют использовать кортежи и при этом иметь возможность обращаться к отдельным их элементам по имени. Записи определяются посредством макросов `defrecord` и `defrecordp` модуля `Record` (**https://hexdocs.pm/elixir/Record.html**).

Учитывая, что записи по своей сути являются кортежами, они работают быстрее словарей (хотя, по большому счету, разница не так уж и велика). С другой стороны, в реализации они не столь элегантны, и к полям нельзя обращаться динамически.

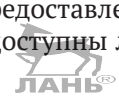
Записи существуют в Elixir главным образом по историческим причинам. Пока не появились словари, они были одним из основных инструментов структурирования данных. Что интересно, многие библиотеки экосистемы Erlang используют записи в качестве интерфейсов. Если вам необходимо подключить библиотеку Erlang с помощью записи, определенной в этой библиотеке, импортируйте эту запись в Elixir и определите ее как запись. В этом вам помогут функция `Record.extract/2` и макрос `defrecord`. Эта идиома используется нечасто, поэтому в данной книге записи подробно рассмотрены не будут. Достаточно просто на всякий случай иметь их в виду.

4.1.5. Прозрачность данных

Модули, разработанные вами ранее, считаются абстракциями, потому что клиентам не доступны подробности их реализации. Например, клиент вызывает функцию `Fraction.new/2` для создания экземпляра абстракции, а затем посылает его какой-нибудь функции этого же модуля.

Стоит отметить, что полная структура данных клиенту всегда известна. Он может получить отдельные значения структуры, даже если разработчик библиотеки этого не предполагал.

Важно понимать, что данные в Elixir по умолчанию прозрачны. Клиенты могут считывать любую информацию из структур (или любого другого типа данных), и предотвратить это крайне сложно. В этом смысле инкапсуляция работает не так, как в популярных объектно-ориентированных языках. В Elixir модули являются основным инструментом создания абстракций данных и предоставления операций для работы с этими данными, но сами данные всегда доступны любым элементам программы.



Проверим это в оболочке:

```
$ iex todo_entry_map.ex

iex(1)> todo_list = TodoList.new() |>
          TodoList.add_entry(%{date: ~D[2018-12-19], title: "Dentist"})
%{~D[2018-12-19] => [%{date: ~D[2018-12-19], title: "Dentist"}]}
```

Список дел с одним элементом

По возвращаемому значению можно с точностью определить структуру списка дел. Сразу понятно, что он строится на основе словаря, как видно и то, как хранятся отдельные его элементы.

Рассмотрим еще один пример. Экземпляр `MapSet` – тоже абстракция, созданная с помощью модуля `MapSet` и соответствующей структуры. С первого взгляда это не очевидно:

```
iex(1)> mapset = MapSet.new([:monday, :tuesday])
#MapSet<[:monday, :tuesday]>
```

Обратите внимание на то, как выглядит вывод абстракции (`#MapSet<…>`). Он имеет такой вид из-за механизма инспектирования Elixir: всякий раз, когда результат выводится в оболочке, вызывается функция `Kernel.inspect/1`, которая превращает структуру в удобочитаемую строку. Вы можете изменить стандартное поведение и обозначить собственный формат инспектирования для своих абстракций. Именно это делает MapSet, и вы научитесь этому в последующем разделе, посвященном протоколам.

В отдельных случаях вам может быть интересен реальный вид структуры данных, а не такой красиво оформленный вывод. Это может пригодиться при отладке, анализе или обратном проектировании кода. Чтобы это сделать, необходимо указать в функции `inspect` специальный параметр:

```
iex(2)> IO.puts(inspect(mapset, structs: false))
 %{__struct__: MapSet, map: %{monday: [], tuesday: []}, version: 2}
```

Теперь вывод отражает настоящий вид структуры, и вы можете без труда распознать абстракцию `MapSet`. Возможность видеть «голые» структуры доказывает, что секретность данных в функциональных абстракциях не может быть обеспечена. Как вам уже известно из второй главы, единственные сложные типы в Elixir – это кортежи, списки и словари. Любые другие абстракции, будь то `MapSet` или созданная вами `TodoList`, в конечном счете будут построены поверх этих трех типов.

Преимуществом прозрачности данных является легкое инспектирование, что особенно важно при отладке программ. Однако при разработке клиентов не стоит

полагаться на внутреннее представление данных, даже несмотря на то, что оно вам известно. Не используйте его при сопоставлении с образцом и не пытайтесь изменить или извлечь отдельные его части, потому что абстракции вроде `MapSet` не могут гарантировать получение данных в определенном формате. Единственное, что они гарантируют, – это то, что функции модуля будут успешно работать, если передать им четко структурированный экземпляр, полученный от того же модуля.

В некоторых модулях встречается общедоступная документация отдельных частей их внутренней структуры. Хорошим примером являются модули даты и времени – `Date`, `Time` и `DateTime`. Просмотрев их документацию, вы заметите явные упоминания о том, что те или иные данные представлены структурой с годом, месяцем, часом и т. д. в качестве полей. Получается, что структура даты публично задокументирована, и вы можете иметь это в виду при ее использовании.

И последнее, что вам нужно знать об инспекции данных, – это функция `IO.inspect/1`, возвращающая саму структуру и выводящая на экран более наглядное ее представление. Она особенно эффективна при отладке фрагментов кода. Рассмотрим такой пример:

```
iex(1)> Fraction.new(1, 4) |>
          Fraction.add(Fraction.new(1, 4)) |>
          Fraction.add(Fraction.new(1, 2)) |>
          Fraction.value()
1.0
```

В данном коде для выполнения ряда операций над структурой `Fraction` используется оператор конвейера. Допустим, вам необходимо выполнять проверку структуры после каждого шага. Просто добавьте вызов функции `IO.inspect/1` после каждой строки:

```
iex(2)> Fraction.new(1, 4) |>
          IO.inspect() |>
          Fraction.add(Fraction.new(1, 4)) |>
          IO.inspect() |>
          Fraction.add(Fraction.new(1, 2)) |>
          IO.inspect() |>
          Fraction.value()

%Fraction{a: 1, b: 4}
%Fraction{a: 8, b: 16}
%Fraction{a: 32, b: 32}
```

Результат вызовов функции IO.inspect

Это возможно благодаря тому, что функция `IO.inspect/1` выводит на экран более читабельную версию структуры данных, а возвращает ее же в неизмененном виде.

На этом мы закончим с теорией по функциональным абстракциям, но продолжим улучшать уже созданный список дел.

4.2. Работа с иерархическими данными

Из этого раздела вы узнаете, как добавить в абстракцию `TodoList` поддержку стандартных операций CRUD. Две из них (создание и чтение) уже реализованы с по-

мощью функций add_entry/3 и entries/1 соответственно. Остается добавить возможность обновления и удаления записей. Для этого каждая запись списка дел должна иметь уникальный идентификатор. Пожалуй, с этого и начнем.

4.2.1. Генерация идентификаторов

Сделаем так, чтобы при добавлении в список новой записи идентификатор генерировался автоматически. Для реализации этого необходимо выполнить следующие действия.

1. Преобразовать список в структуру. Это необходимо потому, что теперь в списке должны храниться два вида данных – коллекция записей и идентификатор следующей записи.
2. Сделать значение ID ключом. До этого момента при хранении записей в коллекции в качестве ключа использовалась дата. Использование значения идентификатора вместо даты позволит быстро выполнять операции вставки, обновления и удаления отдельных записей. Теперь у вас будет одно значение для каждого ключа, и абстракция MultiDict больше не понадобится.

Попробуем это осуществить. В следующем листинге приводятся определения модуля и структуры.

Листинг 4.11 ❖ Структура TodoList (todo_crud.ex)

```
defmodule TodoList do
  defstruct auto_id: 1, entries: %{}   <-- Структура, описывающая список дел

  def new(), do: %TodoList{}           <-- Создание нового экземпляра
  ...
end
```

Теперь список представлен структурой с двумя полями. Поле auto_id содержит значение идентификатора, которое будет присваиваться каждой новой записи при добавлении ее в структуру. Поле entries – коллекция записей, и, как уже было сказано, теперь это словарь со значениями идентификаторов на месте ключей.

Во время определения структуры полям auto_id и entries сразу же назначаются значения по умолчанию, поэтому при создании нового экземпляра их указывать уже не нужно. Функция new/0 создает и возвращает экземпляр структуры.

Теперь займемся реализацией функции add_entry/2. Она будет выполнять следующие задачи:

- ❍ установить ID для только что созданной записи;
- ❍ добавить новую запись в коллекцию;
- ❍ увеличить значение поля auto_id на единицу.

Вот как будет выглядеть ее код.

Листинг 4.12 ❖ Автоматическая генерация значений ID новых записей (todo_crud.ex)

```
defmodule TodoList do
  ...

  def add_entry(todo_list, entry) do
    entry = Map.put(entry, :id, todo_list.auto_id)   <-- Установка значения ID новой записи
```

```
    new_entries = Map.put(
      todo_list.entries,
      todo_list.auto_id,
      entry
    )

    %TodoList{todo_list |
      entries: new_entries,
      auto_id: todo_list.auto_id + 1
    }
  end

  ...
end
```

Добавление новой записи в список

Обновление структуры

Рассмотрим подробнее каждое действие.

Первым делом в теле функции происходит обновление значения `id` записи, хранящегося в поле `auto_id`. Для этого используется функция `Map.put/3`. Исходный словарь может не содержать поле `id`, поэтому стандартная запись `%{entry | id: auto_id}` в данном случае не подойдет.

После обновления запись добавляется в коллекцию, которая хранится в переменной `new_entries`.

Затем необходимо обновить экземпляр структуры `TodoList`, поместив значение переменной `new_entries` в поле `entries` и увеличив значение поля `auto_id` на единицу. Фактически вы вносите сразу несколько изменений в структуру – меняете значение полей и исходную запись (значение поля `id`).

Для вызывающей стороны все вышеизложенное будет выглядеть как одна неделимая операция – либо выполнятся все действия, либо, в случае ошибки, не выполнится ни одно из них. Таково следствие иммутабельности. Факт добавления новой записи становится «известным» только по завершении вызова функции `add_entry/2` и привязки ее результата к переменной. Если что-то пойдет не так, это приведет к возникновению ошибки, и результаты преобразования не будут заметны.

Стоит также отметить, что новый список дел (возвращенный функцией `add_entry/2`) и исходный список будут разделять как можно больше памяти.

После реализации функции `add_entry/2` перейдем к правке функции `entries/2`. Это будет куда сложнее, потому что внутреннее представление структуры было изменено. Ранее данные хранились в словаре с датами и записями в качестве ключей и значений соответственно. Теперь же ключами являются идентификаторы записей, а потому придется выполнить итерацию по всем записям и возвратить только те, которые соответствуют указанной дате. Реализация функции приведена в листинге 4.1.3.

Листинг 4.13 ❖ Фильтрация записей по заданной дате (todo_crud.ex)

```
defmodule TodoList do
  ...

  def entries(todo_list, date) do
    todo_list.entries
    |> Stream.filter(fn {_, entry} -> entry.date == date end)
    |> Enum.map(fn {_, entry} -> entry end)
```

Фильтрация записей по заданной дате

Извлечение только значений

```
  end

  ...
end
```

Эта функция в своей работе отталкивается от того факта, что словарь является перечислением. При использовании словарей в функциях модулей `Enum` или `Stream` каждый элемент словаря воспринимается как пара {ключ, значение}.

Функция выполняет два преобразования. Первое – выбираются только записи, соответствующие заданной дате. После этого вы получаете коллекцию кортежей {`id`, `entry`} (потому что словарь – это перечисление), и необходимо выполнить дополнительное преобразование и извлечь только вторые элементы всех кортежей.

Обратите внимание, что второе преобразование выполнено с использованием модуля `Stream`, в то время как второе – с помощью модуля `Enum`. Как уже отмечалось в третьей главе, это позволяет объединить два этих преобразования в один шаг.

Настало время проверить, как работает новая версия списка дел:

```
$ iex todo_crud.ex iex(1)> todo_list = TodoList.new() |>
          TodoList.add_entry(%{date: ~D[2018-12-19], title: "Dentist"})  |>
          TodoList.add_entry(%{date: ~D[2018-12-20], title: "Shopping"}) |>
          TodoList.add_entry(%{date: ~D[2018-12-19], title: "Movies"})

iex(2)> TodoList.entries(todo_list, ~D[2018-12-19])
[
  %{date: ~D[2018-12-19], id: 1, title: "Dentist"},
  %{date: ~D[2018-12-19], id: 3, title: "Movies"}
]
```

Код работает должным образом. В выводе даже можно увидеть идентификатор каждой записи. Также заметьте, что интерфейс модуля `TodoList` остался прежним. Вы выполнили какое-то количество внутренних преобразований, изменили представление данных и практически переписали весь модуль. Но поскольку интерфейс функций все тот же, вам не придется ничего исправлять в коде клиентов этого модуля.

Ничего особенного в этом нет – это всего лишь положительный эффект от использования абстракций данных. Пример выше демонстрирует, как может выглядеть создание типов данных высокого уровня при работе с не хранящими состояние модулями и иммутабельными данными.

4.2.2. Обновление записей

Теперь, когда у записей есть идентификаторы, можно добавить дополнительные операции для изменения данных. Реализуем функцию `update_entry` для редактирования отдельных записей списка.

Во-первых, подумаем над тем, как именно эта функция будет использоваться. Есть два возможных варианта:

- функции будет передаваться значение идентификатора и анонимная функция. Она будет работать аналогично функции `Map.update`: лямбда будет принимать на вход исходную запись, а возвращать ее измененную версию;
- функции будет передаваться словарь с записью. Если запись с таким же ID существует в коллекции записей, она будет заменена.

Если запись с указанным идентификатором отсутствует в коллекции, то есть два варианта – возбудить ошибку либо ничего не делать. В нашем случае остановимся на первом варианте (с анонимной функцией) и сделаем так, чтобы при отсутствии записи с заданным ID возникала ошибка.

Ниже можно видеть пример использования функции, а именно изменение даты записи с ID, равным 1:

```
iex(1)> TodoList.update_entry(
          todo_list,
          1,      ←——— ID обновляемой записи
          &Map.put(&1, :date, ~D[2018-12-20])    ←——— Изменение даты записи
        )
```

Реализация данной функции представлена в листинге 4.14.

Листинг 4.14 ❖ Обновление записи (todo_crud.ex)

```
defmodule TodoList do
  ...

  def update_entry(todo_list, entry_id, updater_fun) do
    case Map.fetch(todo_list.entries, entry_id) do
      :error ->     ←——— Запись не найдена – возвращается исходный список
        todo_list

      {:ok, old_entry} ->    ←——— Запись существует – изменение записи
        new_entry = updater_fun.(old_entry)                  и возвращение обновленной версии списка
        new_entries = Map.put(todo_list.entries, new_entry.id, new_entry)
        %TodoList{todo_list | entries: new_entries}
    end
  end

  ...
end
```

Давайте проанализируем каждое действие. Сначала функция `Map.fetch/2` выполняет поиск записи с указанным ID и возвращает кортеж `{:ok, value}` при успехе и `:error`, если записи не существует.

Если запись не найдена, то возвращается оригинальная версия списка, а в противном случае вызывается анонимная функция, изменяющая запись. Обновленная запись помещается в коллекцию записей. И наконец, функция помещает новую версию коллекции в экземпляр `TodoList` и возвращает его.

Щепотка сопоставления с образцом

Функция `update_entry/3` выполняет свои задачи, но не самым надежным образом. Анонимная функция может возвратить данные любого типа и даже вызвать повреждение структуры. Вы можете задать тип возвращаемых данных с помощью сопоставления с образцом, как показано ниже:

```
new_entry = %{} = updater_fun.(old_entry)
```

Как вы уже видели в третьей главе, здесь используется вложенное сопоставление, при помощи которого вы указываете, что анонимная функция должна возвращать словарь, который затем помещается в переменную `new_entry`. Если этого не происходит, возникает ошибка.

Можно пойти еще дальше и создать условие, чтобы убедиться, что анонимная функция не изменила идентификатор записи:

```
old_entry_id = old_entry.id
new_entry = %{id: ^old_entry_id} = updater_fun.(old_entry)
```

ID старой записи помещается в отдельную переменную. Затем осуществляется проверка того, что результат, возвращаемый анонимной функцией, имеет тот же ID. Напомню, что наличие фиксирующего оператора (^) в выражении сопоставления означает, что сопоставление производится со значением переменной, в данном случае – со значением переменной `old_entry_id`. Если анонимная функция обновления возвращает запись с другим ID, сопоставление не удастся, и возникнет ошибка.

С помощью сопоставления с образцом также можно изменить интерфейс обновляющей функции, как показано в следующем примере:

```
def update_entry(todo_list, %{} = new_entry) do
  update_entry(todo_list, new_entry.id, fn _ -> new_entry end)
end
```

В этом примере приведено определение функции `update_entry/2`, принимающей на вход список дел и запись в виде словаря. Она делегирует функции `update_entry/3`, которую вы только что реализовали. Корректирующая анонимная функция игнорирует старое значение и возвращает обновленную запись. Как вы должны помнить из третьей главы, функции с одинаковыми именами и разной арностью – две разные функции, то есть сейчас у вас имеется две функции `update_entry`. Какая из них будет вызвана, зависит от количества переданных при вызове аргументов.

И все же имеющаяся абстракция определена недостаточно четко. Записи, добавленные с помощью функций `TodoList.add_entry/2` и `TodoList.update_entry/2`, принимают словарь с любым содержимым, и невозможно контролировать, какая в итоге получится запись. Чтобы ввести ограничения, можно добавить специальную структуру записи (например, `TodoEntry`) и затем использовать сопоставление с образцом для проверки каждой добавляемой записи.

4.2.3. Обновление неизменяемых иерархических данных

В прошлом примере вы незаметно для себя выполнили обновление неизменяемых иерархических данных. Проанализируем, что происходит при вызове `TodoList.update_entry(todo_list, id, updater_lambda)`:

1) запись помещается в отдельную переменную;
2) вызывается корректирующая функция, которая возвращает обновленную версию записи;
3) вызывается функция `Map.put`, помещающая обновленную запись в коллекцию;
4) возвращается новая версия списка дел с обновленной коллекцией записей.

Преобразование данных происходит на шагах 2, 3 и 4. В каждом из них создается новая переменная, содержащая обновленные данные. При этом создается новая версия структуры, а исходная остается неизменной.

Именно так организуется работа с иммутабельными структурами данных. Нельзя напрямую изменить часть данных иерархической модели, хранящихся

глубоко в дереве. Необходимо добраться до той части дерева, которую требуется обновить, преобразовать ее, а затем и всех ее предков. В результате вы получите копию всей модели (списка дел). Новая и старая версии структуры будут разделять как можно больше общей памяти.

Если разделить эти операции на несколько более мелких функций, реализация будет достаточно простой и понятной. Каждая из этих функций будет изменять определенную часть структуры. При обновлении данных, хранящихся глубоко в дереве, функция находит предка самого верхнего уровня, а затем делегирует другой функции, которая выполняет всю остальную работу. Таким образом, процесс прохождения дерева будет разбит на несколько более простых операций, каждая из которых будет иметь дело с определенным уровнем абстракции.

Вспомогательные функции

Описанное выше решение может оказаться слишком громоздким при изменении более сложных иерархических структур, ведь для того, чтобы обновить какой-либо элемент глубоко в дереве, необходимо сначала дойти до него, а затем обновить всех его предков. В Elixir имеются специальные средства для упрощения этого процесса.

Рассмотрим простой пример. Внутренним представлением структуры списка дел является обычный словарь с идентификаторами в качестве ключей и словарями, хранящими поля записи, на месте значений. Создадим такой словарь:

```
iex(1)> todo_list = %{
  1 => %{date: ~D[2018-12-19], title: "Dentist"},
  2 => %{date: ~D[2018-12-20], title: "Shopping"},
  3 => %{date: ~D[2018-12-19], title: "Movies"}
}
```

Допустим, вы захотели вместо кинотеатра пойти в театр. Исходную структуру можно обновить буквально в одно действие с помощью макроса `Kernel.put_in/2`:

```
iex(2)> put_in(todo_list[3].title, "Theater")
```

Обновление иерархической структуры

```
%{
  1 => %{date: ~D[2018-12-19], title: "Dentist"},
  2 => %{date: ~D[2018-12-20], title: "Shopping"},
  3 => %{date: ~D[2018-12-19], title: "Theater"}
}
```

Заголовок записи обновлен

Как это произошло? На внутреннем уровне функция `put_in/2` делает то же самое, что вы делали до этого. Она рекурсивно добирается до необходимого элемента, изменяет его и обновляет всех его предков. Помните, что данные иммутабельны, и исходная структура остается нетронутой, а обновленная привязывается к переменной.

Чтобы выполнить обход дерева рекурсивно, необходимо передать `put_in/2` исходные данные и путь к желаемому элементу. В предыдущем примере это `todo_list` и `[3].title` соответственно. Макрос `put_in/2` пройдет по этому пути, а на обратном пути выстроит обновленную иерархию.

Стоит отметить, что Elixir предоставляет такие же возможности для извлечения данных и их изменения в виде макросов `get_in/2`, `update_in/2` и `get_and_update_in/2`.

Поскольку это макросы, передаваемый в них путь будет обработан только во время компиляции, поэтому он не может быть задан динамически.

Если вам необходимо добавить путь во время выполнения, то есть функции с таким же набором действий, принимающие данные и путь как два отдельных аргумента. Например, функция `put_in`:

```
iex(3)> path = [3, :title]

iex(4)> put_in(todo_list, path, "Theater")  ←——— Передача пути, созданного во время выполнения
```

Такие функции и макросы, как `put_in/2`, работают на основе модуля `Access`, что позволяет использовать словари и другие структуры данных ключ/значение. Чтобы ваши абстракции могли взаимодействовать с этим модулем, достаточно реализовать пару функций, запрашиваемых контрактом модуля `Access`. После этого `put_in` и другие макросы и функции «научатся» работать с вашими абстракциями. Более подробно об этом вы можете узнать из документации модуля `Access` (**https://hexdocs.pm/elixir/Access.html**).

Практика: удаление записи

Модуль `TodoList` практически готов. В нем реализованы функции создания (`add_entry/2`), извлечения (`entries/2`) и обновления (`update_entry/3`) записей. Для полной картины осталось только добавить функцию удаления (`delete_entry/2`), чем вы и займетесь самостоятельно. Это не должно составить особого труда, но если возникнут вопросы, вы всегда можете обратиться к файлу с готовым решением `todo_crud.ex`.

4.2.4. Итеративное обновление

До этого момента изменение данных выполнялось вручную, по одной правке за раз. Попробуем реализовать итеративное обновление. Предположим, имеется первичный список задач:

```
$ iex todo_builder.ex
iex(1)> entries = [
          %{date: ~D[2018-12-19], title: "Dentist"},
          %{date: ~D[2018-12-20], title: "Shopping"},
          %{date: ~D[2018-12-19], title: "Movies"}
        ]
```

Задача – создать экземпляр списка, включающий все эти задачи:

```
iex(2)> todo_list = TodoList.new(entries)
```

Очевидно, функция `new/1` должна создавать новый список итеративно. Как реализовать такую функцию? Оказывается, это совсем не сложно. Ее реализация приведена в листинге 4.15.

Листинг 4.15 ❖ Итеративное создание списка дел (todp_builder.ex)

```
defmodule TodoList do
  ...

  def new(entries \\ []) do
```

```
    Enum.reduce(
      entries,
      %TodoList{},      <-- Исходное значение аккумулятора
      fn entry, todo_list_acc ->
        add_entry(todo_list_acc, entry)   <-- Итеративное обновление аккумулятора
      end
    )
  end
  ...
end
```

Для создания списка с помощью итераций в примере использована функция `Enum.reduce/3`, которая, как вы уже знаете, преобразует перечисление в любой другой тип данных. В нашем случае первичный список экземпляров `Entry` преобразуется в структуру `Todo-List`. Для этого вызывается функция `Enum.reduce/3`, в качестве первого аргумента которой выступает исходный список, в качестве второго – новый экземпляр структуры (исходное значение аккумулятора), и третий аргумент – анонимная функция, вызываемая на каждом шаге итерации.

Лямбда вызывается для каждой записи исходного списка. Ее задачи – добавить запись к текущему аккумулятору (к структуре `TodoList`) и возвратить новое значение аккумулятора. Она делегирует существующей функции `add_entry/2`, изменяя порядок аргументов на обратный. Это необходимо потому, что, когда функция `Enum.reduce/3` вызывает лямбду, она передает ей элемент для итераций (запись) и аккумулятор (структуру `TodoList`). Функция `add_entry` же, наоборот, принимает на вход сначала структуру, а потом запись.

Помните, что определение анонимной функции можно записать в сокращенном виде при помощи оператора захвата:

```
def new(entries \\ []) do
  Enum.reduce(
    entries,
    %TodoList{},
    &add_entry(&2, &1)   <-- Изменение порядка аргументов и делегирование функции add_entry/2
  )
end
```

Вы можете выбрать любое из этих двух определений для своей анонимной функции согласно своим личным предпочтениям. Вариант с оператором захвата выглядит куда короче, но в то же время гораздо сложнее.

4.2.5. Практика: импорт из файла

Настало время немного попрактиковаться. Попробуем создать экземпляр `TodoList` из файла с разделителями-запятыми.

Пусть в текущей папке имеется файл todos.csv. Каждая строка файла описывает отдельную запись списка:

```
2018/12/19,Dentist
2018/12/20,Shopping
2018/12/19,Movies
```

Необходимо создать дополнительный модуль `TodoList.CsvImporter`, с помощью которого будет реализован механизм создания экземпляра `TodoList` из содержимого файла:

```
iex(1)> todo_list = TodoList.CsvImporter.import("todos.csv")
```

В целях упрощения задачи будем считать, что файл всегда находится в доступе в правильном формате, а также что символ запятой не встречается в заголовках записей.

В целом задача не сложная, но все же придется немного над ней посидеть. Я дам вам пару подсказок, чтобы направить ваши мысли в нужное русло.

Во-первых, создайте отдельный файл по следующему шаблону:

```
defmodule TodoList do
  ...
end

defmodule TodoList.CsvImporter do
  ...
end
```

Очень важно двигаться дальше небольшими шагами. Реализуйте часть вычислений и выведите результат на экран с помощью функции `IO.inspect/1`. Решение задачи предполагает объединение нескольких операций в цепочку, и, проверяя каждое свое действие сразу, вы гарантируете себе кратчайший путь к получению желаемого результата.

Список ваших действий должен быть примерно следующим.

1. Откройте файл и удалите все символы переноса строки (`\n`). Подсказка: используйте функции `File .stream!/1`, `Stream.map/2` и `String.replace/2`. Вы уже использовали их в третьей главе, когда пытались найти в файле все строки длиннее 80 символов.
2. Преобразуйте каждую строку в кортеж вида `{{year, month, date}, title}`. Подсказка: разделить строку на составляющие поможет функция `String.split/2`. Выделите первый элемент (дату) и извлеките из строки нужные данные. Функция `String.split/2` возвращает список строк, разделенных заданным символом. После отделения поля с датой необходимо будет преобразовать каждую строковую дату в число, для этого используйте функцию `String.to_integer/1`.
3. Создайте словарь для представления записи.
4. Созданный вами на предыдущем шаге словарь должен, в свою очередь, состоять из словарей. Передайте его в ранее реализованную функцию `TodoList.new/1`.

В каждом из описанных выше шагов функции получают на вход перечисление, преобразуют каждый его элемент и передают получившееся перечисление в качестве аргумента следующей функции. Функция `TodoList.new/1` последнего действия создает конечный список дел.

Двигаясь небольшими шагами, вы вряд ли запутаетесь. Попробуйте начать с открытия файла и вывода каждой строки на экран, а затем удалите из каждой строки символы разрыва и снова выведите результат и т. д.

Для преобразования данных вы можете использовать функции модуля `Enum` или модуля `Stream`. Пожалуй, проще всего будет начать с быстрых функций `Enum` и реализовать все с их помощью, после чего попытаться заменить как можно больше из них аналогичными функциями модуля `Stream`. Напомню, что эти функции позволяют реализовать составные итерации с отложенным выполнением в один шаг. Готовое решение можно найти в файле todo_import.ex.

Тем временем мы практически закончили изучение абстракций высокого уровня. В последнем разделе рассматривается способ реализации полиморфизма в Elixir.

4.3. Полиморфизм с помощью протоколов

Полиморфизм – это решение на основе исходных данных о том, какую часть кода необходимо выполнить, принимаемое во время выполнения программы. В Elixir основным (но не единственным) методом реализации полиморфизма являются *протоколы*.

Перед тем как приступить к изучению теории, посмотрим, как работают протоколы. Полиморфный код уже встречался вам в предыдущих примерах. Например, весь модуль `Enum`, работающий с перечислениями любого рода:

```
Enum.each([1, 2, 3], &IO.inspect/1)
Enum.each(1..3, &IO.inspect/1)
Enum.each(%{a: 1, b: 2}, &IO.inspect/1)
```

Обратите внимание на то, как каждый раз вызывается одна и та же функция `Enum.each/2`, но с разными аргументами – списком, диапазоном и словарем. Как `Enum.each/2` понимает, как нужно обрабатывать ту или иную структуру? Да никак. Код функции `Enum.each/2` является обобщенным и опирается на контракт, называемый протоколом. Этот контракт реализуется для каждого типа данных, который вы хотите использовать в функциях `Enum`. Грубо говоря, это напоминает абстрактные интерфейсы из мира ООП, но с одной загвоздкой.

Давайте посмотрим, как определять и использовать протоколы.

4.3.1. Общие сведения о протоколах

Протоколы – это модули, содержащие только объявления функций без реализации. Их можно считать аналогами абстрактных интерфейсов в объектно-ориентированных языках. Логика обобщенного кода основывается на протоколе и вызове его функций. Вы можете добавлять реализацию протокола для конкретных типов данных.

Рассмотрим пример. Протокол `String.Chars` предоставляется стандартными библиотеками Elixir и используется для преобразования данных в бинарную строку. Вот так выглядит его определение в исходном коде Elixir:

```
defprotocol String.Chars do      <-- Определение протокола
  def to_string(thing)           <-- Объявление функций протокола
end
```

Это чем-то напоминает определение модуля, не хватает разве что реализаций функций.

Посмотрите на аргумент функции (`thing`). Во время выполнения программы тип этого аргумента определяет, какую из реализаций этой функции необходимо вызвать. В Elixir уже имеются готовые реализации этого протокола для атомов, чисел и некоторых других типов данных. Можно это проверить:

```
iex(1)> String.Chars.to_string(1)
"1"

iex(2)> String.Chars.to_string(:an_atom)
"an_atom"
```

Если протокол не реализован для переданного типа, вы увидите ошибку:

```
iex(3)> String.Chars.to_string(TodoList.new())
** (Protocol.UndefinedError) protocol String.Chars not implemented
```

Обычно функцию протокола не требуется вызывать напрямую, чаще всего это осуществляется в обобщенном коде. В случае с `String.Chars` это делает функция `Kernel.to_string/1`, импортируемая по умолчанию:

```
iex(4)> to_string(1)"
1"

iex(5)> to_string(:an_atom)
"an_atom"

iex(6)> to_string(TodoList.new())
** (Protocol.UndefinedError) protocol String.Chars not implemented
```

Как видите, поведения функций `to_string/1` и `String.Chars.to_string/1` одинаковы. Все потому, что `Kernel.to_string/1` делегирует реализации `String.Chars`.

Функции `IO.puts/1` также можно передать любой тип данных, реализующий протокол:

```
iex(7)> IO.puts(1)
1

iex(8)> IO.puts(:an_atom)
an_atom

iex(9)> IO.puts(TodoList.new())
** (Protocol.UndefinedError) protocol String.Chars not implemented
```

Как видите, при попытке вывода экземпляра TodoList возникает ошибка, потому что протокол `String.Chars` для данного типа не реализован.

4.3.2. Реализация протокола

Снова обратимся к исходному коду Elixir и посмотрим, как выглядит реализация протокола для определенного типа данных. В следующем примере можно видеть реализацию `String.Chars` для строк:

```
defimpl String.Chars, for: Integer do
  def to_string(term) do
```

```
    Integer.to_string(term)
  end
end
```

Реализация всегда начинается с макроса `defimpl`, затем указывается имя протокола и соответствующий тип данных. Блок `do/end` содержит реализации всех функций протокола. В данном примере реализация делегирует функции `Integer.to_string/1` из стандартной библиотеки.

Остановимся подробнее на части `for: Type`. `Type` – это атом, который может быть любым из перечисленных псевдонимов типов: `Tuple`, `Atom`, `List`, `Map`, `BitString`, `Integer`, `Float`, `Function`, `PID`, `Port`, `Reference`. Данные псевдонимы соответствуют встроенным типам данных Elixir.

Доступен также псевдоним `Any`, позволяющий указать резервную реализацию. Если протокол для заданного типа не определен, обычно возникает ошибка, за исключением случая, когда в определении протокола указан тип `Any` и его реализация существует. Подробнее об этом можно почитать в документации макроса `defprotocol` (**https://hexdocs.pm/elixir/Kernel.html#defprotocol/2**).

И самое главное – можно задавать произвольный псевдоним типа (только не регулярные атомы):

```
defimpl String.Chars, for: SomeAlias do
  ...
end
```

Эта реализация будет вызвана в случае, если первым аргументом функции протокола (`thing`) окажется структура, определенная в указанном модуле.

Например, реализуем протокол `String.Chars` для структуры `TodoList`:

```
iex(1)> defimpl String.Chars, for: TodoList do
          def to_string(_) do
            "#TodoList"
          end
        end
```

Теперь передадим экземпляр списка функции `IO.puts/1`:

```
iex(2)> IO.puts(TodoList.new())
#TodoList
```

Важно отметить, что реализация протокола может существовать вне модулей. Это дает возможность реализовать протокол для типа, даже если его исходный код нельзя изменять. Также можно поместить реализацию протокола в любое место программы, и среда выполнения все равно будет иметь к нему доступ.

4.3.3. Встроенные протоколы

В Elixir имеется несколько встроенных протоколов. Сейчас мы рассмотрим лишь самые основные из них, а за более подробной информацией советую обратиться к онлайн-документации (**https://hexdocs.pm/elixir**).

Первый протокол вам уже знаком. Это `String.Chars`, определяющий контракт для преобразования данных в бинарную строку. Существует также протокол `List.Chars`, конвертирующий исходные данные в символьную строку (цепочку символов).

Если вам необходимо проконтролировать вывод структуры при отладке (с помощью функции `inspect`), то можете реализовать протокол `Inspect`.

Пожалуй, наиболее важным является протокол `Enumerable`. Реализовав его, вы сможете сделать свою структуру данных перечисляемой, а значит, использовать с ней любые функции модулей `Enum` и `Stream`. Думаю, это без преувеличения лучшее, что могут вам дать протоколы. Оба модуля `Enum` и `Stream` содержат обобщенный код и предоставляют множество полезных функций, которые смогут работать с созданными вами структурами данных сразу после того, как вы добавите реализацию протокола `Enumerable`.

С перечислениями тесно связан еще один протокол – `Collectable`. Напомню, что коллекция – это структура, в которую можно систематически добавлять элементы. Коллекции могут использоваться вместе с генераторами для хранения результатов каких-либо операций либо вместе с функцией `Enum.into/2` для переноса элементов из перечисляемой структуры в структуру-коллекцию.

Помните о том, что вы всегда можете определить свои собственные протоколы и реализовать их для любого типа данных (созданного вами или кем-то другим). Подробнее об этом в документации `Kernel.defprotocol/2`.

Превращение структуры в коллекцию

Рассмотрим более реальный пример. Превратим созданный ранее список дел в коллекцию, чтобы впоследствии его можно было использовать с генератором. Этот пример немного сложнее предыдущих, поэтому не переживайте, если у вас сразу не получится в нем разобраться.

Для того чтобы превратить структуру в коллекцию, необходимо реализовать соответствующий протокол:

```
defimpl Collectable, for: TodoList do
  def into(original) do
    {original, &into_callback/2}
  end

  defp into_callback(todo_list, {:cont, entry}) do
    TodoList.add_entry(todo_list, entry)
  end
  defp into_callback(todo_list, :done), do: todo_list
  defp into_callback(todo_list, :halt), do: :ok
end
```

Возвращение анонимной функции добавления

Реализация функции добавления

В данном примере приводится реализация, возвращающая анонимную функцию добавления. Экспортированная функция `into/1` вызывается в обобщенном коде (например, в генераторе). Затем многократно вызывается анонимная функция, добавляющая каждый элемент в указанную вами структуру данных.

Функция добавления принимает в качестве аргументов список дел и команду. При получении `{:cont, entry}` она создает новую запись, `:done` возвращает список, содержащий все добавленные элементы, `:halt` отменяет операцию и не возвращает ничего.

Посмотрим на это в действии. Скопируйте предыдущий пример в оболочку и попробуйте выполнить следующие действия:

```
iex(1)> entries = [
          %{date: ~D[2018-12-19], title: "Dentist"},
          %{date: ~D[2018-12-20], title: "Shopping"},
          %{date: ~D[2018-12-19], title: "Movies"}
        ]
iex(2)> for entry <- entries, into: TodoList.new(), do: entry
%TodoList{...}
```

Помещение записи в список-коллекцию

Реализовав протокол `Collectable`, вы сделали свою абстракцию `TodoList` доступной для использования в любом обобщенном коде, работающем с этим протоколом (в данном случае генераторы и функция `Enum.into/2`).

Выводы

- Для создания абстракции данных используются модули. Функции модулей создают, изменяют и извлекают данные. Внутренняя структура модулей всегда известна, но при разработке клиентов не стоит на нее полагаться.
- Словари можно использовать для объединения нескольких различных полей структуры в одно.
- Структуры – это особый вид словарей, позволяющий определить абстракции данных, связанные с модулем.
- Полиморфизм реализуется с помощью протоколов. Протокол определяет интерфейс, используемый обобщенным кодом. Вы можете добавлять специальные реализации протокола для конкретных типов данных.

Глава 5

Основы конкурентности

В главе рассматривается:

- организация конкурентного выполнения в BEAM;
- работа с процессами;
- серверные процессы с сохранением состояния;
- особенности времени выполнения.

На данный момент вы имеете достаточную базу по языку Elixir и основным идиомам программирования, и настало время переключиться на платформу Erlang и уделить немного времени на изучение функциональной возможности, играющей главную роль в обеспечении масштабирования, отказоустойчивости и распределенных вычислений в системах Elixir и Erlang, а именно организации конкурентности в виртуальной машине BEAM.

В данной главе будут рассмотрены стандартные подходы и инструменты реализации конкурентности в BEAM. Сначала вы изучите некоторые принципы высокоуровневой организации, а потом перейдете к тонкостям более низкого уровня.

5.1. Конкурентность в BEAM

Erlang – средство разработки высокодоступных систем, то есть систем, способных бесконечно долго выполнять свои задачи и реагировать на запросы пользователей. Чтобы обеспечить бесперебойную работу системы, потребуется удовлетворить ряд технических требований:

- *отказоустойчивость*: минимизировать, локализовать, а затем устранить негативные последствия ошибок времени выполнения;
- *масштабируемость*: справиться с повышенной нагрузкой путем выделения дополнительных аппаратных ресурсов без внесения изменений в код и повторного его развертывания;
- *распределенные вычисления*: организовать работу системы на нескольких компьютерах, чтобы при отказе одного компьютера другой смог его заменить.

Если вышеперечисленные требования будут удовлетворены, ваша система будет высокодоступной, предоставляя услуги пользователям почти без простоя и с минимальным количеством ошибок. В обеспечении высокой доступности особо важную роль играет конкурентность. В виртуальной машине BEAM единицей

конкурентности является *процесс* – основной структурный элемент, позволяющий создавать масштабируемые, отказоустойчивые и распределенные системы.

> **ПРИМЕЧАНИЕ** Важно понимать разницу между процессом BEAM и процессом операционной системы – первые намного более легковесные. Поскольку в этой книге речь идет в большей степени о BEAM, то термин *процесс* употребляется в значении процесса BEAM.

В условиях промышленной эксплуатации серверная система должна обрабатывать множество запросов различных клиентов одновременно, сохранять совместно используемое состояние (кеш, данные сеанса пользователя, данные на сервере (server-wide)), а также выполнять различные фоновые задачи. Чтобы сервер функционировал должным образом, все эти задачи должны решаться достаточно быстро и безотказно.

Поскольку многие задачи поступают одновременно, необходимо по максимуму распараллелить их, тем самым задействовав все возможные ресурсы ЦП. Крайне нежелательна ситуация, когда один запрос обрабатывается слишком долго, блокируя выполнение других запросов и фоновых задач. В таком случае очередь запросов будет постоянно пополняться, а система потеряет способность к отклику.

Кроме того, задачи необходимо изолировать друг от друга. Нельзя допустить, чтобы исключение, возникшее в одном из обработчиков, привело к аварийному завершению другого, не связанного с ним обработчика, фоновой задачи или, что особенно неприятно, всего сервера. Также недопустимо, чтобы в результате такого запроса с ошибкой нарушилась последовательность записи данных в памяти, что в будущем может помешать выполнению другой задачи.

Модель конкурентности BEAM поможет реализовать все вышесказанное. Процессы обеспечат параллельное выполнение задач, позволяя сделать систему *масштабируемой*, то есть способной справиться с повышенной нагрузкой путем выделения дополнительных аппаратных ресурсов.

Процессы также позволяют обеспечить изолированность задач, что, в свою очередь, обеспечивает *отказоустойчивость* системы – способность локализовать и ограничить влияние непредвиденных ошибок времени выполнения, неизбежно возникающих в любой системе. Отказоустойчивая система сохраняет свою работоспособность даже при произвольном отказе какого-либо ее компонента.

Процессы BEAM – это конкурентные потоки выполнения. Два конкурентных процесса могут быть запущены параллельно, при условии что доступны хотя бы два ядра ЦП. В отличие от процессов и потоков выполнения ОС, процессы BEAM – легковесные конкурентные сущности, чье конкурентное выполнение контролируется виртуальной машиной посредством собственного планировщика.

По умолчанию количество планировщиков, используемых виртуальной машиной BEAM, соответствует количеству доступных ядер ЦП. К примеру, при запуске кода на компьютере с четырехъядерным процессором задействуется четыре планировщика, как показано на рис. 5.1.

Каждый планировщик запускается в отдельном потоке, а сама виртуальная машина – в отдельном процессе ОС. Изображенные на рис. 5.1 четыре потока и один процесс ОС – все, что вам необходимо для обеспечения работоспособности серверной системы с высокой степенью многопоточности.

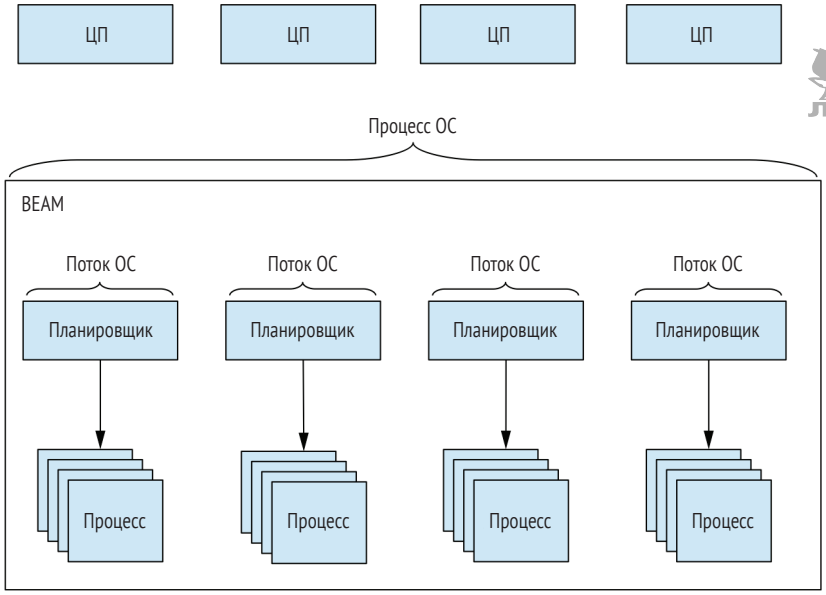


Рис. 5.1 ❖ BEAM – отдельный процесс ОС, использующий несколько потоков для упорядочения большого количества процессов

Планировщик отвечает за порядок выполнения процессов. На выполнение каждого процесса отводится определенный промежуток времени, по истечении которого этот процесс прерывается и запускается новый.

Процессы легковесны. На создание одного процесса тратится не более двух микросекунд, а первоначальный объем занимаемой им памяти составляет всего несколько килобайтов. Для сравнения: потоки ОС обычно потребляют пару мегабайтов только для хранения стека. Отсюда следует, что существует возможность создать большое количество таких процессов; теоретически предельное значение для виртуальной машины достигает 134 млн!

В частности, это может быть необходимо при разработке системы на стороне сервера для реализации управления различными задачами, выполняющимися одновременно. Используя для каждой задачи отдельный процесс, можно задействовать все возможные ресурсы ЦП и по максимуму распараллелить выполнение кода.

Более того, выполнение задач в разных процессах крайне положительно сказывается на надежности и отказоустойчивости сервера. Процессы BEAM полностью изолированы, не разделяют общую память, а аварийное завершение одного процесса никак не влияет на другие. Виртуальная машина BEAM также предоставляет возможность обнаружить ошибку и предпринять какое-либо действие, например перезапустить процесс. Все это позволяет создавать более надежные системы, способные эффективно справляться с непредвиденными ошибками, неизбежно возникающими в процессе эксплуатации.

Наконец, каждый процесс может управлять состоянием и обмениваться сообщениями с другими процессами для изменения или получения этого состоя-

ния. Как вы уже прекрасно знаете, данные в Elixir иммутабельны. Чтобы они не были удалены во время сбора мусора, необходимо постоянно к ним обращаться, передавая результат одной функции на вход другой функции. Процесс может выступать в качестве контейнера для таких данных, то есть такого места, где неизменяемая структура может храниться продолжительное время, возможно, даже бесконечно.

Как видите, конкурентность – это нечто большее, чем простое распараллеливание выполнения кода. Теперь вы имеете представление о высокоуровневой организации работы процессов виртуальной машины BEAM. Поговорим о том, как создавать процессы и работать с ними.

5.2. Работа с процессами

Преимущества процессов ощущаются в полной мере, когда необходимо реализовать параллельное выполнение задач, например отправить к базе данных несколько запросов с длительным временем выполнения. Можно выполнить их как последовательно, так и параллельно, в последнем случае надеясь на более быстрый результат.

Конкурентность и параллелизм

Важно понимать, что под конкурентностью не обязательно имеется в виду параллелизм. Две конкурентные задачи выполняются независимо друг от друга, но это не означает, что они выполняются параллельно. Параллельное выполнение двух конкурентных процессов при наличии только одного ядра ЦП невозможно. Для этого необходимо увеличить количество ядер и вооружиться надежным фреймворком с поддержкой конкурентного программирования. Имейте в виду, что сама по себе конкурентность не всегда гарантирует более быстрое выполнение задач.

В целях упрощения примера будем использовать вот такую имитацию запроса к базе данных с длительным временем выполнения:

```
iex(1)> run_query =
          fn query_def ->
            Process.sleep(2000)   <-- Имитация длительного времени выполнения
            "#{query_def} result"
          end
```

Данный код останавливается на две секунды, тем самым имитируя длительную операцию. После вызова анонимной функции `run_query` оболочка «замирает» на время выполнения функции:

```
iex(2)> run_query.("query 1")
"query 1 result"    <-- Результат через две секунды
```

Таким образом, на выполнение 10 запросов потребуется 10 секунд:

```
iex(3)> Enum.map(1..5, &run_query.("query #{&1}"))
```

```
["query 1 result", "query 2 result", "query 3 result"
 "query 4 result", "query 5 result"]
```

Результат через 10 секунд

Очевидно, о производительности и масштабируемости такого кода не может идти и речи. При условии что запросы уже оптимизированы, единственное, что можно сделать для ускорения процесса, – это выполнить их конкурентно. При этом время выполнения каждого отдельного запроса не сократится, однако общее время выполнения всех запросов значительно уменьшится. Говоря на языке BEAM, запустить что-либо конкурентно означает создать несколько отдельных процессов.

5.2.1. Создание процессов

Чтобы создать процесс, используйте функцию `spawn/1`, импортируемую по умолчанию:

```
spawn(fn ->
 expression_1
   ...
 expression_n
end)
```

Запускается в новом процессе

Функция `spawn/1` принимает на вход анонимную функцию с нулевой арностью, которая будет запущена в новом процессе. Как только создастся процесс, функция `spawn` сразу же возвращает результат, а работа вызывающего ее процесса продолжается. Указанная анонимная функция выполняется в новом процессе, а значит, выполняется конкурентно. По окончании работы функции процесс завершается, и занятая им память освобождается.

Для конкурентного выполнения запроса из предыдущего примера можно использовать следующую запись:

```
iex(4)> spawn(fn -> IO.puts(run_query.("query 1")) end)
#PID<0.48.0>
```

Результат функции spawn

```
result of query 1
```

Результат через две секунды

Как видите, функция `spawn/1` возвращает результат непосредственно после ее вызова, а пока выполняется запрос, вы можете сделать в оболочке еще что-нибудь. Через две секунды на экран выводится результат, полученный после вызова функции `IO.puts/1` в отдельном процессе.

Странная запись `#PID<0.48.0>`, возвращаемая функцией `spawn/1`, – это так называемый *идентификатор процесса* (или сокращенно *pid*). Он необходим во время обращения к процессу, как будет показано далее в этой главе.

Поработаем с конкурентностью еще немного. Для начала создадим вспомогательную анонимную функцию, которая будет конкурентно выполнять запрос и выводить результат:

```
iex(5)> async_query =
          fn query_def ->
            spawn(fn -> IO.puts(run_query.(query_def)) end)
          end
```

```
iex(6)> async_query.("query 1")
#PID<0.52.0>

result of query 1      ◄——— Результат через две секунды
```

Данный код олицетворяет характерный для Elixir прием – передачу данных созданному процессу. Обратите внимание, что функция `async_query` принимает только один аргумент и привязывает его к переменной `query_def`. Затем эти данные передаются только что созданному процессу с помощью механизма замыкания. Внутренняя лямбда-функция (та, что запускается в отдельном процессе) ссылается на переменную внешней области видимости `query_def`. Таким образом, процессы обмениваются между собой данными – содержимое переменной `query_def` передается от основного процесса новому. После этого создается глубокая копия переданных данных, поскольку два процесса не могут разделять общую память.

> **ПРИМЕЧАНИЕ** Любой код в виртуальной машине BEAM запускается в виде группы процессов, и интерактивная оболочка не исключение. Каждое выражение, вводимое вами в IEx, выполняется в специально выделенном процессе. В данном примере в качестве основного процесса выступает процесс оболочки.

Реализовав функцию `async_query`, попробуем конкурентно выполнить пять запросов:

```
iex(7)> Enum.each(1..5, &async_query.("query #{&1}"))
:ok      ◄——— Результат непосредственно после вызова

result of query 5
result of query 4
result of query 3   Результат через две секунды
result of query 2
result of query 1
```

Как и ожидалось, функция `Enum.each/2` возвращает результат незамедлительно (в предыдущем примере с последовательным выполнением запросов приходилось ждать 10 секунд). Более того, все результаты выводятся на экран практически одновременно – спустя секунды после вызова функции, что ровно в пять раз быстрее, чем в прошлый раз. Все это возможно благодаря конкурентному выполнению нескольких задач. При этом важно понимать, что порядок выполнения процессов может отличаться от заданного.

При конкурентном выполнении запросов, в отличие от последовательного, каждый из порожденных процессов выводит свой результат на экран, а вызывающий процесс выполняется отдельно и не имеет доступа к этим результатам. Помните, процессы в Elixir выполняются независимо и изолированно друг от друга.

В большинстве случаев конкурентного выполнения по принципу «запустили и забыли» (*fire and forget*), при котором вызывающий процесс не получает каких-либо уведомлений от порожденных процессов, будет вполне достаточно. Однако иногда присутствует необходимость передачи результата конкурентной операции вызывающему процессу. Для этих целей Elixir предоставляет механизм обмена сообщениями.

5.2.2. Обмен сообщениями

При разработке сложных систем часто необходимо добиться того, чтобы конкурентные задачи взаимодействовали друг с другом определенным образом. Например, имеется основной процесс, порождающий некоторое количество конкурентных процессов, а результаты всех вычислений должны быть обработаны в основном процессе.

Поскольку процессы полностью изолированы, они не могут использовать одни и те же структуры данных для обмена информацией. Поэтому они взаимодействуют между собой посредством передачи сообщений, как показано на рис. 5.2.

Если процесс А собирается сообщить процессу Б какое-то действие, он посылает ему асинхронное сообщение, содержащее терм Elixir, то есть любое выражение, которое может храниться в переменной. После отправки сообщение помещается в почтовый ящик процесса-получателя. При этом выполнение вызывающего процесса продолжается, а получатель может в любое время прочитать и обработать входящее сообщение. Так как процессы не имеют общей памяти, во время отправки сообщения осуществляется его глубокая копия.

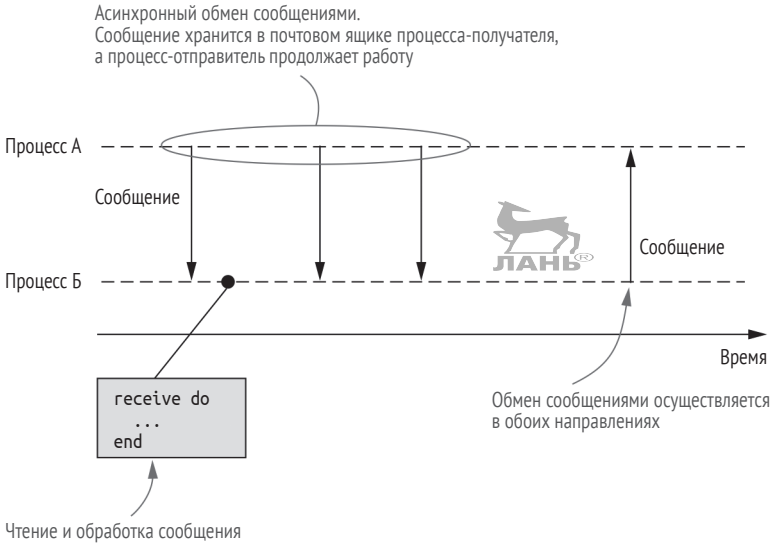


Рис. 5.2 ❖ Обмен сообщениями между процессами

Почтовый ящик процесса представляет собой простую очередь, ограниченную только объемом доступной памяти. Процесс-получатель просматривает сообщения по одному в порядке их поступления, и только после этого они могут быть удалены из очереди.

Чтобы отправить процессу сообщение, необходимо знать его идентификатор (pid). Как вам уже известно из прошлого раздела, функция `spawn/1` как раз возвращает идентификатор порожденного процесса. Получить его можно еще одним способом – вызвав автоматически импортированную функцию `self/0`.

Зная идентификатор процесса-получателя, можно отправить ему сообщение с помощью функции `Kernel.send/2`:

```
send(pid, {:an, :arbitrary, :term})
```

После вызова этой функции сообщение помещается в почтовый ящик получателя, а вызывающий процесс продолжает выполнение.

Чтобы прочесть переданное в почтовый ящик сообщение, используйте конструкцию `receive`:

```
receive do
  pattern_1 -> do_something
  pattern_2 -> do_something_else
end
```

Выражение `receive` работает аналогично выражению `case`, о котором говорилось ранее в третьей главе. Происходит следующее: сообщение считывается из почтового ящика, сопоставляется с одним из обозначенных образцов, и запускается соответствующий код. Этот алгоритм можно легко проверить, если заставить процесс оболочки отправить сообщение самому себе:

```
iex(1)> send(self(), "a message")
```

Отправка сообщения

```
iex(2)> receive do
          message -> IO.inspect(message)
        end
"a message"
```

Получение сообщения

Если вам необходимо обработать какое-либо определенное сообщение, используйте сопоставление с образцом:

```
iex(3)> send(self(), {:message, 1})

iex(4)> receive do
          {:message, id} ->
           IO.puts("received message #{id}")
        end
received message 1
```

Сопоставление сообщения с образцом

Если в почтовом ящике нет сообщений, `receive` входит в режим бесконечного ожидания нового сообщения. Это приводит к блокировке оболочки, и вам необходимо разобраться с этим вручную:

```
iex(5)> receive do
          message -> IO.inspect(message)
        end
```

Оболочка блокируется, т. к. почтовый ящик процесса пуст

То же самое происходит, когда сообщение не соответствует указанным образцам:

```
iex(1)> send(self(), {:message, 1})

iex(2)> receive do
          {_, _, _} ->
            IO.puts("received")
        end
```

Отправленное сообщение не соответствует образцу...

...поэтому оболочка блокируется

Чтобы избежать остановки оболочки, можно указать после выражения `receive` блок `after`, выполняющийся в том случае, если сообщение не было получено по прошествии определенного промежутка времени (в миллисекундах):

```
iex(1)> receive do
          message -> IO.inspect(message)
        after
          5000 -> IO.puts("message not received")
        end

message not received  <-------- Спустя пять секунд
```

Алгоритм работы блока receive

Как вы должны помнить из третьей главы, при неудачном сопоставлении с образцом возникает ошибка. Выражение `receive` является исключением из этого правила. Если сообщение не соответствует ни одному из заявленных образцов, оно помещается обратно в почтовый ящик, после чего начинает обрабатываться следующее сообщение.

Алгоритм работы блока `receive` заключается в следующем.

1. Изъять первое сообщение из почтового ящика.
2. Сопоставить его со всеми указанными образцами, начиная с первого.
3. Запустить соответствующий код, если сопоставление удалось.
4. Если сопоставление прошло неудачно, поместить сообщение обратно в почтовый ящик на то же место в очереди, а затем перейти к обработке следующего сообщения.
5. Если все сообщения в очереди уже обработаны, ожидать поступления нового сообщения, после чего перейти к п. 1 и начать обработку с первого сообщения в очереди.
6. Если указано условие `after` и сопоставление с образцом не завершилось успехом в указанное время, выполнить код, следующий после `after`.

Как вы уже знаете, каждое выражение Elixir возвращает значение, и `receive` в том числе. Результатом выражения `receive` является результат последнего выражения соответствующего предложения:

```
iex(1)> send(self(), {:message, 1})

iex(2)> receive_result =
          receive do
            {:message, x} ->
              x + 2  <-------- Результат receive
          end

iex(3)> IO.inspect(receive_result)
3
```

Итак, в блоке `receive` ищется первое (самое давнее) сообщение в почтовом ящике, соответствующее какому-либо из указанных образцов. Если такое сообщение найдено, выполняется определенный код, а в противном случае `receive` какое-то время (или бесконечно, если не прописано выражение `after`) ожидает поступления новых сообщений.

Синхронный обмен сообщениями

Основной механизм обмена сообщениями между процессами – это асинхронный обмен по принципу «запустили и забыли». Процесс посылает сообщение, а затем продолжает работу, не обращая внимания на действия получателя. В Elixir не су-

ществует специальной конструкции для запроса ответа от получателя. Необходимо заставить оба процесса общаться посредством асинхронного обмена сообщениями.

При отправке сообщения вызывающий процесс должен включить в него свой идентификатор, который затем будет использован получателем для передачи ответа, как показано на рис. 5.3. Мы рассмотрим это на примере чуть позже, когда разговор зайдет о серверных процессах.

Сохранение результатов запросов

Попробуем реализовать обмен сообщениями между конкурентными запросами из примера предыдущего раздела. Первая мысль, которая приходит на ум, – запустить каждый запрос в отдельном процессе и вывести результат на экран, после чего собрать все результаты в основном процессе.

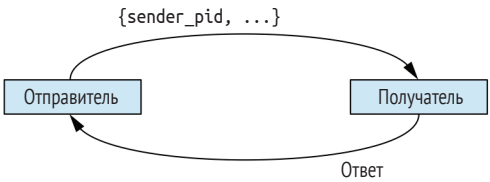


Рис. 5.3 ❖ Синхронный обмен сообщениями, реализованный с помощью асинхронного обмена

Для начала вспомним, как работает анонимная функция `async_query`:

```
iex(1)> run_query =
          fn query_def ->
            Process.sleep(2000)
            "#{query_def} result"
          end

iex(2)> async_query =
          fn query_def ->
            spawn(fn -> IO.puts(run_query.(query_def)) end)
          end
```

Сделаем так, чтобы вместо вывода результата на экран она отправляла его вызывающему процессу:

```
iex(3)> async_query =
          fn query_def ->
            caller = self()  <-- Помещение pid вызывающего процесса в переменную
            spawn(fn ->
              send(caller, {:query_result, run_query.(query_def)})  <-- Отправка ответа вызывающему процессу
            end)
          end
```

В данном примере первым делом идентификатор вызывающего процесса помещается в отдельную переменную `caller`. Это необходимо для того, чтобы рабочий процесс (тот, что выполняет вычисления) имел информацию о pid процесса, которому необходимо направить ответ.

Имейте в виду, что результат функции `self/0` зависит от вызывающего процесса. Если заранее не поместить ее результат в переменную `caller` и попытаться вызвать `send(self(), …)` во внутренней анонимной функции, то ничего не получится. Порожденный процесс отправит сообщение самому себе, т. к. `self/0` возвращает идентификатор процесса, вызывающего эту функцию.

Теперь рабочий процесс может использовать переменную `caller` для возвращения результата вычисления, при этом сообщение должно иметь специальный формат `{:query_result, result}`, позволяющий отличить его от любых других посылаемых вызывающему процессу сообщений.

Выполним запросы:

```
iex(4)> Enum.each(1..5, &async_query.("query #{&1}")
```

В результате данного вызова все запросы выполняются конкурентно, а их результаты помещаются в почтовый ящик вызывающего процесса, в данном случае процесса оболочки `IEx`.

Обратите внимание, что при получении сообщений вызывающий процесс не блокируется и вычисления не прерываются. Единственное, что происходит, – меняется содержимое почтового ящика процесса. Сообщения остаются в ящике до тех пор, пока они не будут прочитаны или процесс не будет прерван.

Давайте выведем результаты. Сначала реализуем анонимную функцию, вынимающую сообщение из почтового ящика и извлекающую из него результат запроса:

```
iex(5)> get_result =
          fn ->
            receive do
              {:query_result, result} -> result
            end
          end
```

Теперь можно создать список со всеми сообщениями из почтового ящика:

```
iex(6)> results = Enum.map(1..5, fn _ -> get_result.() end)
["query 3 result", "query 2 result", "query 1 result",
  "query 5 result", "query 4 result"]
```

Обратите внимание, что для преобразования перечисления в список той же длины использована функция `Enum.map/2`. В данном случае создается диапазон из пяти элементов, а затем он заполняется результатами функции `get_result`. Такое решение возможно лишь потому, что вы знаете количество отправленных сообщений. В противном случае в ожидании новых сообщений цикл превратился бы в бесконечный.

Стоит также отметить, что сообщения поступают в неопределенном порядке. Поскольку все вычисления выполняются конкурентно, порядок их завершения заранее неизвестен.

Давайте еще раз взглянем на финальную версию простой реализации параллельного сопоставления с образцом, которая может быть использована для одновременного выполнения большого количества задач и помещения их результатов в список:

```
iex(7)> 1..5                                    |>
        Enum.map(&async_query.("query #{&1}"))  |>  <-- Начало конкурентного выполнения
        Enum.map(fn _ -> get_result.() end)         <-- Объединение результатов в список
```

5.3. Серверные процессы с сохранением состояния

Решение разовых задач путем порождения процессов для каждой из них — не единственный пример реализации конкурентности. В Elixir довольно распространенной практикой является создание процессов с длительным временем выполнения, способных отвечать на различные сообщения. Такие процессы могут сохранять свое внутреннее состояние, а другие процессы могут его запрашивать и даже им управлять.

В этом смысле серверные процессы напоминают объекты. Они хранят состояние и могут взаимодействовать с другими процессами путем обмена сообщениями. Согласно модели конкурентности, несколько серверных процессов могут работать параллельно.

Серверные процессы являются одним из основных понятий систем Erlang/Elixir, и в последующих разделах мы остановимся на них подробнее.

5.3.1. Серверные процессы

Серверным процессом называется такой процесс, который выполняется длительное время (или бесконечно) и может обрабатывать различные запросы (сообщения). Чтобы добиться бесконечного выполнения процесса, необходимо использовать бесконечную хвостовую рекурсию. Как упоминалось ранее, если последним действием функции является вызов другой функции (или ее самой), вместо стандартной операции помещения ее параметров в стек происходит что-то вроде безусловного перехода. Следовательно, зациклившаяся функция, вызывающая саму себя, не вызовет переполнение стека и потребление дополнительной памяти.

Это особенно выгодно при реализации серверных процессов, то есть создании бесконечного цикла с ожиданием сообщений на каждом его шаге. При получении сообщения оно обрабатывается, и цикл продолжается. Попробуем сделать это, превратив пример с запросами в серверный процесс. Набросок будущего решения приведен в листинге 5.1.

Листинг 5.1 ❖ Бесконечный серверный процесс, выполняющий запросы (database_server.ex)

```
defmodule DatabaseServer do
  def start do
    spawn(&loop/0)  <-- Запуск цикла в конкурентном режиме
  end

  defp loop do
    receive do
      ...           <-- Обработка сообщения
    end

    loop()          <-- Продолжение работы цикла
```

```
  end
  ...
end
```

Функция `start/0` – это так называемая *функция интерфейса*, используемая клиентами для запуска бесконечного серверного процесса. Это достигается при помощи приватной функции `loop/0`, в задачи которой входит ожидание и обработка сообщения, а также вызов самой себя, что в результате создает бесконечный цикл. При этом подобный цикл не оказывает интенсивной нагрузки на ядро процессора, так как ожидание сообщения приостанавливает выполняющийся процесс, благодаря чему не затрачиваются дополнительные машинные циклы.

Обратите внимание, что функции этого модуля запускаются в разных процессах. Функция `start/0` вызывается клиентами и выполняется в клиентском процессе, а приватная функция `loop/0` – в серверном процессе. И это совершенно нормально, ведь модули и процессы никак не связаны между собой. Модуль – это просто коллекция функций, которые могут быть вызваны в любых процессах.

При работе над серверным процессом имеет смысл помещать весь связанный с ним код в отдельный модуль. Функции этого модуля обычно можно разделить на две категории – функции интерфейса и функции реализации. *Функции интерфейса* являются публичными, выполняются в вызывающем процессе и маскируют информацию о создании процесса и протоколе взаимодействия. *Функции реализации* обычно приватные и запускаются в серверном процессе.

> **ПРИМЕЧАНИЕ** Как и в случае с классическими циклами, вам не придется создавать рекурсивный цикл самостоятельно. Для этого в Elixir существует стандартная абстракция `GenServer` (обобщенный серверный процесс), упрощающая разработку серверных процессов с сохранением состояния. В ее основе лежит все та же рекурсия, но реализована она в самом `GenServer`. Подробнее эта тема рассматривается в следующей главе.

А теперь рассмотрим полную реализацию функции `loop/0`.

Листинг 5.2 ❖ Цикл сервера базы данных (database_server.ex)

```
defmodule DatabaseServer do
  ...

  defp loop do
    receive do                                    ← Ожидание сообщения
      {:run_query, caller, query_def} ->
        send(caller, {:query_result, run_query(query_def)})   ← Отправка запроса и передача ответа вызывающему процессу
      end

    loop()
  end

  defp run_query(query_def) do
    Process.sleep(2000)                           | Выполнение запроса
    "#{query_def} result"
  end

  ...
end
```

В данном коде можно наблюдать протокол взаимодействия между вызывающим процессом и сервером базы данных. Вызывающий процесс посылает сообщение в формате {:run_query, caller, query_def}, а серверный процесс обрабатывает его путем выполнения запроса и передачи его результата обратно вызывающему процессу.

Как правило, подробности о структуре переданных и полученных сообщений обычно должны быть неизвестны клиенту. Чтобы их скрыть, лучше всего использовать функцию интерфейса. Введем функцию run_async/2, с помощью которой клиенты смогут выполнять ту или иную операцию с сервером, в данном случае запрос к серверу. Клиент будет просто вызывать функцию run_async/2 и получать необходимый результат, не имея представления о формате сообщений, которыми обмениваются между собой процессы. Реализация функции приведена в листинге 5.3.

Листинг 5.3 ❖ Реализация функции run_async/2 (database_server.ex)

```
defmodule DatabaseServer do
  ...
  def run_async(server_pid, query_def) do
    send(server_pid, {:run_query, self(), query_def})
  end
  ...
end
```

В функцию run_async/2 передается pid серверного процесса и запрос, который необходимо выполнить. Единственное, что она делает, – это отправляет серверу соответствующее сообщение. Вызов run_async/2 клиентом предполагает, что серверный процесс начнет выполнение запроса во время работы клиента.

После выполнения запроса сервер посылает сообщение вызывающему процессу. Чтобы получить этот результат, реализуем еще одну функцию интерфейса – get_result/0.

Листинг 5.4 ❖ Реализация функции get_result/0 (database_server.ex)

```
defmodule DatabaseServer do
  ...

  def get_result do
    receive do
      {:query_result, result} -> result
    after
      5000 -> {:error, :timeout}
    end
  end

  ...
end
```

Функция get_result/0 вызывается, когда клиенту необходимо получить результат запроса. Для ожидания и получения сообщения используется выражение receive, а условие after останавливает ожидание по прошествии определенного количества времени. Это имеет смысл, если, например, во время выполнения запроса что-то пошло не так и ответ не отправился.

Сервер базы данных готов. Попробуем поработать с ним:

```
iex(1)> server_pid = DatabaseServer.start()

iex(2)> DatabaseServer.run_async(server_pid, "query 1")
iex(3)> DatabaseServer.get_result()
"query 1 result"

iex(4)> DatabaseServer.run_async(server_pid, "query 2")
iex(5)> DatabaseServer.get_result()
"query 2 result"
```

Посмотрите, как несколько запросов обрабатываются в одном и том же процессе. Сначала запускается первый запрос, затем еще один. Это доказывает, что серверный процесс продолжает работать даже после получения сообщения.

Клиенту не виден формат сообщений, ведь они обернуты в функции, поэтому он общается с сервером с помощью функций. В этом общении очень важную роль играет идентификатор серверного процесса. Он становится известным после вызова `DatabaseServer.start/0`, после чего используется для отправки запросов серверу.

В серверном процессе запрос обрабатывается асинхронно. Вызвав `DatabaseServer.run_async/2` в оболочке, вы можете продолжать работать в ней и получить результат запроса тогда, когда он вам нужен.

Последовательность действий в серверных процессах

Важно понимать, что серверный процесс представляет собой единый поток выполнения. Это цикл, обрабатывающий одно сообщение за раз. Поэтому, если отправить пять асинхронных запросов одному и тому же серверному процессу, они будут обработаны по очереди, а результат последнего из них будет получен спустя 10 секунд.

И это очень даже неплохо, так как помогает планировать работу системы. Серверный процесс можно считать местом синхронизации. Если необходимо выполнить несколько действий синхронно, одно за другим, можно создать один серверный процесс, передать в него все запросы, и он обработает их по очереди.

Очевидно, в данном случае особенность последовательного выполнения действий совершенно не к месту. Чтобы получить результат как можно быстрее, несколько запросов должно быть выполнено конкурентно. Как же быть?

При условии что запросы никак не связаны друг с другом, вы можете создать несколько серверных процессов, а затем для выполнения каждого запроса выбирать один процесс. Если таких процессов будет достаточно много и вам удастся равномерно распределить на них нагрузку, то задачи будут выполняться одновременно и максимально быстро.

Перенесем эти мысли на код. Для начала создадим пул процессов сервера базы данных:

```
iex(1)> pool = Enum.map(1..100, fn _ -> DatabaseServer.start() end)
```

Данное выражение создает 100 процессов и сохраняет их идентификаторы в список. Может показаться, что такое количество – это слишком много, но, как вы помните, процессы в Elixir являются легковесными, они занимают небольшое количество памяти (около 2 Кб) и создаются очень быстро (в течение нескольких

микросекунд). Более того, все эти процессы будут заняты ожиданием сообщения, а значит, будут находиться в состоянии бездействия, не растрачивая ресурсы ЦП.

Отправив запрос, необходимо каким-то образом решить, какой именно процесс будет его выполнять. Наиболее простой способ – использовать функцию `:rand.uniform/1`, принимающую на вход положительное число и возвращающую произвольное число в диапазоне [1..N]. В следующем выражении с помощью этой функции выполнение каждого из пяти запросов поручается одному из 100 возможных процессов:

```
iex(2)> Enum.each(
          1..5,
          fn query_def ->
            server_pid = Enum.at(pool, :rand.uniform(100) - 1)
            DatabaseServer.run_async(server_pid, query_def)
          end
        )
```

Выбор произвольного процесса

Выполнение запроса

Очевидно, что это решение не отличается эффективностью. Для получения pid произвольного процесса используется функция `Enum.at/2`. Так как для их хранения используется список, а поиск произвольного его элемента – операция со сложностью O(N), это негативно сказывается на производительности. Гораздо лучше было бы использовать вместо списка словарь с индексами процессов на месте ключей и их идентификаторами на месте значений. Существуют и другие возможные решения, среди которых можно выделить карусельное распределение. Но пока остановимся на простом решении со списком.

После того как запросы распределятся по рабочим процессам, необходимо подумать о получении их результатов. Реализуется это довольно просто:

```
iex(3)> Enum.map(1..5, fn _ -> DatabaseServer.get_result() end)
["5 result", "3 result", "1 result", "4 result", "2 result"]
```

Так как запросы выполняются конкурентно, их результаты будут получены намного быстрее.

5.3.2. Сохранение состояния процесса

Использование серверного процесса предоставляет возможность сохранять его состояние. Например, при обращении к базе данных необходимо иметь некий дескриптор соединения, который «знает», как организуется общение с сервером. Если серверный процесс реализует обмен данными по TCP, он должен хранить информацию о соответствующем сокете.

Чтобы сохранить состояние процесса, можно добавить в функцию `loop` один или несколько дополнительных аргументов:

```
def start do
  spawn(fn ->    initial_state = ...
                 loop(initial_state)
  end)
end

defp loop(state) do
```

Инициализация состояния во время создания процесса

Передача состояния функции loop

```
  ...
  loop(state)  ◄──── Сохранение состояния во время выполнения цикла
end
```

Попробуем таким же способом создать подключение к серверу баз данных. Для имитации дескриптора соединения будем использовать случайное число. Для начала необходимо инициализировать подключение во время запуска процесса, как показано в примере далее.

Листинг 5.5 ❖ Инициализация состояния процесса (stateful_database_server.ex)

```
defmodule DatabaseServer do
  ...
  def start do
    spawn(fn ->      connection = :rand.uniform(1000)
      loop(connection)
    end)
  end
  ...
end
```

В данном примере сначала устанавливается соединение, а затем соответствующий дескриптор передается функции `loop`. В реальности для установки соединения вместо генерации случайного числа используются интерфейсы доступа к базам данных вроде ODBC.

Теперь перепишем функцию `loop`.

Листинг 5.6 ❖ Использование подключения во время выполнения запроса (stateful_database_server.ex)

```
defmodule DatabaseServer do
  ...

  defp loop(connection) do
    receive do
      {:run_query, from_pid, query_def} ->
        query_result = run_query(connection, query_def)  ◄── Использование подключения во время выполнения запроса
        send(from_pid, {:query_result, query_result})
    end

    loop(connection)  ◄──── Сохранение состояния в аргументе функции loop
  end

  defp run_query(connection, query_def) do
    Process.sleep(2000)
    "Connection #{connection}: #{query_def} result"
  end

  ...
end
```

Идея довольно проста. Функция `loop` отвечает за сохранение состояния (connection), принимая его в качестве первого аргумента, а доработанная функция `run_query` использует подключение во время выполнения запроса к базе данных. Дескриптор подключения (в данном случае число) добавляется к результату запроса.

Итак, теперь можно сказать, что сервер базы данных готов. Заметьте, интерфейсы его публичных функций остались теми же, значит, и использоваться они будут так же. Посмотрим, как он работает:

```
iex(1)> server_pid = DatabaseServer.start()

iex(2)> DatabaseServer.run_async(server_pid, "query 1")
iex(3)> DatabaseServer.get_result()
"Connection 753: query 1 result"

iex(4)> DatabaseServer.run_async(server_pid, "query 2")
iex(5)> DatabaseServer.get_result()
"Connection 753: query 2 result"
```

Результаты двух разных запросов получены с указанием одного и того же дескриптора, который сохраняется внутри выполняющего цикл процесса и остается невидимым для остальных процессов.

5.3.3. Изменяемое состояние

В предыдущем примере показано сохранение постоянного состояния процесса, но его можно запросто сделать изменяемым:

```
def loop(state) do
  new_state =
    receive do          | Вычисление нового состояния
      msg1 ->           | в зависимости от сообщения
        ...

      msg2 ->
        ...
    end

  loop(new_state)   <---- Запуск цикла с новым состоянием
end
```

Именно так в Elixir выглядит стандартный подход к реализации сервера с сохранением состояния. Процесс ожидает сообщения, а затем, в зависимости от его содержимого, вычисляется новое состояние. И наконец, запускается рекурсивный цикл с этим состоянием, тем самым изменяя старое состояние на новое. При обработке следующего сообщения уже будет использовано новое состояние.

С точки зрения внешней структуры, сохраняющие состояние процессы являются изменяемыми. Посылая такому процессу сообщение, вызывающая сторона может оказать влияние на его состояние и результат соответствующего запроса, что может привести к возникновению побочных эффектов. Но не стоит забывать, что используемые сервером структуры данных иммутабельны. Состояние может быть представлено любой переменной Elixir, начиная с обычных чисел и заканчивая сложными абстракциями, такими как `TodoList` (из четвертой главы).

Посмотрим на это в действии. Начнем с простого примера – реализуем калькулятор, в качестве состояния сохраняющий число. Начальное состояние процесса равно 0, а изменять его можно с помощью запросов `add`, `sub`, `mul` и `div`. Для получения состояния процесса используется запрос `value`.

Полученный сервер можно использовать следующим образом:

```
iex(1)> calculator_pid = Calculator.start()
iex(2)> Calculator.value(calculator_pid)
0
iex(3)> Calculator.add(calculator_pid, 10)
iex(4)> Calculator.sub(calculator_pid, 5)
iex(5)> Calculator.mul(calculator_pid, 3)
iex(6)> Calculator.div(calculator_pid, 5)
iex(7)> Calculator.value(calculator_pid)
3.0
```

Запуск процесса

Проверка начального значения

Выполнение запросов

Проверка текущего значения

Начинаем с запуска процесса и проверки его начального состояния. Затем выполняем несколько запросов для изменения этого состояния и проверяем результат операций `(((0 + 10) - 5) * 3) / 5`, который равен 3.

Теперь перейдем непосредственно к реализации данного сервера. Первым делом пропишем его внутренний цикл.

Листинг 5.7 ❖ Конкурентный калькулятор с сохранением состояния (calculator.ex)

```
defmodule Calculator do
  ...

  defp loop(current_value) do
    new_value =
      receive do
        {:value, caller} ->
          send(caller, {:response, current_value})
          current_value

        {:add, value} -> current_value + value
        {:sub, value} -> current_value - value
        {:mul, value} -> current_value * value
        {:div, value} -> current_value / value

        invalid_request ->
          IO.puts("invalid request #{inspect invalid_request}")
            current_value
        end

    loop(new_value)
  end

  ...
end
```

Запрос на получение состояния

Запросы на выполнение арифметических операций

Неподдерживаемый тип запросов

Цикл обрабатывает различные сообщения. Сообщение `:value` используется для получения состояния сервера. Вызывающий процесс должен поместить в сообщение свой pid, чтобы потом ему можно было отправить ответ. Обратите внимание, последнее выражение этого блока возвращает текущее значение состояния `current_value`. Это необходимо, потому что результат блока `receive` помещается в переменную `new_value` и затем используется как новое состояние сервера. С помощью возвращения `current_value` можно убедиться, что запрос `:value` не изменяет состояние процесса.

Арифметические операции вычисляют новое состояние на основе текущего значения и полученного в сообщении аргумента. В отличие от дескрипторов сообщений :value, дескрипторы арифметических операций не посылают ответ вызывающему процессу. Благодаря этому данные операции могут быть запущены асинхронно, а как именно, вы увидите немного позже, когда займетесь реализацией функций интерфейса.

Последнее предложение выражения receive соответствует всем остальным сообщениям, которые не поддерживаются сервером. Они выводятся на экран вместе с текущим значением состояния current_value, не изменяя его.

Следующий шаг – реализация функций интерфейса, которые будут использовать клиенты. Она приведена в листинге 5.8.

Листинг 5.8 ❖ Функции интерфейса калькулятора (calculator.ex)

```
defmodule Calculator do
  def start do
    spawn(fn -> loop(0) end)
  end

  def value(server_pid) do
    send(server_pid, {:value, self()})
    receive do
      {:response, value} ->
        value
    end
  end

  def add(server_pid, value), do: send(server_pid, {:add, value})
  def sub(server_pid, value), do: send(server_pid, {:sub, value})
  def mul(server_pid, value), do: send(server_pid, {:mul, value})
  def div(server_pid, value), do: send(server_pid, {:div, value})
  ...
end
```

Запуск сервера и инициализация состояния

Запрос значения

Арифметические операции

Функции интерфейса имеют простую реализацию и соответствуют протоколу, указанному в функции loop/1. Запрос типа :value – пример синхронного обмена сообщениями, о котором говорилось в разделе 5.2.2. В нем нет ничего особенного – всего лишь отправка сообщения и ожидание незамедлительного ответа. Важно то, что это ожидание приостанавливает вызывающий процесс до того момента, пока не будет получен ответ. Именно это и делает такой обмен синхронным.

А вот арифметические операции выполняются асинхронно. Ответ не отправляется, а значит, вызывающий процесс не тормозится ожиданием. Поэтому он может отправить несколько таких запросов и продолжить свою работу, в то время как операции будут выполняться в серверном процессе конкурентно. Помните, что сообщения обрабатываются сервером в порядке их поступления, а значит, и запросы тоже.

Зачем выполнять арифметические операции асинхронно? Потому что время их выполнения не важно. Пока состояние сервера не будет запрошено (функцией value/1), клиент не будет приостановлен, что положительно сказывается на производительности.

Реорганизация цикла

Чем больше запросов вы посылаете серверу, тем более сложной становится функция `loop`, и в какой-то момент она станет настолько необъятной, что будет напоминать огромное выражение типа `switch/case`.

Попробуем реорганизовать код этой функции: добавим сопоставление с образцом и вынесем блок обработки сообщений в отдельную функцию с несколькими предложениями. Теперь код функции `loop` выглядит простым и аккуратным:

```
defp loop(current_value) do
  new_value =
    receive do
      message -> process_message(current_value, message)
    end

  loop(new_value)
end
```

В этом коде можно видеть общую схему работы сервера. Сообщение поступает на сервер, затем оно обрабатывается. Обработка главным образом заключается в том, чтобы вычислить новое состояние на основе предыдущего состояния и содержимого этого сообщения. И в завершение запускается цикл с новым состоянием вместо старого.

`process_message/2` – простая функция с несколькими предложениями, принимающая на вход текущее состояние и сообщение. Ее задача – выполнить операции над полученным сообщением и возвратить новое состояние:

```
defp process_message(current_value, {:value, caller}) do
  send(caller, {:response, current_value})
  current_value
end

defp process_message(current_value, {:add, value}) do
  current_value + value
end
...
```

Итак, мы провели небольшую реорганизацию цикла серверного процесса. Теперь он выглядит более сжато, а обработка сообщений вынесена в отдельную функцию с несколькими предложениями, каждое предложение которой отвечает за обработку определенного рода сообщений.

5.3.4. Сложные состояния

Состояние обычно представляет собой нечто более сложное, чем обычное число. Однако принцип работы с ним не меняется – оно сохраняется с помощью приватной функции `loop`. Чем сложнее состояние, тем более громоздким становится код серверного процесса, поэтому отличным решением будет вынести логику управления состоянием в отдельный модуль, а в задачах серверного процесса оставить только передачу сообщений и сохранение состояния.

Рассмотрим реализацию вышесказанного на примере абстракции `TodoList` из четвертой главы. Для начала напомним себе, как использовалась данная абстракция:

```
iex(1)> todo_list = TodoList.new() |>
          TodoList.add_entry(%{date: ~D[2018-12-19], title: "Dentist"}) |>
          TodoList.add_entry(%{date: ~D[2018-12-20], title: "Shopping"}) |>
          TodoList.add_entry(%{date: ~D[2018-12-19], title: "Movies"})

iex(2)> TodoList.entries(todo_list, ~D[2018-12-19])
[
  %{date: ~D[2018-12-19], id: 1, title: "Dentist"},
  %{date: ~D[2018-12-19], id: 3, title: "Movies"}
]
```

Как вы помните, `TodoList` – это полностью функциональная абстракция. Чтобы она продолжала существовать, необходимо постоянно обращаться к результату последней операции со структурой.

В следующем примере приводится реализация модуля, хранящего данную абстракцию в приватном состоянии. Рассмотрим такой случай использования:

```
iex(1)> todo_server = TodoServer.start()

iex(2)> TodoServer.add_entry(todo_server,
                             %{date: ~D[2018-12-19], title: "Dentist"})

iex(3)> TodoServer.add_entry(todo_server,
                             %{date: ~D[2018-12-20], title: "Shopping"})

iex(4)> TodoServer.add_entry(todo_server,
                             %{date: ~D[2018-12-19], title: "Movies"})

iex(5)> TodoServer.entries(todo_server, ~D[2018-12-19])
[
  %{date: ~D[2018-12-19], id: 3, title: "Movies"},
  %{date: ~D[2018-12-19], id: 1, title: "Dentist"}
]
```

В данном примере сначала запускается сервер, затем с помощью его идентификатора осуществляются преобразования данных. По сравнению с функциональным подходом, здесь не нужно передавать результат предыдущей операции в виде аргумента следующей. Вместо этого для всех действий над списком дел используется одна и та же переменная `todo_server`.

В некотором смысле переменную `todo_server` можно воспринимать в качестве ссылки на объект или указателя из объектно-ориентированных языков. Результаты ваших действий на сервере будут сохраняться до тех пор, пока запущен сервер, однако операции с сервером, в отличие от операций с объектами, выполняются конкурентно.

Перейдем к реализации этого сервера. Прежде всего поместите все модули в один файл, как показано в следующем листинге.

Листинг 5.9 ❖ Модули TodoServer (todo_server.ex)

```
defmodule TodoServer do
  ...
end

defmodule TodoList do
  ...
end
```

Сделав это, вы можете легко получить доступ к обоим модулям, просто подгрузив один файл в оболочку. В более сложных системах для этого обычно используется проект mix, что будет рассмотрено более подробно в седьмой главе.

Будем использовать прежнюю реализацию абстракции TodoList, представленную в четвертой главе. В ней есть все необходимое для организации работы серверного процесса.

Построим базовую структуру серверного процесса.

Листинг 5.10 ❖ Базовая структура TodoServer (todo_server.ex)

```
defmodule TodoServer do
  def start do
    spawn(fn -> loop(TodoList.new()) end)
  end

  defp loop(todo_list) do
    new_todo_list =
      receive do
        message -> process_message(todo_list, message)
      end

    loop(new_todo_list)
  end
  ...
end
```

Использование списка дел в качестве состояния

В общем-то, ничего нового. Сначала запускается функция цикла, в которую передается новый экземпляр структуры TodoList в качестве начального состояния. Полученные сообщения обрабатываются в цикле, и функция process_message/2 изменяет состояние и возвращает его в обновленном виде. И последним действием идет запуск цикла с новым состоянием.

Для каждого поддерживаемого типа запроса необходимо добавить в функцию process_message/2 соответствующее предложение, а также реализовать отдельную функцию интерфейса. Реализуем все это сначала для запроса add_entry.

Листинг 5.11 ❖ Запрос add_entry (todo_server.ex)

```
defmodule TodoServer do
  ...
  def add_entry(todo_server, new_entry) do
    send(todo_server, {:add_entry, new_entry})
  end
  ...
  defp process_message(todo_list, {:add_entry, new_entry}) do
     TodoList.add_entry(todo_list, new_entry)
  end

  ...
end
```

Функция интерфейса

Обработка сообщения

Функция интерфейса передает серверу новые входные данные. Напомню, что функция loop вызывает функцию process_message/2, которая в данном случае деле-

гирует функции TodoList.add_entry/2 и возвращает обновленный экземпляр TodoList. Этот экземпляр затем становится новым состоянием сервера.

Аналогичным образом можно реализовать запрос entries, но не забудьте добавить ожидание ответа. Код запроса приведен в листинге 5.12.

Листинг 5.12 ❖ Запрос entries (todo_server.ex)

```
defmodule TodoServer do
  ...

  def entries(todo_server, date) do
    send(todo_server, {:entries, self(), date})

    receive do
      {:todo_entries, entries} -> entries
    after
      5000 -> {:error, :timeout}
    end
  end

  ...

  defp process_message(todo_list, {:entries, caller, date}) do
    send(caller, {:todo_entries, TodoList.entries(todo_list, date)})
    todo_list
  end

  ...
end
```

Отправка ответа вызывающему процессу

Состояние не изменяется

В данном примере переплетается несколько описанных ранее приемов. Процесс посылает сообщение, после чего находится в ожидании ответа. Соответствующее предложение функции process_message/2 делегирует TodoList, после чего вызывающему процессу отправляется ответ и исходное состояние сервера. Это необходимо, потому что функция loop/2 принимает на вход в качестве нового состояния результат функции process_message/2.

Точно так же вы можете добавить в свой сервер поддержку запросов update_ entry и delete_entry. Предлагаю вам сделать это самостоятельно в качестве упражнения.

Какой подход выбрать: конкурентный или функциональный?

Процесс, сохраняющий изменяемое состояние, в каком-то смысле можно считать изменяемой структурой данных. Вы можете запустить сервер и выполнить на нем несколько запросов, но не стоит принципиально отказываться от использования функционального метода обработки неизменяемых данных в процессах.

Модели данных должны быть построены на основе чистых функциональных абстракций, как это было сделано с TodoList. Это позволяет добиться целостности и неделимости данных, а также предоставляет возможность их независимого тестирования и повторного использования в различных контекстах.

Получается, что процесс используется поверх функциональных абстракций в качестве своего рода конкурентного контроллера, он сохраняет состояние и может быть использован другими процессами системы в целях управления или частичного считывания этого состояния.

Например, при разработке веб-сервера для управления несколькими списками дел лучше всего выделить отдельный серверный процесс для каждого списка. Во время обработки HTTP-запроса будет осуществляться поиск соответствующего процесса и затем проведение в нем заданной операции. Каждое действие со списком будет выполняться конкурентно, что позволит добиться большей производительности и масштабируемости сервера. Кроме этого, вы не столкнетесь с проблемами синхронизации, так как каждый список будет обрабатываться в отдельном процессе. Как вы помните, задачи в процессе выполняются последовательно, поэтому несколько равнозначных запросов к одному и тому же списку упорядочиваются и обрабатываются по очереди. Не переживайте, если эта информация сейчас кажется слишком тяжеловесной. Это будет рассмотрено на примерах в главе 7.

5.3.5. Регистрация процессов

Чтобы один процесс мог взаимодействовать с другими, ему необходимо знать их местонахождение. В BEAM каждому процессу присваивается уникальный идентификатор. Чтобы процесс А мог обмениваться сообщениями с процессом Б, ему необходимо передать идентификатор процесса Б. В этом смысле идентификатор напоминает ссылку или указатель, характерные для мира ООП.

В некоторых ситуациях передача и хранение идентификаторов загромождают код. Если вы планируете создать только один экземпляр сервера того или иного типа, то можете дать процессу *локальное имя* и указывать его при обмене сообщениями с другими процессами. Имя называется локальным, потому что оно имеет значение только в текущем экземпляре BEAM. Это особенно важно в случаях, когда необходимо подключить несколько экземпляров BEAM к распределенной системе, как будет показано в главе 12.

Процесс можно *зарегистрировать* с помощью функции `Process.register(pid, name)`, при этом его имя должно быть указано в виде атома. Вот небольшой пример:

```
iex(1)> Process.register(self(), :some_name)
```
Регистрация процесса

```
iex(2)> send(:some_name, :msg)
```
Передача сообщения с использованием имени

```
iex(3)> receive do
          msg -> IO.puts("received #{msg}")
        end
received msg
```
Проверка факта получения сообщения

К зарегистрированным именам предъявляется ряд требований:

- имя может быть представлено только атомом;
- процесс может иметь только одно имя;
- два процесса не могут иметь одинаковые имена.

Если эти условия не удовлетворены, возникает ошибка.

Попробуйте немного попрактиковаться, поменяв в предыдущем примере обычный процесс на зарегистрированный. Интерфейс сервера упростится, так как ему больше не нужно хранить и передавать pid.

Рассмотрим один из возможных случаев использования такого сервера:

```
iex(1)> TodoServer.start()

iex(2)> TodoServer.add_entry(%{date: ~D[2018-12-19], title: "Dentist"})
```

```
iex(3)> TodoServer.add_entry(%{date: ~D[2018-12-20], title: "Shopping"})
iex(4)> TodoServer.add_entry(%{date: ~D[2018-12-19], title: "Movies"})

iex(5)> TodoServer.entries(~D[2018-12-19])
[%{date: ~D[2018-12-19], id: 3, title: "Movies"},
 %{date: ~D[2018-12-19], id: 1, title: "Dentist"}]
```

Для того чтобы у вас получилось так же, зарегистрируйте серверный процесс под каким-нибудь именем (например, `:todo_server`). Затем перепишите функции интерфейса, чтобы в них при отправке сообщений процессу использовалось зарегистрированное имя. Если у вас возникнут трудности, вы всегда можете обратиться к файлу registered_todo_server.ex.

Работать с зарегистрированным серверным процессом гораздо проще, поскольку вам больше не нужно хранить pid и передавать его функциям интерфейса. Вместо этого для обмена сообщениями функции интерфейса на внутреннем уровне используют зарегистрированное имя.

Регистрация процессов играет очень важную роль. Благодаря зарегистрированным именам для взаимодействия процессов идентификаторы больше не нужны, и процессы проще обнаружить. Это приобретает практическую ценность при повторном запуске процессов (как показано в главах 8 и 9) и разработке распределенных систем (объясняется в главе 12).

На этом и закончим знакомство с сохраняющими состояние процессами. Они являются важнейшей частью Elixir-систем, поэтому вы продолжите использовать их в дальнейших примерах этой книги. А теперь настало время поговорить о нескольких самых важных особенностях времени выполнения процессов BEAM.

5.4. Особенности времени выполнения

Вы уже знаете немало информации о том, как работают процессы в Elixir. Чтобы иметь более полное представление о конкурентной модели виртуальной машины BEAM, предлагаю немного погрузиться в изучение ее внутреннего устройства. Не волнуйтесь, мы не будем копать слишком глубоко, а лишь рассмотрим самые важные моменты.

5.4.1. Последовательность выполнений действий в процессах

Об этом уже не раз говорилось ранее, но все же стоит еще раз отметить: несколько процессов может быть запущено параллельно, но в рамках одного процесса все действия выполняются последовательно, процесс либо выполняет какой-то код, либо находится в ожидании сообщений. Если на один процесс свалится множество сообщений от других, это может существенным образом снизить общую скорость обработки сообщений.

Рассмотрим в качестве примера реализацию эхо-сервера с низкой скоростью обработки запросов.

Листинг 5.13 ❖ Демонстрация узкого места процесса (Process_bottleneck.ex)

```
defmodule Server do
  def start do
```

```
    spawn(fn -> loop end)
  end

  def send_msg(server, message) do
    send(server, {self(), message})
    receive do
      {:response, response} -> response
    end
  end

  defp loop do
    receive do
      {caller, msg} ->
        Process.sleep(1000)   <-- Имитация длительной обработки
        send(caller, {:response, msg})   <-- Отправка сообщения обратно
    end

    loop()
  end
end
```

Получив сообщение, сервер отправляет его обратно вызывающей стороне, перед этим останавливаясь на секунду для имитации длительной обработки.

Теперь создадим пять конкурентных клиентов:

```
iex(1)> server = Server.start()

iex(2)> Enum.each(
          1..5,
          fn i ->
            spawn(fn ->   <-- Порождение конкурентного клиента
              IO.puts("Sending msg ##{i}")
              response = Server.send_msg(server, i)   <-- Передача серверу сообщения
              IO.puts("Response: #{response}")
            end)
          end
        )
```

Запустив этот код, вы увидите на экране следующие строки:

```
Sending msg #1
Sending msg #2
Sending msg #3
Sending msg #4
Sending msg #5
```

Пока все идет по плану. Пять процессов запущены конкурентно. Но вот незадача: ответы от сервера приходится ждать слишком долго – мы получаем по одному ответу в секунду:

```
Response: 1   <-- Через одну секунду
Response: 2   <-- Через две секунды
Response: 3   <-- Через три секунды
Response: 4   <-- Через четыре секунды
Response: 5   <-- Через пять секунд
```

Почему так происходит? Эхо-сервер может обрабатывать только одно сообщение в секунду, а так как все остальные процессы зависят от него, они ограничиваются его пропускной способностью.

Что с этим делать? При обнаружении узкого места необходимо оптимизировать процесс изнутри. Как правило, поток выполнения сервера довольно прост: он получает и обрабатывает сообщения по одному за раз. Ваша задача – сделать так, чтобы он обрабатывал сообщения незамедлительно после их поступления. В данном примере оптимизация будет заключаться в удалении вызова `Process.sleep/1`.

В том случае если вам не удается ускорить обработку сообщений, попробуйте добавить еще несколько отдельных процессов и тем самым распараллелить поступающий поток задач, что должно увеличить производительность при использовании многоядерной системы. Увы, это единственное, что вы можете сделать. Однако это не поможет заставить плохо продуманный алгоритм работать эффективнее.

5.4.2. Бездонные почтовые ящики процессов

Теоретически объем почтовых ящиков процессов не фиксирован, а на практике он ограничивается объемом доступной памяти. Поэтому если процесс постоянно запаздывает, то есть сообщения поступают быстрее, чем он может их обработать, объем его почтового ящика будет постоянно увеличиваться и поглощать все больше памяти. В итоге, заняв всю доступную память, всего один такой процесс может привести к аварийной остановке всей системы.

У этой проблемы есть и более сложный случай, при котором сервер вообще не обрабатывает сообщения. Посмотрите на следующий цикл:

```
def loop
  receive do
    {:message, msg} -> do_something(msg)
  end

  loop()
end
```

Сервер с таким циклом будет обрабатывать только сообщения вида `{:message, something}`. Все остальные сообщения будут оставаться в почтовом ящике, просто занимая память.

Кроме того, почтовые ящики с большим количеством сообщений отрицательно сказываются на производительности. Вспомните, как работает сопоставление с образцом в выражении `receive`: сообщения анализируются по очереди от старых к новым до тех пор, пока не будет найдено то из них, которое соответствует образцу. Допустим, у процесса есть миллион необработанных сообщений. Когда приходит новое, `receive` сначала проходит весь миллион старых сообщений и только потом доходит до нового. Соответственно, на это теряется большое количество времени, и производительность сервера снижается.

Чтобы избавиться от данной проблемы, вам необходимо ввести в блок `receive` *универсальное условие*, которое будет обрабатывать нестандартные типы сооб-

щений. Достаточно просто обозначить, что получено сообщение неизвестного формата:

```
def loop
  receive
    {:message, msg} -> do_something(msg)
    other -> log_unknown_message(other)    <-- Универсальное условие
  end

  loop()
end
```

Теперь сервер сможет обрабатывать любые сообщения, помечая сообщения нестандартного вида.

Стоит также отметить, что BEAM предоставляет инструменты для анализа процессов во время выполнения. Например, вы можете запросить у каждого процесса информацию о количестве находящихся в почтовом ящике сообщений и тем самым выяснить, в каких из них очередь переполнена. Это будет подробно рассмотрено в главе 13.

5.4.3. Конкурентность без разделения ресурсов

Как отмечалось ранее, процессы не имеют общей памяти, поэтому при обмене сообщениями создается глубокая копия их содержимого:

```
send(target_pid, data)    <-- Создание глубокой копии переменной data
```

Кроме того, использование замыкания с переменной в теле функции `spawn` также приводит к созданию глубокой копии переменной:

```
data = ...

spawn(fn ->
  ...
  some_fun(data)    <-- Создается глубокая копия переменной data
  ...
end)
```

Стоит учитывать это при вынесении части кода в отдельный процесс. Глубокое копирование происходит в оперативной памяти довольно быстро, и передача объемного сообщения не должна вызывать проблем. А вот если многие процессы часто обмениваются друг с другом объемными сообщениями, производительность системы может снизиться. Понятие размера сообщений в данном случае весьма субъективно. Очевидно, что такие простые типы данных, как числа, атомы и кортежи с несколькими элементами, не требуют много памяти, в отличие от списков из миллионов сложных структур. Предельное значение лежит где-то посередине и зависит от конкретного случая.

Исключением из этого правила являются бинарные данные (в том числе и строки) размером более 64 байт. Глубокого копирования при обмене такими данными не производится, вместо этого они хранятся в отдельной общей динамической памяти. Это особенно выгодно, когда необходимо отправить что-либо большому количеству процессов, в которых не требуется расшифровка строк.

В чем же заключается смысл конкурентного программирования без разделения ресурсов? Прежде всего упрощается код каждого отдельно взятого процесса. Поскольку процессы не имеют общей памяти, вам не понадобится использовать сложные механизмы синхронизации вроде блокировок и взаимных исключений. Следующее преимущество – стабильность всей системы: один процесс не имеет доступа к памяти других процессов. В свою очередь, это обеспечивает целостность и отказоустойчивость системы. И наконец, конкурентное программирование без разделения ресурсов позволяет осуществлять эффективный сбор мусора.

Так как процессы не имеют общей памяти, сбор мусора может осуществляться на уровне процесса. Каждому процессу изначально выделяется небольшой объем динамической памяти (около 2 Кб на 64-битной машине BEAM). Когда процессу требуется дополнительная память, в отведенный ему для выполнения промежуток времени происходит еще и сбор мусора для этого процесса. Таким образом, вместо одного общего сбора мусора, останавливающего всю систему на какое-то время, производится несколько более мелких операций, что делает систему более отзывчивой. Можно поручить одному планировщику освободить память для конкретного процесса, в то время как остальные продолжат выполнять свои задачи.

5.4.4. Внутреннее устройство планировщиков

Каждый планировщик BEAM на самом деле представляет собой поток операционной системы, управляющий выполнением процессов BEAM. По умолчанию виртуальная машина использует планировщики по числу доступных логических процессоров. Настройки можно менять по своему усмотрению с помощью различных флагов эмулятора Erlang.

Для этого используется следующий синтаксис:

```
$ iex --erl "put Erlang emulator flags here"
```

Список всех флагов Erlang можно найти по ссылке **http://erlang.org/doc/man/erl.html**.

Обычно предполагается, что n планировщиков управляют m процессами, где m почти всегда значительно больше n. Такая схема называется *m:n-поточностью*, и она означает, что большее количество логических микропотоков запускается при использовании меньшего количества потоков ОС, как показано на рис. 5.4.

Изнутри каждый планировщик представляет собой очередь выполнения – не что иное, как список процессов BEAM, которыми он управляет. Каждому процессу на выполнение отводится определенный временной промежуток, после чего он ставится на паузу и запускается другой процесс. Такой промежуток равнозначен приблизительно 2000 вызовам функций (называемых *редукциями*).

Elixir – функциональный язык. Программы на нем состоят из простых и быстрых функций, а потому переключение контекста происходит очень часто, обычно в течение менее одной миллисекунды. Благодаря этому системы на основе BEAM остаются отзывчивыми. Если один процесс выполняет длительную операцию, как, например, высчитывание биллионов знаков после запятой у числа пи, планировщик не будет ждать ее завершения, а начнет выполнять другие процессы в очереди.

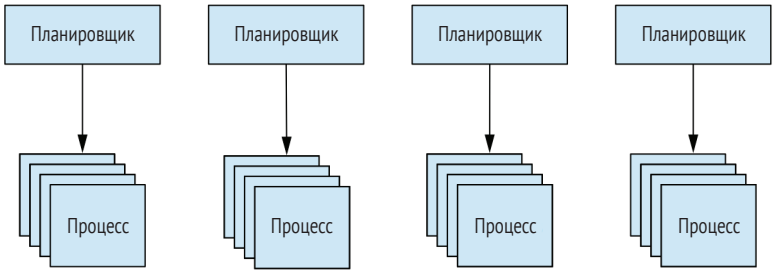


Рис. 5.4 ❖ m:n-поточность: несколько планировщиков контролируют работу большого количества процессов BEAM

Бывают особые ситуации, когда процесс может передать управление планировщику до того, как отведенное ему время закончится. Наиболее примечательные из них – это использование выражения `receive` и вызов функции `Process.sleep/1`. В обоих случаях выполняющий их процесс приостанавливается, и планировщик запускает следующие процессы.

Стоит также отметить и операции ввода-вывода, которые выполняются в разных потоках, называемых *асинхронными*. При вызове функции ввода-вывода вызывающий процесс замирает, передавая инициативу другим процессам. По завершении операции ввода-вывода планировщик возобновляет выполнение вызывающего процесса. Благодаря такой схеме выглядит так, будто код ввода-вывода работает синхронно, когда на самом деле он выполняется асинхронно. По умолчанию BEAM запускает 10 асинхронных потоков, но вы можете поменять эти настройки с помощью Erlang-флага `+A n`.

Вы также можете воспользоваться службами опроса ядра вроде *epoll* или *kqueue* для осуществления ввода-вывода без блокировки процесса, если они поддерживаются вашей системой. Для этого при запуске BEAM укажите флаг `+K true`.

У неявной передачи инициативы есть еще одно преимущество. Если большинство процессов постоянно стоит на паузе (например, когда ядро выполняет ввод-вывод или пока эти процессы находятся в ожидании сообщений), планировщики становятся еще более эффективными и повышают общую производительность системы.

Выводы

- Процесс виртуальной машины BEAM – легковесная конкурентная единица выполнения. Процессы полностью изолированы и не имеют общей памяти.
- Процессы могут взаимодействовать между собой посредством асинхронных сообщений. Синхронные передача и получение реализуются вручную на основе этого базового механизма.
- Серверный процесс – это процесс, выполняющийся длительное время (или бесконечно) и обрабатывающий различного рода сообщения. Реализация серверных процессов строится на бесконечной рекурсии.
- Серверные процессы могут сохранять собственное состояние, передавая его в рекурсивный вызов в качестве аргумента.

Глава 6

Обобщенные серверные процессы

В главе рассматривается:

- создание обобщенных серверных процессов;
- использование `GenServer`.

В предыдущей главе вы познакомились с основными приемами конкурентного программирования. Вы научились создавать процессы и организовывать их взаимодействие, а также узнали, что серверные процессы могут сохранять собственное состояние, реагировать на сообщения, обрабатывать данные, посылать ответ и даже изменять свое состояние.

Серверные процессы играют ключевую роль в Elixir и Erlang и очень часто используются при разработке систем с высокой степенью конкурентности, поэтому придется уделить им еще немного времени, чтобы изучить их во всех подробностях. Из этой главы вы узнаете, как избавиться от шаблонного кода в серверных процессах, а именно в реализации бесконечной рекурсии, управления состоянием и обмена сообщениями.

Erlang предоставляет вспомогательный модуль для реализации серверных процессов, являющийся частью фреймворка под названием OTP (открытая телекоммуникационная платформа). Несмотря на свое название, он никак не связан с телекоммуникационными системами. Он предоставляет шаблоны и абстракции для создания компонентов, сборки релизов, разработки серверных процессов, обработки и устранения последствий ошибок времени выполнения, журналирования, управления событиями и обновления кода.

В этой книге вы познакомитесь с различными составляющими OTP, а в данной главе мы остановимся на одной из наиболее важных из них – `GenServer` – модуле, упрощающем код серверных процессов. Прежде чем перейти к его изучению, попробуем реализовать упрощенную версию серверного процесса на основе примитивов, рассмотренных в пятой главе.

6.1. Создание обобщенного серверного процесса

В предыдущей главе вы могли наблюдать некоторые примеры серверных процессов. Несмотря на то что они были созданы для абсолютно разных целей, в их реа-

лизациях просматриваются общие моменты. По сути, любой северный процесс должен:

- порождать новый процесс;
- запускать в нем бесконечный цикл;
- хранить его состояние;
- реагировать на сообщения;
- отправлять ответ вызывающему процессу.

Независимо от предназначения серверного процесса, от него обязательно потребуется выполнение этих задач, поэтому разумно будет вынести его код в отдельное место. Конкретные реализации смогут в будущем повторно использовать его для решения определенных задач. Давайте посмотрим, как создать такой обобщенный код.

6.1.1. Подключение к обобщенному коду с помощью модулей

Обобщенный код будет решать различные задачи, характерные для серверных процессов, предоставляя право на определенные решения конкретным реализациям. Например, обобщенный код породит процесс, а его начальное состояние должно быть определено в конкретной реализации. Аналогичным образом обобщенный код будет отвечать за получение и отправку ответа, а конкретная реализация – за то, каким образом сообщение должно быть обработано и какой именно ответ должен быть отправлен.

Другими словами, обобщенный код организует базовую работу процесса, а конкретная реализация восполняет недостающие детали. В связи с этим необходимо иметь некий механизм подключения, позволяющий обобщенному коду обращаться к конкретной реализации, когда ей необходимо принять то или иное решение.

Проще всего для этих целей использовать модули. Как вы знаете, имя модуля представлено атомом. Этот атом можно поместить в переменную и в дальнейшем использовать ее для вызова функций этого модуля:

```
iex(1)> some_module = IO   <-- Сохранение имени модуля в переменной
iex(2)> some_module.puts("Hello")   <-- Динамический вызов
Hello
```

Вы можете использовать такой способ для использования функций обратного вызова в обобщенном коде, а конкретно следовать порядку действий, описанному ниже.

1. Сделать так, чтобы обобщенный код принимал имя подключаемого модуля в качестве аргумента. Такой модуль будет называться *модулем обратного вызова*.
2. Хранить атом модуля как часть состояния процесса.
3. Обращаться к функциям модуля обратного вызова, когда это необходимо.

Очевидно, чтобы это заработало, в модуле обратного вызова должен быть реализован и экспортирован определенный набор функций, которые вы изучите далее по мере наполнения примера функциональностью.

6.1.2. Реализация обобщенного кода

Итак, начнем работать над обобщенным серверным процессом. Прежде всего необходимо запустить процесс и инициализировать его состояние, как показано в следующем листинге.

Листинг 6.1 ❖ Запуск серверного процесса (server_process.ex)

```
defmodule ServerProcess do
  def start(callback_module) do
    spawn(fn ->
      initial_state = callback_module.init()
      loop(callback_module, initial_state)
    end)
  end

  ...
end
```

Использование обратного вызова для инициализации состояния

Функция `ServerProcess.start/1` принимает имя модуля в виде атома в качестве аргумента и порождает процесс, в котором для создания начального состояния используется функция обратного вызова `init/0`. Безусловно, необходимо иметь реализацию этой функции в подключаемом модуле, чтобы данный код заработал. И последнее действие – запуск цикла, поддерживающего работоспособность серверного процесса и сохраняющего его состояние. Возвращаемым значением функции `ServerProcess.start/1` является идентификатор процесса, который можно использовать для общения с другими процессами.

Теперь перейдем к реализации цикла, обеспечивающего функционирование процесса, ожидающего и обрабатывающего сообщения. В данном примере будем использовать шаблон синхронного обмена сообщениями – когда серверный процесс получает сообщение, обрабатывает его, отправляет ответ вызывающей стороне и изменяет состояние процесса.

Обобщенный код будет отвечать за получение и отправку сообщений, тогда как конкретная реализация – за обработку полученного сообщения, отправку ответа и новое состояние. Код этого решения приведен в листинге 6.2.

Листинг 6.2 ❖ Обработка сообщений в серверном процессе (server_process.ex)

```
defmodule ServerProcess do
  ...
  defp loop(callback_module, current_state) do
    receive do
      {request, caller} ->
        {response, new_state} =
          callback_module.handle_call(
            request,
            current_state
          )

        send(caller, {:response, response})

        loop(callback_module, new_state)
    end
```

Использование функции обратного вызова для обработки сообщения

Передача ответа

Запуск цикла с новым состоянием

```
  end
  ...
end
```

В данном случае формат ожидаемых сообщений – это кортеж вида {request, caller}. Элемент request содержит данные, идентифицирующие запрос и имеющие значение для конкретной реализации. Функция обратного вызова handle_call/2 принимает на вход эти данные и текущее состояние процесса, а возвращает кортеж вида {response, new_state}. После этого обобщенный код передает ответ вызывающему процессу (по требованию) и запускает цикл с новым состоянием.

И последнее, что нужно сделать, – это добавить функцию для отправки запросов серверу, реализация которой показана в следующем примере.

Листинг 6.3 ❖ Вспомогательная функция отправки запросов (server_process.ex)

```
defmodule ServerProcess do
  ...
  def call(server_pid, request) do
    send(server_pid, {request, self()})   <-- Передача сообщения

    receive do
      {:response, response} ->   <-- Ожидание ответа
        response   <-- Отправка ответа
      end
  end
end
```

Итак, теперь у вас есть абстракция обобщенного серверного процесса. Давайте посмотрим, как можно ее использовать.

6.1.3. Использование обобщенной абстракции

Чтобы опробовать сервер в действии, реализуем простое хранилище типа ключ/значение. Это будет процесс, используемый для хранения соответствий между произвольными термами.

Напоминаю, что модуль обратного вызова должен содержать реализации двух функций – init/0, создающей начальное состояние, и handle_call/2, обрабатывающей определенные запросы. Код модуля приведен далее.

Листинг 6.4 ❖ Реализация хранилища ключ/значение (server_process.ex)

```
defmodule KeyValueStore do
  def init do
    %{}   <-- Начальное состояние процесса
  end

  def handle_call({:put, key, value}, state) do
    {:ok, Map.put(state, key, value)}
  end   <-- Обработка запроса типа put

  def handle_call({:get, key}, state) do
    {Map.get(state, key), state}
  end   <-- Обработка запроса типа get
end
```

Вот, собственно, и все. Благодаря тому что шаблонный код бесконечного цикла и обмена сообщениями вынесен в отдельные модули, конкретная реализация серверного процесса выглядит гораздо более емкой и сосредоточена на выполнении своей основной задачи.

Обратите особое внимание на обработку запросов различных типов в функции с несколькими предложениями `handle_call/2`. Именно здесь принимается решение о том, как серверу обработать тот или иной запрос. Модуль `ServerProcess` состоит из обобщенного кода, который слепо передает запросы от клиентских процессов к модулю обратного вызова.

Проверим, как работает сервер:

```
iex(1)> pid = ServerProcess.start(KeyValueStore)

iex(2)> ServerProcess.call(pid, {:put, :some_key, :some_value})
:ok

iex(3)> ServerProcess.call(pid, {:get, :some_key})
:some_value
```

Серверный процесс запускается с помощью выражения `ServerProcess.start(KeyValueStore)`. Именно в нем происходит подключение конкретной реализации `KeyValueStore` к обобщенному коду `ServerProcess`. Два последующих вызова `ServerProcess.call/2` посылают сообщения серверному процессу, который, в свою очередь, вызывает функцию `KeyValueStore.handle_call/2` для их обработки.

На практике важно исключить любые упоминания об используемой абстракции `ServerProcess` в коде клиентов. Это можно сделать путем ввода вспомогательных функций, как показано ниже.

Листинг 6.5 ❖ Обертки над функциями модуля `ServerProcess` (server_process.ex)

```
defmodule KeyValueStore do
  def start do
    ServerProcess.start(KeyValueStore)
  end

  def put(pid, key, value) do
    ServerProcess.call(pid, {:put, key, value})
  end

  def get(pid, key) do
    ServerProcess.call(pid, {:get, key})
  end

  ...
end
```

Теперь для выполнения различных действий с хранилищем клиент может использовать функции `start/0`, `put/3` и `get/2`. Эти функции также называют *функциями интерфейса*. Клиенты могут использовать функции интерфейса модуля `KeyValueStore` для запуска серверного процесса и взаимодействия с ним.

Функции обратного вызова `init/0` и `handle_call/2`, напротив, используются обобщенным кодом на внутреннем уровне. Запомните: функции интерфейса выполняются в клиентских процессах, а функции обратного вызова – в серверных.

6.1.4. Поддержка асинхронных запросов

Текущая версия сервера ServerProcess способна обрабатывать только синхронные запросы. Настало время расширить его функционал, добавив поддержку асинхронных запросов типа «запустили и забыли», когда клиент посылает сообщение серверу и не ожидает ответа.

В мире OTP существуют два типа запросов: *call* и *cast* (синхронный и асинхронный соответственно), и далее по тексту будет использовано то же именование.

Итак, нам необходимо реализовать запрос типа cast, а значит, нужно изменить формат сообщений, передаваемых от клиентов к серверу. Это позволит серверу отличить запросы разного типа и в соответствии с этим обработать их по-разному.

Для этого можно просто добавить информацию о типе запроса в кортеж, передаваемый от клиента серверу, как показано ниже.

Листинг 6.6 ❖ Добавление типа запроса в сообщение (server_process_cast.ex)

```
defmodule ServerProcess do
  ...
  def call(server_pid, request) do
    send(server_pid, {:call, request, self()})    <-- Указание типа запроса (call)
    ...
  end

  defp loop(callback_module, current_state) do
    receive do
      {:call, request, caller} ->    <-- Обработка запроса типа call
        ...
    end
  end

  ...
end
```

Теперь добавим поддержку запросов типа cast. В рамках текущего примера, когда сообщение попадает на сервер, конкретная реализация должна обработать его и возвратить новое состояние. Ответ вызывающей стороне не отправляется, поэтому функция обратного вызова должна возвратить только обновленное состояние. Реализация этого приведена в листинге 6.7.

Листинг 6.7 ❖ Добавление в серверный процесс поддержки запросов типа cast (server_process_cast.ex)

```
defmodule ServerProcess do
  ...
  def cast(server_pid, request) do
    send(server_pid, {:cast, request})    <-- Указание типа запроса (cast)
  end

  defp loop(callback_module, current_state) do
    receive do
      {:call, request, caller} ->
        ...
      {:cast, request} ->    <-- Обработка сообщения с меткой cast
        new_state =
```

```
          callback_module.handle_cast(
            request,
            current_state
          )

        loop(callback_module, new_state)
    end
  end

  ...
end
```

Для того чтобы обработать запрос типа cast, необходимо использовать функцию обратного вызова `handle_cast/2`. Она обработает сообщение и возвратит обновленное состояние, после чего будет вызвана в функции `loop`, и цикл запустится с новым состоянием. Вот и все, поддержка cast-запросов добавлена.

И наконец, перепишем код клиента с использованием запросов типа cast. Помните, что cast-запросы не требуют ответа от сервера и поэтому не подходят для всех возможных запросов. В текущем примере запросы типа `get` должны быть синхронными, поскольку в ответ на такие запросы серверу необходимо предоставить пару ключ/значение. Запросы типа `put`, напротив, будут асинхронными, так как клиенту ответ не нужен.

Листинг 6.8 ❖ Реализация запроса `put` в виде запроса `cast` (server_process_cast.ex)

```
defmodule KeyValueStore do
  ...

  def put(pid, key, value) do
    ServerProcess.cast(pid, {:put, key, value})   <-- Выполнение запроса put как запроса cast
  end

  ...

  def handle_cast({:put, key, value}, state) do   <-- Обработка put-запроса
    Map.put(state, key, value)
  end

  ...
end
```

Проверим работу сервера:

```
iex(1)> pid = KeyValueStore.start()

iex(2)> KeyValueStore.put(pid, :some_key, :some_value)

iex(3)> KeyValueStore.get(pid, :some_key)
:some_value
```

Внеся пару изменений в обобщенный код, вы добавили своему серверному процессу функциональности. Конкретные реализации теперь могут представить любой запрос либо как cast, либо как call.

6.1.5. Упражнение: реорганизация сервера для списка дел

Важным преимуществом обобщенной абстракции `ServerProcess` является возможность создавать различные конкретные реализации серверных процессов,

используют один и тот же код для решения разных задач. Например, в пятой главе вы разработали простенький сервер для списка дел, сохраняющий абстракцию списка дел в качестве внутреннего состояния. Этот сервер также можно реализовать на основе `ServerProcess`.

У вас появилась отличная возможность попрактиковаться. Скопируйте весь код из файла `todo_server.ex` и сохраните его в отдельный файл. Добавьте в этот же файл последнюю версию модуля `ServerProcess`. Ваша задача – переписать модуль `TodoServer` так, чтобы он мог взаимодействовать с `ServerProcess`.

Как закончите, сравните код до и после. Новая версия сервера `TodoServer` должна быть проще и короче, даже при условии поддержки сервером всего двух типов запросов. При возникновении трудностей вы можете обратиться к исходнику `server_process_todo.ex`.

> **ПРИМЕЧАНИЕ** Помещать несколько модулей в один и тот же файл и иметь несколько копий кода серверного процесса в разных файлах – не самое лучшее решение. В главе 7 вы научитесь более правильно организовывать свой код с помощью инструмента `mix`. А пока пожертвуем красотой во имя простоты.

Базовая абстракция обобщенного серверного процесса готова. Текущая ее реализация проста до безобразия, но она демонстрирует основные принципы создания серверов на Elixir. Настало время познакомиться с `GenServer` – полноценной OTP-абстракцией обобщенного серверного процесса.

6.2. Использование GenServer

В условиях реальной промышленной разработки бессмысленно с нуля создавать абстракции вроде `ServerProcess`, ведь Elixir предлагает готовое и более продуманное в плане функционала решение под названием `GenServer`, покрывающее большинство возможных случаев использования и проверенное на практике в сложных конкурентных системах.

Вот лишь некоторые из наиболее заметных особенностей `GenServer`:

- поддержка запросов типа call и cast;
- настраиваемое время ожидания ответа для call-запросов;
- оповещение ожидающих ответа клиентских процессов об аварийном завершении сервера;
- поддержка распределенных систем.

Хочу заметить, что `GenServer` не появился из ниоткуда магическим образом. В основе его кода лежат рассмотренные в пятой главе примитивы конкурентности и особые методы обеспечения отказоустойчивости, о которых вы узнаете в главе 9. `GenServer` полностью реализован на Elixir и Erlang. Решение всех самых сложных задач производится модулем `:gen_server`, входящим в стандартную библиотеку Erlang. Также в модуле `GenServer` стандартной библиотеки Elixir реализованы некоторые дополнительные обертки.

В данном разделе вы научитесь создавать серверные процессы с помощью `GenServer`, но для начала следует рассмотреть такое понятие, как поведение OTP.

6.2.1. Поведения OTP

Согласно терминологии Erlang, *поведение* – это обобщенный код, реализующий часто используемый шаблон. Поведения обычно оформляются в отдельный модуль, к которому подключается соответствующий модуль обратного вызова. Модуль обратного вызова должен удовлетворять определенному в поведении контракту, то есть включать в себя реализации некоторого набора функций и быть способным их экспортировать. Вы можете вызвать эти функции в модуле поведения, тем самым определив специализацию своего обобщенного кода.

Именно так и работает модуль `ServerProcess`. Он обеспечивает общий функционал серверного процесса, а конкретная его реализация предоставляет модуль обратного вызова с функциями `init/0`, `handle_call/2` и `handle_cast/2`. `ServerProcess` является простым примером поведения.

Существует возможность определить контракт поведения и проверить на этапе компиляции, реализует ли модуль обратного вызова требуемые функции. Более подробно об этом читайте в официальной документации Elixir (**https://hexdocs.pm/elixir/Module.html#module-behaviour-notice-the-british-spelling**).

Стандартная библиотека Erlang включает в себя следующие поведения OTP:

- `gen_server` – обобщенная реализация серверного процесса с сохранением состояния;
- `supervisor` – поведение для обработки ошибок и устранения их последствий в конкурентных системах;
- `application` – обобщенная реализация компонентов и библиотек;
- `gen_event` – поддержка обработки событий;
- `gen_statem` – поведение для запуска конечного автомата в серверном процессе с сохранением состояния.

Elixir предоставляет обертки для наиболее часто используемых поведений Erlang в виде модулей `GenServer`, `Supervisor` и `Application`, которые подробно освещаются в данной книге в главах 7, 8–9 и 11 соответственно. Остальные поведения не менее эффективны, но используются не так часто и рассмотрены в рамках книги не будут. Когда вы научитесь пользоваться `GenServer` и `Supervisor`, вам не составит труда в случае необходимости освоить и другие поведения самостоятельно. Для получения более подробной информации о поведениях `gen_event` и `gen_statem` обратитесь к документации Erlang (**http://erlang.org/doc/design_principles/des_princ.html**).

6.2.2. Подключение к GenServer

Грубо говоря, `GenServer` используется так же, как и `ServerProcess`. Формат возвращаемых значений несколько отличается, но основная идея одна и та же.

Поведение `GenServer` требует реализации семи функций обратного вызова, но чаще всего вы будете пользоваться только некоторыми из них. Стандартные реализации всех необходимых функций можно добавить в проект, указав модуль `Genserver` после макроса *use*:

```
iex(1)> defmodule KeyValueStore do
          use GenServer
        end
```

Когда компилятору встречается данная команда, он вызывает специальный макрос из модуля GenServer, внедряющий некоторое количество функций в вызывающий модуль (в данном случае KeyValueStore). Можно убедиться в этом в оболочке:

```
iex(2)> KeyValueStore.__info__(:functions)
[child_spec: 1, code_change: 3, handle_call: 3, handle_cast: 2, handle_info: 2, init: 1,
terminate: 2]
```

В данном случае для проверки используется функция info/1, реализация которой автоматически добавляется в каждый модуль Elixir во время компиляции. Она выводит список всех экспортированных функций модуля (кроме себя самой).

Как видите, с помощью выражения use GenServer в модуль было добавлено несколько функций обратного вызова, необходимых для использования поведения GenServer.

Разумеется, реализацию каждой из этих функций можно впоследствии переопределить по своему усмотрению, создав в своем модуле функции с такими же именами и арностью.

Итак, теперь для подключения вашего модуля обратного вызова к поведению все готово. Запустите процесс при помощи функции GenServer.start/2:

```
iex(3)> GenServer.start(KeyValueStore, nil)
{:ok, #PID<0.51.0>}
```

Сработало. Так же, как и при использовании абстракции ServerProcess, запускается серверный процесс, и поведение использует KeyValueStore в качестве модуля обратного вызова. Второй аргумент функции GenServer.start/2 – это настраиваемый параметр, передаваемый процессу во время его инициализации. На данном этапе он вам не нужен, поэтому функции передается значение nill. Также обратите внимание, что GenServer.start/2 возвращает кортеж вида {:ok, pid}.

6.2.3. Обработка запросов

Пришло время подготовить модуль KeyValueStore для работы с GenServer. Для этого вам необходимо реализовать три функции обратного вызова – init/1, handle_cast/2 и handle_call/3, как показано в листинге 6.9. Эти функции работают практически так же, как и одноименные функции модуля ServerProcess, за исключением пары моментов, перечисленных ниже.

1. Функция init/1 принимает на вход один аргумент. Она же указывается вторым аргументом в функции GenServer.start/2, и с помощью нее вы можете передать какие-либо данные серверу при его запуске.
2. Результат функции init/1 должен быть представлен в виде кортежа {:ok, initial_state}.
3. Функция handle_cast/2 принимает в качестве аргументов запрос и состояние, а возвращает результат в виде {:noreply, new_state}.
4. Функция handle_call/3 принимает на вход запрос, информацию о вызывающей стороне и состояние. Возвращаемый ей результат должен иметь вид: {:reply, response, new_state}.

Листинг 6.9 ❖ Реализация функций обратного вызова GenServer (key_value_gen_server.ex)

```
defmodule KeyValueStore do
  use GenServer

  def init(_) do
    {:ok, %{}}
  end

  def handle_cast({:put, key, value}, state) do
    {:noreply, Map.put(state, key, value)}
  end

  def handle_call({:get, key}, _, state) do
    {:reply, Map.get(state, key), state}
  end
end
```

Второй аргумент функции `handle_call/3` – это кортеж, содержащий ID запроса (используемый внутри поведения `GenServer`) и pid вызывающего процесса. В большинстве случаев эта информация вам не понадобится, поэтому в данном примере она игнорируется.

Теперь осталось только одно – реализовать функции интерфейса. Для взаимодействия с процессом `GenServer` вы можете использовать функции модуля `GenServer`, в частности `GenServer.start/2` для запуска процесса и `GenServer.cast/2` и `GenServer.call/2` для отправки запросов. Реализация приведена в следующем листинге.

Листинг 6.10 ❖ Функции интерфейса (key_value_gen_server.ex)

```
defmodule KeyValueStore do
  use GenServer

  def start do
    GenServer.start(KeyValueStore, nil)
  end

  def put(pid, key, value) do
    GenServer.cast(pid, {:put, key, value})
  end

  def get(pid, key) do
    GenServer.call(pid, {:get, key})
  end

  ...
end
```

Вот и все! Всего пару минут – и вместо примитивного сервера `ServerProcess` у вас появился полноценный `GenServer`. Протестируем его:

```
iex(1)> {:ok, pid} = KeyValueStore.start()

iex(2)> KeyValueStore.put(pid, :some_key, :some_value)

iex(3)> KeyValueStore.get(pid, :some_key)
:some_value
```

Все работает как надо.

Между ServerProcess и GenServer достаточно много отличий, но некоторые из них хочется выделить особым образом.

Во-первых, функция GenServer.start/2 работает синхронно. Другими словами, она возвращает результат только после того, как в серверном процессе завершится обратный вызов init/1. Соответственно, запустивший сервер клиентский процесс приостанавливается, и возобновляется он только после инициализации серверного процесса.

Во-вторых, функция GenServer.call/2 не зависает на ожидании ответа. По умолчанию, если ответ не приходит в течение пяти секунд, в клиентском процессе возникает ошибка. Вы можете изменить этот промежуток, используя GenServer.call(pid, request, timeout), где время указывается в миллисекундах. Кроме того, если ожидающий ответа процесс неожиданно завершается, GenServer обнаруживает это и возбуждает соответствующую ошибку в вызывающем процессе.

6.2.4. Обработка простых сообщений

Сообщения, оправляемые серверному процессу с помощью функций GenServer.call и GenServer.cast, содержат не только данные запроса, но и дополнительную информацию. Напомню, что происходило в примере из раздела 6.1:

```
defmodule ServerProcess do
  ...

  def call(server_pid, request) do
    send(server_pid, {:call, request, self()})   <-- Отправка сообщения с запросом типа call
    ...
  end

  def cast(server_pid, request) do
    send(server_pid, {:cast, request})   <-- Отправка сообщения с запросом типа cast
  end

  ...

  defp loop(callback_module, current_state) do
    receive do
      {:call, request, caller} ->   <-- Особая обработка сообщения call-запроса
        ...

      {:cast, request} ->   <-- Особая обработка сообщения cast-запроса
        ...
    end
  end
  ...
end
```

Обратите внимание, что в этих вызовах не просто посылается запрос, но и указываются такие дополнительные данные, как тип запроса, а для запросов типа call – еще и вызывающий процесс.

В GenServer реализован похожий механизм – для декорации сообщений используются атомы :"$gen_cast" и :"$gen_call". Формат этих сообщений значения

не имеет, но важно понимать, что внутри GenServer используется определенный формат сообщений и они обрабатываются особым образом.

Иногда существует необходимость обработать не характерные для GenServer сообщения. Представим, что нужно периодически подчищать состояние серверного процесса. Вы можете использовать функцию Erlang :timer.send_interval/2, которая будет время от времени посылать сообщения вызывающему процессу. Так как это сообщение не типично для GenServer, оно не распознается как cast или call. Для подобных сообщений GenServer будет вызывать функцию обратного вызова handle_info/2 и предоставлять вам возможность выполнить с этим сообщением какое-либо действие.

В коде это выглядит следующим образом:

```
iex(1)> defmodule KeyValueStore do
          use GenServer

          def init(_) do
            :timer.send_interval(5000, :cleanup)  <-- Периодическая отправка сообщений
            {:ok, %{}}
          end

          def handle_info(:cleanup, state) do  <-- Обработка сообщения :cleanup
            IO.puts "performing cleanup..."
            {:noreply, state}
          end
        end

iex(2)> GenServer.start(KeyValueStore, nil)
performing cleanup...   |
performing cleanup...   | Вывод на экран каждые 5 секунд
performing cleanup...   |
```

Во время инициализации процесса вы даете команду посылать сообщение :cleanup каждые пять секунд. Это сообщение затем обрабатывается в функции обратного вызова handle_info/2, работающей точно так же, как handle_cast/2, и возвращающей результат в виде кортежа {:noreply, new_state}.

6.2.5. Прочие особенности GenServer

Существует множество других особенностей и тонкостей GenServer, которые я оставил без внимания. О каких-то из них вы узнаете в последующих разделах книги, но вам определенно стоит уделить немного времени и пробежаться по документации модуля GenServer (**https://hexdocs.pm/elixir/GenServer.html**) и лежащего в его основе модуля Erlang (**http://erlang.org/doc/man/gen_server.html**).

И все же я хотел бы остановиться на некоторых особенностях подробнее.

Проверка на этапе компиляции

Одна из проблем механизма обратных вызовов заключается в том, что можно незаметно совершить ошибку при определении функции. Рассмотрим такой пример:

```
iex(1)> defmodule EchoServer do
          use GenServer
```

```
  def handle_call(some_request, server_state) do
    {:reply, some_request, server_state}
  end
end
```

Итак, у нас есть простой эхо-сервер, обрабатывающий запросы типа call и отправляющий клиенту ответ. Протестируем его:

```
iex(2)> {:ok, pid} = GenServer.start(EchoServer, nil)
{:ok, #PID<0.96.0>}

iex(3)> GenServer.call(pid, :some_call)
** (exit) exited in: GenServer.call(#PID<0.96.0>, :some_call, 5000)
    ** (EXIT) an exception was raised:
        ** (RuntimeError) attempted to call GenServer #PID<0.96.0> but
                          no handle_call/3 clause was provided
```

Получив запрос, серверный процесс аварийно завершился из-за того, что условие `handle_call/3` отсутствует, хотя оно было реализовано в модуле. Почему так произошло? Если приглядеться к определению `EchoServer`, можно увидеть, что арность функции `handle_call` равна двум, а `GenServer` требует наличия функции с арностью, равной трем.

Можно решить данную проблему, указав компилятору, что определенная функция должна соответствовать контракту того или иного поведения. Для этого необходимо прописать атрибут модуля *@impl* перед непосредственной реализацией функции обратного вызова:

```
iex(1)> defmodule EchoServer do
          use GenServer

          @impl GenServer
          def handle_call(some_request, server_state) do
            {:reply, some_request, server_state}
          end
        end
```

Сигнал компилятору, что далее следует определение функции обратного вызова

Запись `@impl GenServer` дает компилятору понять, что следующая за ней функция будет определена как функция обратного вызова для поведения `GenServer`. Как только эта строка будет скомпилирована в оболочке, вы увидите предупреждение:

```
warning: got "@impl GenServer" for function handle_call/2 but this behaviour does not specify
such callback.
```

Компилятор сообщает вам, что `GenServer` не распознает `handle_call/2`, и уже во время компиляции вы узнаете об ошибке. Хорошим тоном считается указывать атрибут `@impl` для каждой функции обратного вызова.

Регистрация имени

Как вам уже известно, процесс можно зарегистрировать под локальным именем (атомом), притом понятие «локальный» означает, что имя регистрируется только в текущем экземпляре BEAM. Это позволяет создавать отдельные процессы, к которым можно обращаться по имени, не зная их pid.

Локальная регистрация – очень важный механизм, поскольку он используется в шаблонах проектирования отказоустойчивости и распределенных систем. Более подробно он рассматривается в последующих главах, но стоит отметить, что вы можете указать имя процесса при вызове функции `GenServer.start`:

```
GenServer.start(
  CallbackModule,
  init_param,
  name: :some_name    <-- Регистрация процесса под указанным именем
)
```

Теперь запросы к серверу можно отправлять с использованием имени процесса:

```
GenServer.call(:some_name, ...)
GenServer.cast(:some_name, ...)
```

Чаще всего в качестве имени процесса используется имя модуля. Как уже объяснялось в разделе 2.4.2, имена модулей – это атомы, поэтому их можно без опасения передавать в опции `:name`. Вот как это выглядит:

```
defmodule KeyValueStore do
  def start() do
    GenServer.start(KeyValueStore, nil, name: KeyValueStore)  <-- Регистрация серверного процесса
  end

  def put(key, value) do
    GenServer.cast(KeyValueStore, {:put, key, value})  <-- Отправка запроса зарегистрированному процессу
  end

  ...
end
```

Теперь функции `KeyValueStore.put` не нужно передавать pid, она отправит запрос зарегистрированному процессу.

Также имя `KeyValueStore` в аргументах функций можно заменить на специальную запись `__MODULE__`. Во время компиляции на месте этой записи появится имя модуля, в котором находится код:

```
defmodule KeyValueStore do
  def start() do
    GenServer.start(__MODULE__, nil, name: __MODULE__)  <-- Регистрация серверного процесса
  end

  def put(key, value) do
    GenServer.cast(__MODULE__, {:put, key, value})  <-- Отправка запроса зарегистрированному процессу
  end

  ...
end
```

После компиляции данный код будет выглядеть точно так же, как предыдущий, но последующая работа с ним упростится. К примеру, если вы решите переименовать `KeyValueStore` в `KeyValue.Store`, это достаточно будет сделать только в одном месте.

Остановка сервера

Разные функции обратного вызова могут возвращать ответы различного вида. В предыдущих примерах вы могли видеть наиболее типовые из них:

- {:ok, initial_state} от функции init/1;
- {:reply, response, new_state} от функции handle_call/3;
- {:noreply, new_state} от функций handle_cast/2 и handle_info/2.

Существуют и другие виды взаимодействия с сервером, и наиболее важное из них – его остановка.

В функции init/1 вы можете отменить запуск сервера. В этом случае необходимо возвратить {:stop, reason} или :ignore, и вместо запуска бесконечного цикла серверный процесс завершится.

Если init/1 возвращает {:stop, reason}, то результатом функции start/2 будет кортеж {:error, reason}, а если init/1 возвращает :ignore, то и start/2 возвратит :ignore. Разница между этими двумя результатами лежит в их назначении. Если сервер не может продолжать работу из-за возникшей ошибки, то следует использовать {:stop, reason}, а в противном случае необходимо возвращать :ignore.

Серверный процесс можно остановить, возвратив {:stop, reason, new_state} функциями обратного вызова handle_*. Если останов происходит в обычном порядке, то в качестве причины указывается атом :normal. Если необходимо перед завершением процесса отправить сообщение вызывающей стороне во время выполнения функции handle_call/3, используйте {:stop, reason, response, new_state}.

У вас может возникнуть вопрос, для чего необходимо возвращать обновленное состояние при остановке сервера. Дело в том, что перед тем как завершить серверный процесс, GenServer вызывает функцию обратного вызова terminate/2, передавая ей причину останова и конечное состояние процесса. Это может понадобиться во время очистки состояния.

И еще один способ остановки сервера – вызов функции GenServer.stop/3 в клиентском процессе. Функция посылает синхронный запрос серверу, после чего поведение обрабатывает этот запрос и завершает серверный процесс.

6.2.6. Жизненный цикл процесса

Важно понимать, как работают процессы на основе GenServer и где (в каких процессах) выполняются те или иные функции. Обратимся к рис. 6.1 и вкратце сформулируем жизненный цикл типового серверного процесса.

Клиентский процесс запускает сервер, вызывая функцию GenServer.start и предоставляя модуль обратного вызова (1). Так создается новый серверный процесс на основе поведения GenServer.

Клиент может отправлять запросы серверу с помощью стандартной функции send или различных функций GenServer. При получении сообщения GenServer вызывает функции обратного вызова для его обработки, соответственно, они всегда выполняются в серверном процессе.

Состояние процесса сохраняется в цикле GenServer, но определяется и изменяется оно функциями обратного вызова. Функция init/1 определяет начальное состояние, которое затем передается функциям handle_* (2). Каждая из этих функций получает на вход текущее состояние и возвращает его обновленную версию, которая затем используется в цикле GenServer вместо прежнего состояния.

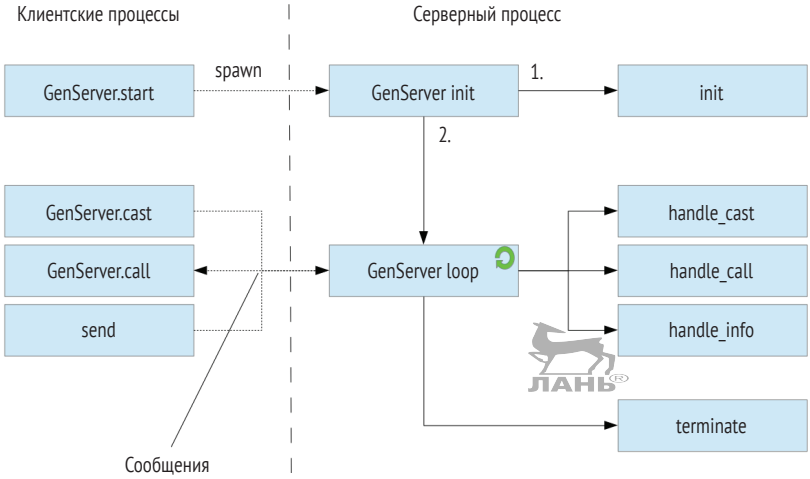


Рис. 6.1 ❖ Жизненный цикл процесса на основе GenServer

Модель акторов

Erlang непреднамеренно оказался реализацией модели акторов, впервые описанной Карлом Хьюиттом (Carl Hewitt). Актор – это конкурентная вычислительная сущность, инкапсулирующая состояние и взаимодействующая с другими такими же сущностями. Обрабатывая сообщение, актор может определять новое состояние, которое будет использовано при обработке следующего сообщения. Это чем-то похоже на работу процессов на основе `GenServer` в Erlang. Однако, как неоднократно заявлял Роберт Вирдинг (Robert Virding, один из создателей Erlang), разработчики пришли к этой идее сами, а о существовании модели акторов узнали гораздо позже.

Существуют некоторые разногласия в том, является ли Erlang должной реализацией модели акторов, поэтому в сообществе Erlang эта терминология не употребляется. В данной книге она также не будет использована, но следует иметь в виду, что для Erlang актором будет являться серверный процесс, чаще всего `GenServer`.

6.2.7. Совместимые с OTP процессы

В силу различных причин при разработке промышленных систем не стоит использовать стандартные процессы, порожденные функцией `spawn`. Все ваши процессы должны быть *совместимы* с OTP. Такие процессы следуют соглашениям OTP и могут быть использованы в деревьях супервизоров (описанных в главе 9), а возникающие в них ошибки заносятся в журнал в более подробном виде.

Процессы, основанные на поведениях OTP, таких как `GenServer` и `Supervisor`, совместимы с OTP. Elixir предоставляет и другие модули, с помощью которых можно создать такие процессы. Например, модуль `Task` (**https://hexdocs.pm/elixir/Task.html**) отлично подойдет для запуска быстрых процессов, решающих одиночные задачи по обработке передаваемых им данных. Модуль Agent (**https://hexdocs.pm/elixir/Agent.html**) – более простая (но менее эффективная) альтернатива `GenServer`,

используемая для создания процессов, единственной задачей которых является управление состоянием. Оба этих модуля будут рассмотрены в главе 10.

Кроме этого, существуют различные совместимые с OTP абстракции, реализованные в сторонних библиотеках. Например, *GenStage* (**https://github.com/elixir-lang/gen_stage**) используется для обратного давления и управления нагрузкой. Модуль *Phoenix.Channel*, входящий в состав фреймворка Phoenix (**http://phoenix-framework.org/**), – для поддержки двусторонней связи между клиентом и сервером поверх WebSocket, HTTP и других протоколов.

В данной книге просто не хватит места на описание всех существующих абстракций, совместимых с OTP, и вам все же придется изучить их самостоятельно. Стоит также отметить, что большинство таких абстракций придерживается принципов `GenServer`. Все упомянутые выше абстракции OTP, за исключением модуля `Task`, построены на основе `GenServer`, а потому, на мой взгляд, `GenServer` является самой важной частью OTP. Если вы как следует разберетесь с ним, то все остальные абстракции будет гораздо проще освоить.

6.2.8. Упражнение: создание сервера для списка дел на основе GenServer

Закрепим пройденный в данной главе материал с помощью простого, но очень интересного упражнения. Вам предстоит переписать сервер для списка дел, реализованный ранее в этой главе, подключив его к поведению `GenServer`. Это не должно составить труда, но вы всегда можете обратиться к готовому решению, находящемуся в файле todo_server.ex.

Обязательно выполните это упражнение или хотя бы проанализируйте готовое решение и разберитесь в нем, потому что в последующих главах вы будете только усложнять и расширять функционал этого простого серверного процесса, чтобы в итоге получить распределенную систему с высокой степенью конкурентности.

Выводы

- Обобщенный серверный процесс – это абстракция, реализующая общие для всех серверных процессов задачи, такие как организация рекурсивного цикла и обмен сообщениями.
- Обобщенный серверный процесс может быть реализован в виде поведения. Поведение поддерживает работоспособность процесса, а конкретная реализация может подключиться к нему с помощью модулей обратного вызова.
- Поведение обращается к функциям обратного вызова, когда конкретной реализации требуется принять какое-либо решение.
- `GenServer` – это поведение, реализующее обобщенный серверный процесс.
- Модуль обратного вызова для работы с `GenServer` должен содержать реализации определенных функций. Наиболее часто используемые из них: `init/1`, `handle_cast/2`, `handle_call/3` и `handle_info/2`.

- Взаимодействовать с процессом `GenServer` можно через модуль `GenServer`.
- Серверному процессу можно отправить два вида запросов – call и cast.
- Запрос типа cast срабатывает по принципу «запустили и забыли»: вызывающий процесс посылает сообщение серверу и сразу же приступает к выполнению другой задачи.
- Запрос типа call – это синхронный запрос, ожидающий ответа: вызывающий процесс посылает сообщение серверу и либо ждет ответа указанное время и продолжает работу по его истечении, либо происходит аварийная остановка сервера.

Глава 7

Создание конкурентной системы

В главе рассматривается:

- работа с проектом mix;
- управление несколькими списками дел;
- сохранение данных;
- логика работы процессов.

Примеры конкурентных вычислений, которые вы изучили в предыдущих главах, были основаны лишь на одном экземпляре серверного процесса. Типовые системы на Elixir/Erlang обычно выстраиваются из огромного множества процессов, многие из которых являются серверными процессами с сохранением состояния. Для сложной системы средних масштабов характерно наличие нескольких тысяч процессов, а для систем побольше – сотни тысяч или даже миллионов. Не забывайте, что процессы легковесны, и создавать их можно сколь угодно много. А благодаря конкурентному обмену сообщениями устройство таких систем приобретает более понятный вид. Выполнение нескольких задач в отдельных процессах крайне эффективно и положительным образом влияет на масштабируемость и надежность ваших систем.

В этой главе будет рассмотрено создание более сложной системы на основе большого количества взаимодействующих между собой процессов. Конечная практическая цель этой книги – распределенный HTTP-сервер, управляющий множеством пользователей, одновременно производящих какие-либо действия со списками дел. В данной главе вы займетесь разработкой инфраструктуры для выполнения операций над списками и сохранения их на диске.

Для начала рассмотрим, как организуется управление более сложными проектами с помощью инструмента `mix`.

7.1. Работа с проектом mix

Код нашего будущего сервера становится все сложнее, и хранить каждый новый модуль в отдельном файле уже слишком проблематично. Самое время научиться работать с многофайловыми проектами.

Во второй главе упоминалось об инструменте *mix*, используемом для создания, сборки и запуска проектов, а также управления зависимостями, запуска тестов и создания специфических для проекта задач. В этой главе вы узнаете всю необходимую информацию для создания и запуска проекта mix, а дополнительные особенности инструмента будут представлены в последующих главах по мере необходимости.

Создадим проект для системы управления списком дел. Для этого выполните в оболочке следующую команду:

```
$ mix new todo
```

После этого создастся папка todo со структурой проекта. В ней будет находиться несколько файлов, включая readme, файлы для поддержки юнит-тестов и файл .gitignore. Проекты mix очень простые, в них автоматически генерируется небольшое количество файлов.

> **СОВЕТ** В данной книге инструмент mix рассматривается на поверхностном уровне, и внимание уделяется только некоторым его функциональным особенностям, необходимым для решения поставленных задач. Чтобы узнать о нем больше, советую обратиться к руководству «Введение в mix» (**https://elixir-lang.org/getting-started/mix-otp/introduction-to-mix.html**). Вы также можете выполнить в командной строке команды *mix help*, чтобы увидеть список всех возможных команд, и *mix help command* для получения информации по конкретной команде. И наконец, доступна онлайн-документация по адресу: **https://hexdocs.pm/mix**.

Как только проект будет создан, вы можете открыть папку с ним и запускать из нее задачи. Например, можно скомпилировать проект с помощью команды *mix compile* или запустить тест, выполнив *mix test*.

Также существует альтернативный метод запуска оболочки IEx, при котором особенно удобно взаимодействовать с проектами mix, – команда *iex -S mix*. Она выполняет два действия: компиляцию проекта (как и в случае с mix compile) и запуск оболочки, после чего все модули проекта становятся доступными. Это означает, что все сгенерированные файлы BEAM (бинарные файлы, представляющие скомпилированные модули) находятся по адресам загрузки.

Используя mix, вы можете размещать свой код по нескольким файлам и папкам. Если поместить файлы с расширением .ex в папку lib, они будут автоматически включены в следующую сборку. Также вы можете создавать различные подпапки в папке lib, а один файл с расширением .ex может содержать множество модулей.

Строгих правил именования и организации файлов в папках не существует, но стоит придерживаться некоторых общепринятых соглашений.

1. Размещайте модули под общим псевдонимом модуля более высокого уровня. Например, давайте модулям имена Todo.List, Todo.Server и т. д. Таким образом вам удастся избежать конфликтов имен при объединении нескольких проектов в одну систему.
2. В общем случае каждый отдельный файл должен содержать только один модуль. Небольшой вспомогательный модуль для внутреннего использования можно помещать в один и тот же файл с использующим его модулем. Протоколы для этого модуля также могут быть реализованы в этом же файле.

3. Имя файла должно совпадать с именем реализованного в нем модуля и должно быть прописано в змеином регистре. Например, модуль `TodoServer` должен находиться в файле todo_server.ex папки lib.
4. Структура папок должна соответствовать составным именам модулей. Путь к модулю `Todo.Server` должен быть таким: lib/todo/server.ex.

Следовать этим правилам не обязательно, но они соблюдались разработчиками самого Elixir и многих сторонних библиотек.

Что ж, теперь самое время заняться проектом. У вас уже есть два готовых модуля – `TodoList` и `TodoServer`. Конечные версии обоих модулей можно найти в файле todo_server.ex из главы 6. Согласно вышеописанным правилам, имена модулей следует заменить на `Todo.List` и `Todo.Server` и добавить их в проект `todo`. Вот что нужно будет сделать:

1) удалить файл todo/lib/todo.ex;
2) переименовать файл в todo/test/todo_test.exs;
3) скопировать код модуля `TodoList` и вставить его в файл todo/lib/todo/list.ex, переименовать модуль в `Todo.List`;
4) скопировать код модуля `TodoServer` и вставить его в файл todo/lib/todo/server.ex file, переименовать модуль в `Todo.Server`;
5) заменить имена `TodoServer` и `TodoList` на `Todo.Server` и `Todo.List` в каждом их упоминании.

Запустите систему с помощью команды `iex -S mix` и убедитесь, что она работает:

```
$ iex -S mix

iex(1)> {:ok, todo_server} = Todo.Server.start()

iex(2)> Todo.Server.add_entry(todo_server,
          %{date: ~D[2018-12-19], title: "Dentist"})

iex(3)> Todo.Server.entries(todo_server, ~D[2018-12-19])
[%{date: ~D[2018-12-19], id: 1, title: "Dentist"}]
```

Теперь у вас есть mix-проект для системы списка дел, функционал которого вам предстоит расширить с изучением следующих разделов книги.

7.2. Управление несколькими списками дел

Прежде чем перейти к обсуждению управления несколькими списками дел, давайте еще раз вспомним, что у вас имеется на данный момент:

- чистая функциональная абстракция `Todo.List`;
- серверный процесс, способный поддерживать один список дел долгое время.

Существует два способа научить имеющийся код работать с несколькими списками:

- реализовать чистую функциональную абстракцию `TodoListCollection` и переписать код так, чтобы `Todo.Server` использовал ее в качестве внутреннего состояния;
- запустить для каждого списка дел свой экземпляр серверного процесса.

Проблема первого способа в том, что в итоге запросы всех пользователей будет обрабатывать один-единственный процесс. Это решение сложно назвать масштабируемым. Если с системой будет взаимодействовать большое количество пользователей, их запросы будут блокировать друг друга в борьбе за общий ресурс – серверный процесс, выполняющий все возможные задачи.

Альтернативный способ – использовать столько процессов, сколькими списками вы планируете управлять. В этом случае запросы к каждому серверу будут обрабатываться конкурентно, а система будет более отзывчивой и масштабируемой.

Чтобы запустить несколько серверных процессов, вам потребуется реализовать дополнительную сущность, которая будет создавать экземпляры абстракции `Todo.Server` или поручать работу уже существующим. Эта сущность должна хранить состояние, представленное структурой ключ/значение, устанавливающей соответствие между именами списков и идентификаторами серверных процессов. Данное состояние должно быть изменяемым (количество списков может со временем изменяться), а также доступным в течение всего жизненного цикла сервера.

Итак, необходимо добавить в систему еще один процесс – кеш списка дел. Вы запустите только один экземпляр этого процесса и будете использовать его для создания и получения pid серверного процесса, соответствующего заданному имени. В модуль потребуется экспортировать лишь две функции – `start/0`, запускающую процесс, и `server_process/2`, возвращающую pid серверного процесса по заданному имени и запускающую процесс, если он еще не запущен.

7.2.1. Создание кеш-процесса

Начнем работать над реализацией кеш-процесса. Первым делом скопируйте все содержимое папки todo в папку todo_cache. Затем добавьте в нее новый файл todo_cache/lib/todo/cache.ex, в котором будет храниться код модуля `Todo.Cache`.

Теперь необходимо решить, в каком виде лучше всего представить состояние процесса. Ситуация такова, что процессу передается имя списка, а он должен возвратить соответствующий идентификатор процесса. В таком случае будет разумно использовать словарь, устанавливающий соответствие между именами списков дел и идентификаторами их серверных процессов. Реализация этого показана в следующем листинге.

Листинг 7.1 ❖ Инициализация кеша (todo_cache/lib/todo/cache.ex)

```
defmodule Todo.Cache do
  use GenServer

  def init(_) do
    {:ok, %{}}
  end

  ...
end
```

Следующим действием необходимо добавить реализацию запроса `server_process` и решить, какого он будет типа. Поскольку данный запрос должен возвращать вызывающему процессу ответ (pid серверного процесса), то решать тут нечего – запрос должен быть типа call. Его реализация показана в следующем листинге.

Листинг 7.2 ❖ Обработка запроса server_process (todo_cache/lib/todo/cache.ex)

```
defmodule Todo.Cache do
  ...

  def handle_call({:server_process, todo_list_name}, _, todo_servers) do
    case Map.fetch(todo_servers, todo_list_name) do
      {:ok, todo_server} ->    <-- Сервер присутствует в словаре
        {:reply, todo_server, todo_servers}

      :error ->      <-- Сервера нет в словаре
        {:ok, new_server} = Todo.Server.start()    <-- Запуск нового сервера

        {
          :reply,
          new_server,
          Map.put(todo_servers, todo_list_name, new_server)
        }
    end
  end
  ...
end
```

В данном примере для отправки запроса к словарю используется функция `Map.fetch/2`. Если заданному ключу соответствует какое-либо значение, оно возвращается вызывающему процессу, а состояние остается в прежнем виде. В противном случае запускается новый сервер, возвращается его pid и данная пара ключ/значение помещается в словарь.

И наконец, не стоит забывать о функциях интерфейса.

Листинг 7.3 ❖ Функции интерфейса (todo_cache/lib/todo/cache.ex)

```
defmodule Todo.Cache do
  ...

  def start do
    GenServer.start(__MODULE__, nil)
  end

  def server_process(cache_pid, todo_list_name) do
    GenServer.call(cache_pid, {:server_process, todo_list_name})
  end

  ...
end
```

Обратите внимание, что функции `GenServer.start/2` в качестве первого аргумента передается `__MODULE__`. Во время компиляции это выражение заменяется на имя текущего модуля. Это очень удобно, так как позволяет избавиться от дублирования кода и избежать ошибок при смене имени модуля.

Теперь кеш-процесс полностью готов и можно его протестировать. Запустите оболочку командой `iex -S mix` и проделайте следующие действия:

```
iex(1)> {:ok, cache} = Todo.Cache.start()

iex(2)> Todo.Cache.server_process(cache, "Bob's list")
#PID<0.69.0>
```

Создание нового процесса при первом запросе

```
iex(3)> Todo.Cache.server_process(cache, "Bob's list")
#PID<0.69.0>
```

Возвращение pid того же процесса при втором запросе

```
iex(4)> Todo.Cache.server_process(cache, "Alice's list")
#PID<0.72.0>
```

Запрос с другим именем возвращает другой pid

Возвращаемое значение идентификатора определяет серверный процесс, ответственный за конкретный список дел. Для управления этим списком вы можете пользоваться уже знакомым вам способом:

```
iex(5)> bobs_list = Todo.Cache.server_process(cache, "Bob's list")

iex(6)> Todo.Server.add_entry(bobs_list,
          %{date: ~D[2018-12-19], title: "Dentist"})

iex(7)> Todo.Server.entries(bobs_list, ~D[2018-12-19])
[%{date: ~D[2018-12-19], id: 1, title: "Dentist"}]
```

Само собой, список Alice при выполнении этих манипуляций остается в неизменном виде:

```
iex(8)> Todo.Cache.server_process(cache, "Alice's list") |>
          Todo.Server.entries(~D[2018-12-19])
[]
```

Имея такой кеш-процесс, вы можете управлять несколькими списками дел независимо друг от друга. В следующем примере показано создание 100 000 серверов и проверка количества запущенных проектов:

```
iex(1)> {:ok, cache} = Todo.Cache.start()

iex(2)> :erlang.system_info(:process_count)
54

iex(3)> Enum.each(
          1..100_000,
          fn index ->
            Todo.Cache.server_process(cache, "to-do list #{index}")
          end
        )

iex(4)> :erlang.system_info(:process_count)
100054
```

Для получения числа запущенных процессов в данном примере используется функция *:erlang.system_info/1*.

Как вы могли заметить, запустив лишь один процесс (кеш), следующим действием вы получили информацию о 54 запущенных процессах. Не стоит беспокоиться – все это процессы, используемые Elixir и Erlang на внутреннем уровне.

7.2.2. Создание тестов

Собрав весь свой код в один проект `mix`, вы можете начать писать автоматизированные тесты. В Elixir для этих целей имеется встроенный тестовый фреймворк *ex_unit*. Все очень просто: напишите тест и запустите его с помощью команды `mix test`.

Рассмотрим небольшой пример тестирования поведения запроса `Todo.Cache.server_process/2`. В листинге ниже приводится каркас файла теста.

Листинг 7.4 ❖ Каркас файла теста (todo_cache/test/todo_cache_test.exs)

```
defmodule TodoCacheTest do
  use ExUnit.Case ◄──── Подготовка модуля к тестированию

  ...
end
```

Имейте в виду, чтобы тест выполнялся, файл теста должен находиться в папке теста, а его имя должно оканчиваться на `_test.exs`. Как уже говорилось во второй главе, расширение .exs используется для скриптов Elixir и означает, что файл не скомпилирован на диске. Инструмент `mix` будет интерпретировать этот файл каждый раз при выполнении тестов.

В файле скрипта должен быть определен модуль, содержащий код тестов. Выражение `use ExUnit.Case` используется для подготовки модуля к тестированию, а именно добавляет в него некоторое количество шаблонного кода для его совместимости с `ex_unit` и импортирует необходимые вспомогательные макросы.

Среди этих макросов можно выделить макрос `test`, который можно использовать для определения тестов, как показано в следующем листинге на примере `Todo.Cache.server_process/2`.

Листинг 7.5 ❖ Тестирование запроса `server_process` (todo_cache/test/todo_cache_test.exs)

```
defmodule TodoCacheTest do
  use ExUnit.Case

  test "server_process" do ◄──── Определение теста
    {:ok, cache} = Todo.Cache.start()
    bob_pid = Todo.Cache.server_process(cache, "bob")

    assert bob_pid != Todo.Cache.server_process(cache, "alice")
    assert bob_pid == Todo.Cache.server_process(cache, "bob")
  end                                                          Утверждения теста

  ...
end
```

Чтобы определить тест, необходимо прописать `test test_description do … end`. Описание теста – это строка, которая вставляется в вывод, если тест не пройден. Код самого теста помещается в блок `do`.

Макрос `test` служит примером возможностей метапрограммирования в Elixir. Данный макрос генерирует функцию, содержащую шаблонный код и код блока `do`. Эта функция будет вызвана модулем `ex_unit` во время выполнения тестов.

Конкретно в данном тесте сначала запускается кеш-процесс и выбирается определенный серверный процесс. Затем ожидаемое поведение тестируется с помощью макроса `assert`, принимающего выражение и проверяющего его результат. Если проверка не пройдена, `assert` возбуждает ошибку и дает описательный вывод. Эта ошибка затем ловится `ex_unit` и выводится на экран.

Например, взгляните на первое утверждение:

```
assert bob_pid != Todo.Cache.server_process(cache, "alice")
```

В нем проверяется, что списки Alice и Bob управляются разными процессами.

Так же, как и `test`, `assert` – это макрос, и вызывается он во время компиляции. Макрос интроспектирует следующее за ним выражение и преобразует его в другой код, который выглядит примерно так:

```
left_value = bob_pid
right_value = Todo.Cache.server_process(cache, "alice")
comparison_result = left_value != right_value
if comparison_result == false do
  # raise an error
end
```

Иными словами, макрос `assert` генерирует код, который не сможет успешно выполниться, если выражение `bob_pid != Todo.Cache.server_process(cache, "alice")` вернет `false`.

Большим преимуществом assert является тот факт, что вам не понадобится запоминать абсолютно новый набор функций вроде `assert_equal`, `assert_not_equal` или `assert_gt` или писать свои утверждения. Вместо этого для тестирования поведения вы можете использовать все те же выражения, что и обычно, включающие стандартные операторы сравнения ==, !=, >, < и т. д.

Можно даже создать утверждение с проверкой успешности сопоставления с образцом. Давайте рассмотрим небольшой пример. Добавим еще один тест для проверки поведения операций сервера. Для простоты пропишем этот тест в файле с предыдущими. Код примера приведен в листинге 7.6.

Листинг 7.6 ❖ Тестирование операций сервера (todo_cache/test/todo_cache_test.exs)

```
defmodule TodoCacheTest do
  use ExUnit.Case

  ...

  test "to-do operations" do
    {:ok, cache} = Todo.Cache.start()
    alice = Todo.Cache.server_process(cache, "alice")
    Todo.Server.add_entry(alice, %{date: ~D[2018-12-19], title: "Dentist"})
    entries = Todo.Server.entries(alice, ~D[2018-12-19])

    assert [%{date: ~D[2018-12-19], title: "Dentist"}] = entries
  end
end
```

Утверждение с сопоставлением

В данном примере создается один серверный процесс, в список добавляется одна запись, и производится выборка записей по заданной дате. С помощью сопоставления с образцом вы утверждаете, что список содержит только один элемент с указанными датой и заголовком. Сопоставление с образцом позволяет проверить только соответствующие поля и размер результата в одном выражении.

На данный момент у вас имеется один файл с несколькими тестами. Проект test в папке todo_cache также содержит еще один файл с именем todo_list_test.

exs, проверяющий поведение модуля `Todo.List`. В данной книге код этого файла не представлен из соображений краткости.

> **ПРИМЕЧАНИЕ** Следует отметить, что примеры в данной книге не были разработаны через тестирование и не были протестированы надлежащим образом. Их главная цель – простота и иллюстрация решения поставленной задачи. Как правило, такой код не пригоден к тестированию, и в нем во избежание ошибок присутствует доля импровизации.

Запустите все тесты с помощью команды `mix test`:

```
$ mix test
.......

Finished in 0.05 seconds
7 tests, 0 failures
```

Фреймворк `ex_unit` предоставляет множество других возможностей, но здесь они рассмотрены не будут. Чтобы узнать больше о модульном тестировании в Elixir, вы можете обратиться к официальным документациям `ExUnit` (**https://hexdocs.pm/ex_unit**) и `mix test` (**https://hexdocs.pm/mix/Mix.Tasks.Test.html**).

7.2.3. Анализ зависимостей процесса

Поразмыслим немного над имеющейся на данный момент системой. Вы добавили поддержку управления большим количеством экземпляров списка, а в конце концов нужно прийти к использованию этой инфраструктуры на HTTP-сервере. Для HTTP-серверов Elixir/Erlang характерен запуск отдельного процесса для каждого запроса. Получается, при выполнении многими конечными пользователями каких-либо действий одновременно можно ожидать, что множество процессов BEAM будут одновременно обращаться к кеш-процессу и серверным процессам. Взаимодействие этих процессов показано на рис. 7.1.

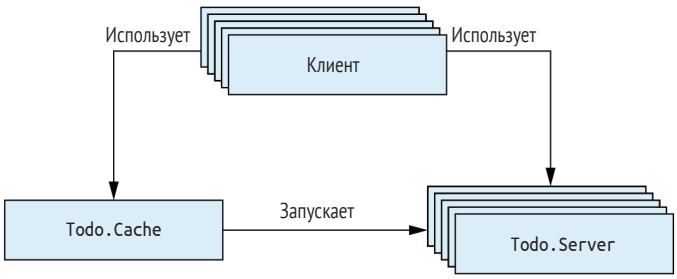


Рис 7.1 ❖ Взаимодействие процессов

Каждый блок на этом рисунке – это отдельный процесс. Блоки с надписью «Клиент» иллюстрируют произвольных клиентов, например процессы – обработчики HTTP-запросов. Взглянув на эту схему, можно сходу заметить некоторые характерные особенности конкурентного поведения вашей системы:

- несколько клиентов (возможно, очень большое их количество) посылают запросы одному и тому же кеш-процессу;

- несколько клиентов взаимодействуют с несколькими серверными процессами.

Первая особенность предполагает возможность возникновения в системе узкого места. Так как кеш-процесс у вас всего один, он может обрабатывать только один запрос `server_process` за раз, независимо от количества ресурсов ЦП.

На практике эта проблема может оказаться не такой уж существенной. Если запрос выполняется, скажем, за одну микросекунду, то процесс может обрабатывать до 1 млн запросов в секунду, что вполне достаточно для решения большинства задач. Но если же запрос выполняется 100 миллисекунд, то обработать удастся лишь 10 запросов в секунду, и ваша система не сможет выдержать большую нагрузку.

Достаточно легко вывести формулу наиболее эффективной работы процесса. Сколько бы ни поступило конкурентных запросов, процесс обрабатывает их по одному. Значит, процесс будет работать эффективно только в том случае, если скорость обработки запросов будет равна частоте их поступления. В противном случае вам придется как-то оптимизировать работу процесса или предпринять иные меры.

В нашем случае кеш-процесс выполняет элементарную операцию – поиск по словарю с возможным последующим запуском нового процесса и обновление словаря. Тестирование на моем компьютере показало, что при наличии 1 млн списков новый сервер создается и помещается в словарь за 10 микросекунд, а поиск по словарю осуществляется за 2 микросекунды. Этого вполне достаточно, чтобы справиться с нагрузкой 100 000 запросов в секунду, и это нормальная производительность для данной задачи. Если вы хотите выполнить данный тест на своем компьютере, следуйте указаниям в файле todo_cache/lib/load_test.ex.

Последовательное выполнение действий в процессах дает дополнительное преимущество. Так как процесс обрабатывает за раз только один запрос, его внутреннее состояние обновляется последовательно. Вы можете быть уверены в том, что состояние не будет одновременно изменяться по нескольку раз, а значит, возможность возникновения состояния гонки исключена.

> **СОВЕТ** Если вам необходимо синхронизировать какой-то фрагмент кода, то есть убедиться в том, что один и тот же код не выполняется одновременно и многократно, то лучше всего выполнять такой код в отдельном процессе. Когда нескольким клиентам нужно запустить данный код, они отправляют запрос этому процессу, который выступает в качестве места синхронизации, выполняя каждый запрос по очереди.

Стоит сказать пару слов о взаимодействии клиента с серверными процессами. Как только клиент получает pid сервера, действия со списком начинают выполняться конкурентно по отношению ко всем остальным действиям в системе. Предполагается, что операции со списком будут довольно сложными, поэтому крайне эффективно выполнять их конкурентно, что позволяет добиться масштабируемости системы, то есть ее способности управлять многочисленными списками, задействовав максимум возможных ресурсов.

Как вам известно, ожидающий сообщения процесс приостанавливается и не расходует ресурсы ЦП. Значит, сколько бы ни было запущено процессов, потреблять память будут только те из них, которые выполняют какие-либо вычисления. В таком случае клиентский процесс также не использует ресурсы ЦП во время ожидания завершения серверного процесса.

Вы также можете быть уверены в том, что два клиента не смогут внести изменения в один и тот же список дел одновременно, так как списком управляет только один процесс. Даже если миллион клиентов захочет произвести какие-либо действия с одним списком, их запросы будут упорядочены и обработаны по одному в соответствующем серверном процессе.

Вы разработали базовую систему для управления несколькими списками дел. Добавим в нее возможность сохранения данных, чтобы они могли быть доступны после перезапуска сервера.

7.3. Сохранение данных

В данном разделе вы расширите функционал кеш-процесса, добавив в него возможность постоянного хранения данных. Основное внимание будет уделено не столько реализации сохранения, сколько пониманию логики работы процессов – тому, как правильно распределять задачи по нескольким серверным процессам, анализировать зависимости и реагировать на узкие места. За основу будет взят код проекта `todo_cache`, который будет постепенно доработан. Вы обеспечите постоянство хранения данных на диске с помощью их кодировки во внешний формат термов Erlang. Полный код этого решения можно найти в папке persistable_todo_cache.

7.3.1. Кодирование и сохранение

Чтобы закодировать произвольный терм Elixir/Erlang, будем использовать функцию *`:erlang.term_to_binary/1`*, принимающую на вход терм Erlang и возвращающую закодированную последовательность байтов, т. е. бинарное значение. Передаваемый функции терм может быть любой сложности, включать глубокие иерархии вложенных списков и кортежей. Результат будет сохраняться на диске, и в дальнейшем при запросе этих данных они будут расшифрованы в терм Erlang с помощью обратной функции *`:erlang.binary_to_term/1`*.

Вооружившись этими знаниями, создадим еще один процесс – базу данных на основе модуля `Todo.Database`. Это будет простой процесс, поддерживающий два вида запросов: `store` и `get`. Для сохранения данных клиенты будут указывать ключ и сами данные. Они будут сохраняться в файл с именем, соответствующим ключу. Такой подход далек от идеала и ненадежен, но он позволит сконцентрироваться на реализации конкурентного выполнения.

Полный код процесса базы данных приведен в следующем листинге.

Листинг 7.7 ❖ Процесс базы данных (persistable_todo_cache/lib/todo/database.ex)

```
defmodule Todo.Database do
  use GenServer

  @db_folder "./persist"

  def start do
    GenServer.start(__MODULE__, nil,
     name: __MODULE__   <-------- Локальная регистрация процесса
    )
```

```
  end

  def store(key, data) do
    GenServer.cast(__MODULE__, {:store, key, data})
  end

  def get(key) do
    GenServer.call(__MODULE__, {:get, key})
  end

  def init(_) do
    File.mkdir_p!(@db_folder)
    {:ok, nil}
  end

  def handle_cast({:store, key, data}, state) do
    key
    |> file_name()
    |> File.write!(:erlang.term_to_binary(data))

    {:noreply, state}
  end

  def handle_call({:get, key}, _, state) do
    data = case File.read(file_name(key)) do
      {:ok, contents} -> :erlang.binary_to_term(contents)
      _ -> nil
    end

    {:reply, data, state}
  end

  defp file_name(key) do
    Path.join(@db_folder, to_string(key))
  end
end
```

Проверка, что такая папка существует

Сохранение данных

Чтение данных

Этот пример по большей части сочетает в себе ранее рассмотренные решения. Сначала на жестко заданное значение папки базы данных устанавливается атрибут модуля `@db_folder`. Как говорилось ранее в разделе 2.3.6, атрибуты во время выполнения используются как константы, что в данном случае позволяет зашифровать имя папки базы данных единожды в коде.

Сервер базы данных локально зарегистрирован под определенным именем, что упрощает взаимодействие с ним и избавляет вас от необходимости постоянно передавать идентификатор `Todo.Database`. Но есть и небольшой недостаток – запустить можно только один экземпляр процесса базы данных.

Стоит заметить, что запрос `store` – это cast-запрос, в то время как `get` – это call-запрос. В данном примере `store` имеет тип cast, потому что клиенту не требуется ответ от сервера. Использование cast-запросов обеспечивает масштабируемость системы, потому что сразу после отправки запроса вызывающий процесс возвращается к выполнению своих задач.

У этого подхода есть огромный минус: вызывающей стороне не сообщается об успехе или неудаче запроса, более того, для нее нет никаких гарантий, что запрос дошел до нужного процесса. Такова особенность запросов cast. Они обеспечивают доступность системы за счет того, что клиентские процессы не тормозятся пос-

ле отправки запроса, но это достигается в ущерб целостности данных, поскольку нельзя знать наверняка, был ли запрос успешным.

На основе данного примера вы реализуете запрос `store` как запрос типа cast, тем самым делая свою систему более масштабируемой и отзывчивой, но не имея гарантии о внесении каких-либо изменений в базу данных.

Во время инициализации используйте функцию `File.mkdir_p!/1` для создания соответствующей папки, если она не была создана ранее. Восклицательный знак в конце имени означает, что функция возбудит ошибку, если папку по какой-то причине не удастся создать. Данные сохраняются путем кодирования заданного терма в бинарную последовательность и размещения ее на диске. Извлечение данных выполняется обратным образом. Если указанного файла на диске не существует, возвращается nil.

7.3.2. Использование базы данных

Процесс базы данных готов, и теперь можно попробовать использовать его в разработанной вами системе. Необходимо сделать следующее:

1) убедиться, что процесс базы данных запущен;
2) пересохранять список дел после каждой модификации;
3) запросить список во время первого извлечения данных.

Для запуска сервера используйте функцию `Todo.Cache.init/1`, как показано в листинге 7.8. Это быстрый и легкий способ, которого на данный момент вполне достаточно.

Листинг 7.8 ❖ Запуск базы данных (persistable_todo_cache/lib/todo/cache.ex)

```
defmodule Todo.Cache do
  ...

  def init(_) do
    Todo.Database.start()
    {:ok, %{}}
  end

  ...
end
```

В данном случае для хранения данных используется подпапка persist текущей папки.

Сохранение данных

Следующим действием необходимо обеспечить сохранение списка после того, как он был изменен. Очевидно, это должно быть сделано сервером. Как вы помните, запросу `store` необходимо передать ключ, в качестве которого будем использовать имя списка дел. Это имя хранится только в кеш-процессе, соответственно, его нужно передать и в серверный процесс тоже, а значит, поменять формат его состояния на `{list_name, todo_list}`. Код этого действия приводиться не будет, но вот список необходимых корректировок:

- функции `Todo.Server.start` нужно принимать имя списка дел и передавать его функции `GenServer.start/2`;

- функция `Todo.Server.init/1` должна использовать этот параметр и сохранять его в состоянии процесса;
- функции обратного вызова `Todo.Server.handle` необходимо обновить для работы с новым форматом состояния.

Запуская новый сервер, кеш-процесс должен передавать ему имя списка дел.

Благодаря вышеописанным преобразованиям серверу теперь известно его имя, и сохранение данных реализуется очень просто.

Листинг 7.9 ❖ Сохранение данных (persistable_todo_cache/lib/todo/server.ex)

```
defmodule Todo.Server do
  ...
  def handle_cast({:add_entry, new_entry}, {name, todo_list}) do
    new_list = Todo.List.add_entry(todo_list, new_entry)
    Todo.Database.store(name, new_list)    <-- Сохранение данных
    {:noreply, {name, new_list}}
  end
  ...
end
```

Проверим, все ли работает. Выполните команду `iex -S mix` и проделайте следующие действия:

```
iex(1)> {:ok, cache} = Todo.Cache.start()

iex(2)> bobs_list = Todo.Cache.server_process(cache, "bobs_list")

iex(3)> Todo.Server.add_entry(bobs_list,
          %{date: ~D[2018-12-19], title: "Dentist"})
```

При успешном выполнении кода на диске создается файл persist/bobs_list.

Чтение данных

Осталось только научить систему считывать данные с диска при запуске сервера. Для этого будем использовать простейшее решение, показанное в листинге 7.10.

Листинг 7.10 ❖ Чтение данных (persistable_todo_cache/lib/todo/server.ex)

```
defmodule Todo.Server do
  ...

  def init(name) do
    {:ok, {name, Todo.Database.get(name) || Todo.List.new()}}
  end

  ...
end
```

В этом примере из базы данных считываются данные или пустой список в случае их отсутствия. В данном случае этого элементарного решения вполне достаточно, но в реальных проектах нужно иметь в виду, что функция обратного вызова `init/1` может работать длительное время. Как вы помните, функция `GenServer.start` возвращает результат только после инициализации процесса. Соответственно, длительное выполнение функции `init/1` приведет к блокировке процесса, за-

пускающего сервер, что, в свою очередь, остановит кеш-процесс, необходимый для работы многочисленных клиентов.

Эту проблему можно аккуратно обойти. Для этого сделайте так, чтобы процесс отправил сообщение самому себе в теле функции `init/1`, и затем проведите инициализацию процесса в соответствующей функции обратного вызова `handle_info`:

```
def init(params) do
  send(self(), :real_init)  <-- Процесс посылает сообщение самому себе
  {:ok, nil}  <-- Инициализация не произошла
end

...

def handle_info(:real_init, state) do
  ...
end
```

Выполнение длительной инициализации

Посылая себе сообщение, процесс добавляет запрос в очередь, после чего функция `init/1` сразу же возвращает результат. Функция `GenServer.start` не блокируется, и вызывающий ее процесс продолжает выполнять другие задачи. В это же время запускается цикл нового процесса и первым же делом обрабатывает первое сообщение в очереди, то есть `:real_init`.

Это решение уместно, только если процесс не был зарегистрирован под локальным именем. Когда процесс не зарегистрирован, для взаимодействия с ним требуется знать его pid, который становится известен только после возвращения функцией `init/1` результата. Следовательно, можно быть уверенным в том, что ваше сообщение будет первым в очереди на обработку.

Однако если процесс зарегистрирован, существует возможность, что какой-то из клиентов отправит запрос первее, используя зарегистрированное имя. Это может случиться потому, что в момент вызова функции `init/1` процесс уже зарегистрирован (благодаря внутренним механизмам `GenServer`). Для решения данной проблемы есть пара хитростей, простейшая из которых – не использовать опцию `:name`, а вместо этого регистрировать процесс вручную в функции обратного вызова `init/1` после отправки сообщения:

```
def init(params) do
  ...

  send(self(), :real_init)
  register(self(), :some_name)  <-- Регистрация имени вручную
end
```

В любом случае теперь чтение данных из базы осуществляется после создания серверного процесса, и это можно легко проверить. Закройте предыдущую сессию оболочки и запустите новую. Сделайте следующее:

```
iex(1)> {:ok, cache} = Todo.Cache.start()
iex(2)> bobs_list = Todo.Cache.server_process(cache, "bobs_list")

iex(3)> Todo.Server.entries(bobs_list, ~D[2018-12-19])
[%{date: ~D[2018-12-19], id: 1, title: "Dentist"}]
```

Как видите, список дел не пустой, а значит, обратное преобразование данных работает должным образом.

7.3.3. Анализ системы

Давайте проанализируем, как работает текущая версия системы. На рис. 7.2 показано взаимодействие процессов системы.

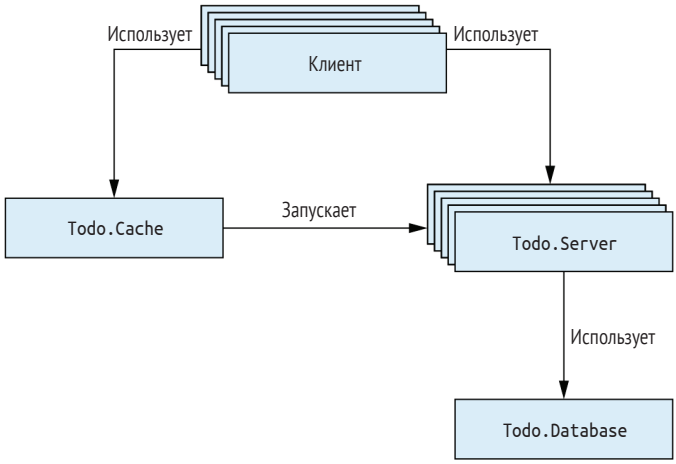


Рис. 7.2 ❖ Взаимодействие процессов

Вы добавили в систему всего один процесс, но это может негативно сказаться на работе системы в целом. Процесс базы данных выполняет кодирование/расшифровку термов и, что еще хуже, операции ввода-вывода на диске. В зависимости от количества клиентов и размеров списков это может значительно снизить производительность системы. Давайте посмотрим на все места в коде, в которых выполняются запросы к базе данных:

```
defmodule Todo.Server do
  ...

  def init(name) do
    {:ok, {name, Todo.Database.get(name) || Todo.List.new()}}   <-- Синхронный запрос
  end

  ...

  def handle_cast({:add_entry, new_entry}, {name, todo_list}) do
    ...
    Todo.Database.store(name, todo_list)    <-- Асинхронный запрос
    ...
  end

  ...
end
```

С точки зрения клиентской стороны, запрос `store` не кажется проблемным, поскольку он выполняется асинхронно: клиент посылает его и тут же возвращается к своим задачам. Но если запросы к базе данных поступают чаще, чем они могут быть обработаны, очередь сообщений в почтовом ящике процесса будет только

расти и поглощать память. В конечном счете это может повлиять на работу системы в целом и даже привести к аварийному завершению процесса ОС BEAM.

Запрос `get` может вызвать дополнительные проблемы. Это синхронный запрос, при котором сервер ожидает ответа от базы данных. Пока сервер находится в ожидании, он не может обрабатывать новые сообщения. Кроме того, так как это происходит с самой инициализации, кеш-процесс блокируется до тех пор, пока не будут извлечены данные. Безусловно, при высоких нагрузках система просто окажется бесполезной.

Стоит снова заметить, что синхронный запрос останавливает сервер на определенное время. Как вы помните, запрос `GenServer.call` делал это в течение пяти секунд, и для достижения большей отзывчивости данный параметр можно уменьшить. И все же даже если запрос не успевает выполниться в указанное время, он не удаляется из почтового ящика процесса-получателя, а остается в нем и снова встает в очередь на обработку.

7.3.4. Устранение узкого места процесса

Очевидно, с возникшей проблемой узкого места процесса базы данных необходимо как-то бороться. Существует множество доступных методов, но здесь будут рассмотрены только некоторые из них.

Отказ от процессов

Самый простой метод ликвидации узкого места процесса – отказаться от использования процесса. Спросите себя: действительно ли так необходим процесс или можно организовать код в виде модуля?

Есть несколько причин запускать участок кода в отдельном серверном процессе:

- этот код управляет состоянием с длительным временем жизни;
- этот код управляет ресурсом, который может и должен быть повторно использован, например TCP-соединение, соединение с базой данных, дескриптор файла, канал процесса ОС и др.;
- критический участок кода должен быть синхронизирован, и только процесс может запускать этот код в любой момент.

Если ни одно из этих условий не актуально, отдельный процесс вам не понадобится. Вы можете спокойно выполнять этот участок кода в клиентских процессах, что позволит избавиться от узкого места процесса и обеспечить параллельное выполнение и масштабируемость.

В рамках текущей задачи действительно можно организовать сохранение данных в файл прямо в серверном процессе. Все операции с одним и тем же списком в одном и том же процессе выполняются по очереди без возникновения состояния гонки. Проблема данного подхода в том, что конкурентное выполнение никак не контролируется. Если 100 000 клиентов захотят одновременно выполнить операции ввода-вывода, это скажется негативным образом на работе всей системы.

Конкурентная обработка запросов

Еще один метод заключается в том, чтобы поддерживать процесс базы данных в рабочем состоянии и обрабатывать в нем операции конкурентно. Это будет

эффективно в том случае, когда запросы основываются на общем состоянии, но могут быть обработаны независимо друг от друга. Принцип работы этого метода можно увидеть на рис. 7.3.

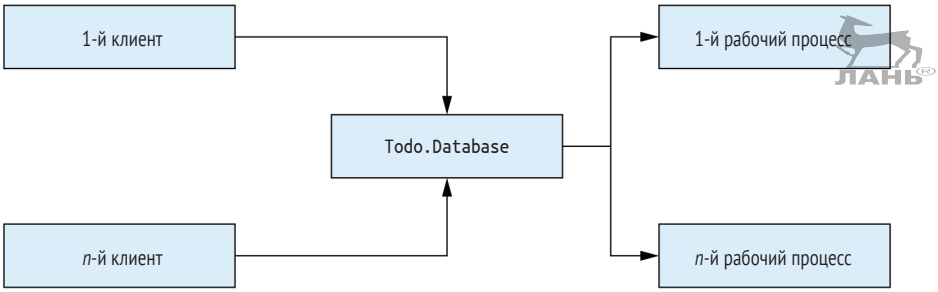


Рис. 7.3 ❖ Конкурентная обработка запросов

Как видите, все запросы по-прежнему упорядочиваются в серверном процессе, но он порождает одноразовые рабочие процессы для обработки каждого из этих запросов. Если код вашего процесса базы данных простой и выполняется быстро, то с помощью большого количества конкурентно запущенных рабочих процессов вы сможете добиться высокой степени масштабируемости.

Чтобы это реализовать, вам необходимо запускать каждую операцию в отдельном одноразовом процессе. Для запросов типа cast необходимо внести изменения в тело их обработчика:

```
def handle_cast({:store, key, data}, state) do
  spawn(fn ->
    key
    |> file_name()                                    Обрабатывается в порожденном процессе
    |> File.write!(:erlang.term_to_binary(data))
  end)

  {:noreply, state}
end
```

Функция-обработчик порождает новый рабочий процесс и незамедлительно возвращает результат. Пока рабочий процесс выполняет обработку, процесс базы данных может принимать новые запросы.

Для запросов типа call реализовать такой же подход уже сложнее, поскольку порожденный процесс должен возвращать ответ:

```
def handle_call({:get, key}, caller, state) do
  spawn(fn ->     <-------------------------------- Порождение процесса для считывания данных
    data = case File.read(file_name(key)) do
      {:ok, contents} -> :erlang.binary_to_term(contents)
      _ -> nil
    end

    GenServer.reply(caller, data)  <---------- Отправка ответа от порожденного процесса
  end)

  {:noreply, state}  <---------- Процесс базы данных ответ не отправляет
end
```

Серверный процесс порождает новый рабочий процесс и затем возвращает `{:noreply, state}`, подавая `GenServer` сигнал о том, что ответ в этом месте отправлен не будет. В то же время порожденный процесс обрабатывает запрос и возвращает ответ вызывающему процессу с помощью функции `GenServer.reply/2`. Это один из тех случаев, когда необходимо использовать второй аргумент функции `handle_call/3` – pid вызывающего процесса и уникальный идентификатор запроса. Эта информация в дальнейшем понадобится порожденному процессу для отправки ответного сообщения.

Данный подход позволяет сократить время обработки в процессе базы данных и при этом добиться конкурентного выполнения операций. Его недостатки перекликаются с недостатками предыдущего метода: конкурентное выполнение никак не контролируется, и большое количество клиентов может совершать операции ввода-вывода на диск одновременно, что в конечном счете приведет к потере отзывчивости системы.

Ограничение конкурентности с помощью пула процессов

Стандартным решением вышеописанной проблемы будет введение ограничения на количество порождаемых процессов. Скажем, процесс базы данных может создавать не более трех рабочих процессов и хранить их pid в своем внутреннем состоянии. Когда процесс получает запрос, он делегирует его одному из рабочих процессов по кругу или по принципу разделения нагрузки. Схему данного решения можно видеть на рис. 7.4.

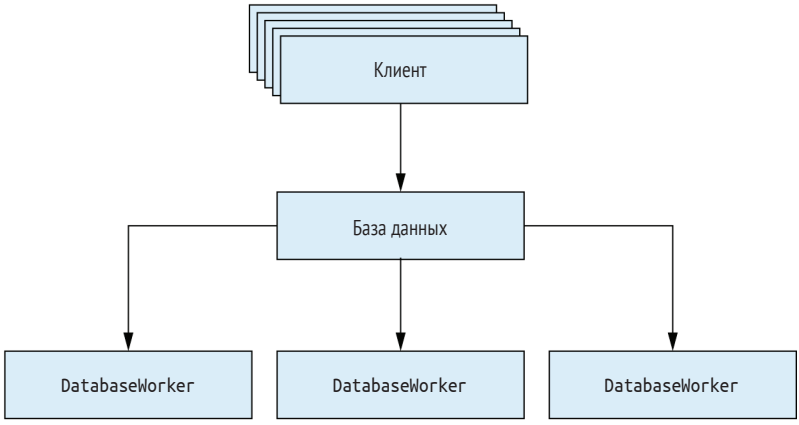


Рис. 7.4 ❖ Использование пула рабочих процессов

Все запросы по-прежнему приходят процессу базы данных, а затем перенаправляются рабочим процессам. Очевидно, данный метод держит конкурентное выполнение под контролем и особенно эффективно работает в системах, в которых недостаточно ресурсов для обработки большого количества конкурентных запросов.

В случае с нашим примером этот метод работает должным образом, его мы и выберем. В каких-либо других ситуациях иной подход может оказаться более эффективным. Смысл этого раздела был в том, чтобы показать вам, как можно выполнить анализ системы на уровне процессов. Всегда помните о том, что не-

сколько процессов могут выполняться конкурентно, в то время как в рамках одного процесса действия последовательны. Если какие-либо вычисления можно распараллелить, подумайте над тем, чтобы запустить их в разных процессах и наоборот, если ту или иную операцию необходимо выполнить синхронно, делайте это в одном и том же процессе.

Пул соединений с базой данных

В данном примере увеличение числа конкурентно выполняющихся процессов не приведет к значительному увеличению производительности. В этом смысле приведенная оптимизация служит больше показательным примером, нежели эффективным решением. Однако в реальности база данных должна быть способна достаточно быстро обрабатывать множество конкурентных запросов. Для этого так или иначе придется ограничить количество одновременных операций с базой данных. Вот для чего необходим пул процессов.

Необходимости в самостоятельной разработке такого пула нет: в экосистеме Elixir/Erlang доступно несколько библиотек с обобщенным кодом, например довольно популярная библиотека Poolboy (**https://github.com/devinus/poolboy**). В зависимости от того, какую библиотеку базы данных вы выбрали, вам необходимо будет либо самим объединить ее с Poolboy (или другим пулом), либо это будет сделано с помощью другой библиотеки (как это делается с использованием библиотеки Ecto (**https://github.com/elixir-lang/ecto**), основанной на PoolBoy). В главе 11, когда вы уже научитесь управлять зависимостями приложения, текущая реализация будет заменена на PoolBoy.

7.3.5. Упражнение: пул процессов и синхронизация

Настало время немного попрактиковаться. В данном упражнении вы заставите базу данных делегировать выполнение операций трем рабочим процессам. Кроме того, необходимо обеспечить синхронизацию по ключу (по имени списка дел) на уровне базы данных: данные под одним и тем же ключом должны обрабатываться одним и тем же рабочим процессом.

Чтобы было не так сложно, я дам вам несколько подсказок.

1. Используйте разработанный вами ранее код и постепенно изменяйте его. Единственное, что потребуется переработать в этот раз, – это реализация модуля `Todo.Database`. Остальные модули оставьте нетронутыми.
2. Создайте новый модуль `Todo.DatabaseWorker`. Он будет практически полной копией имеющегося модуля `Todo.Database`, но процесс базы данных в этот раз не будет зарегистрирован под определенным именем, потому что вам потребуется запустить несколько его экземпляров.
3. Функция `Todo.DatabaseWorker.start` должна получать в качестве аргумента имя папки с базой данных и передавать его вторым аргументом функции `GenServer.start/2`. Этот аргумент затем получает функция обратного вызова `init/1`, после чего он сохраняется в состоянии рабочего процесса.
4. Модуль `Todo.Database` потребуется в значительной степени переработать, но его интерфейс должен остаться прежним. Это означает, что он по-прежнему

будет реализовывать локально зарегистрированный процесс, использовать который можно с помощью функций `start/0`, `get/1` и `store/2`.

5. Во время инициализации `Todo.Database` запустите три рабочих процесса и поместите их pid в словарь с индексами на месте значений (нумерация с нуля).
6. В модуль `Todo.Database` добавьте реализацию запроса `choose_worker`, возвращающего pid рабочего процесса по заданному ключу.
7. `choose_worker` должен всегда возвращать один и тот же идентификатор для одного и того же имени. Самый простой способ этого достичь – ввести подсчет числового хеша ключа и ограничить значение диапазоном [0, 2]. Это можно сделать с помощью вызова `:erlang.phash2(key, 3)`.
8. Функции интерфейса `get` и `store` модуля `Todo.Database` должны вызывать `choose_worker` для получения pid рабочего процесса и обращения к функциям интерфейса модуля `DatabaseWorker`, используя его в качестве первого аргумента.

Старайтесь двигаться небольшими шагами и как можно чаще запускать тесты. Например, реализовав модуль `Todo.DatabaseWorker`, сразу же выполните команду `iex -S mix` и протестируйте его отдельно.

То же самое справедливо и для модуля `Todo.Database`. Сперва инициализируйте состояние и вызовите `IO.inspect` из функции `init/1`, чтобы убедиться в его корректности. Далее реализуйте `choose_worker` и проверьте его работу в оболочке. И наконец, добавьте функции интерфейса и протестируйте систему целиком.

А теперь о том, как узнать, что запросы для разных ключей точно выполняются в разных процессах. Используйте функцию `IO.inspect` в рабочем процессе для вывода на экран текущего pid и ключа. Вы можете написать что-то вроде: `IO.inspect "#{self()}: storing #{key}"`. Используйте эту функцию всегда и везде, она здорово помогает избежать ошибок при разработке.

Если у вас что-то не получилось, готовое решение находится в папке todo_cache_pooling. Разберитесь в нем как следует, ведь в последующих главах оно будет становиться только сложнее.

7.4. Логика работы процессов

Вы уже видели достаточно примеров использования серверных процессов. Целью всех этих примеров было показать, насколько легко с помощью процессов планировать работу сложной конкурентной системы.

Серверный процесс – простейший элемент, что-то вроде конкурентного объекта. На внутреннем уровне он представляет собой сущность, последовательно принимающую и обрабатывающую запросы и сохраняющую свое состояние. Внешне же это конкурентная единица, демонстрирующая четко определенный интерфейс передачи данных.

Еще один способ понимания серверных процессов – представлять их в виде сервисов. Каждый процесс – небольшой сервис, ответственный за какую-либо задачу. В примере со списками дел имеется сервер, управляющий конкретным списком. За обработку разных списков отвечают разные серверы, что делает систему более эффективной. Один и тот же список всегда управляется одним и тем же процессом, что позволяет обеспечить целостность данных и избежать состоя-

ния гонки. Кеш-процесс – это сервис, приводящий в соответствие имена списков и обрабатывающие их серверы. И наконец, процесс базы данных – это сервис, обрабатывающий запросы к базе данных. Внутри себя он распределяет задачи между определенным количеством рабочих процессов, контролируя, чтобы каждый из них всегда имел дело с одним и тем же списком дел.

Все эти сервисы (процессы) в большинстве своем независимы, но в некоторых случаях им требуется взаимодействовать друг с другом. Для этих целей используются синхронные и асинхронные запросы. Очевидно, если клиенту требуется ответ, необходимо использовать запросы типа call, однако иногда они так же эффективно работают, даже если ответ не нужен. Главная проблема cast-запросов – это то, что клиент не получает никаких гарантий успешного выполнения запроса и его результата.

Оба типа запросов имеют свои преимущества и недостатки. Cast-запросы обеспечивают отзывчивость системы (вызывающий процесс не блокируется) в ущерб целостности данных (результат запроса неизвестен). В свою очередь, call-запросы позволяют достичь целостности (вызывающая сторона получает ответ), но снижают отзывчивость системы (процесс блокируется во время ожидания ответа).

Наконец, запросы типа call также могут использоваться для оказания обратного давления на клиентские процессы. Так как эти запросы блокируют клиента, они не дают ему выполнять слишком много задач. Таким образом, клиент синхронизируется с сервером и не может послать запросов больше, чем может обработать сервер. Напротив, при использовании cast-запросов клиенты могут перегрузить серверный процесс, почтовый ящик которого будет ломиться от сообщений и потреблять все больше и больше памяти. В конце концов, память может закончиться, и виртуальная машина прекратит работу.

Эффективность того или иного подхода зависит от конкретной ситуации и обстоятельств. Если вы не можете сделать выбор, попробуйте начать с call-запросов во имя целостности данных. В процессе разработки в определенных местах, где производительность и отзывчивость системы будут падать, вы уже сможете переключиться на cast-запросы.

Выводы

- Если от системы требуется выполнение различных задач, чаще всего лучше запускать их в отдельных процессах. Это обеспечивает отказоустойчивость и масштабируемость системы.
- На внутреннем уровне процесс – последовательная единица, обрабатывающая запросы по очереди. Один процесс может обеспечить целостность своего состояния, но значительно снижает производительность при одновременном обслуживании большого количества клиентов.
- Стоит с умом подходить к выбору типа запроса. Call-запросы являются синхронными и, как следствие, блокируют вызывающий процесс. Если при отправке запроса ответ не требуется, cast-запросы могут повысить производительность, но при этом лишить клиента какой-либо информации об успешности и результате запроса.
- Вы можете использовать проекты `mix` для управления сложными системами, состоящими из множества модулей.

Глава 8

Основы отказоустойчивости

В главе рассматриваются:

- ошибки времени выполнения;
- ошибки в конкурентных системах;
- супервизоры.

Отказоустойчивость – важнейший принцип виртуальной машины BEAM. Erlang изначально создавался для разработки надежных систем, способных продолжать работу даже при возникновении ошибок времени выполнения.

Смысл концепции отказоустойчивости заключается в том, чтобы обнаружить ошибки, минимизировать их влияние на систему и впоследствии устранить их без вмешательства человека. В достаточно сложных системах может случиться что угодно: случайные ошибки, отказ важных компонентов, выход аппаратных средств из строя. Система также может не выдержать перегрузку вследствие растущего количества входящих запросов. В распределенных системах, помимо этих проблем, могут происходить сбои в работе удаленных компьютеров из-за их непредвиденного отключения или разрыва сетевого соединения.

Невозможно знать наверняка, что именно приведет к отказу системы, поэтому придется предусмотреть все возможные случаи, обеспечив хотя бы частичную ее работоспособность при возникновении каких-либо проблем. Например, если сервер базы данных перестанет реагировать на запросы, необходимые данные могут быть получены из кеша. Можно также упорядочить все входящие запросы на сохранение данных и попробовать обработать их позже, когда соединение с базой данных восстановится.

Кроме этого, системе необходимо уметь обнаруживать ошибки и восстанавливаться после них. В случае с предыдущим примером система должна снова и снова пытаться подключиться к базе данных и возобновить обслуживание в полной мере, как только ей это удастся.

Именно так должна выглядеть отказоустойчивая самовосстанавливающаяся система. Что бы ни случилось, она продолжает по возможности выполнять свои задачи и стремится как можно скорее вернуться в рабочее состояние.

Все эти мысли значительным образом меняют подход к обработке ошибок. Вместо того чтобы судорожно пытаться уменьшить количество ошибок, необходимо сконцентрировать свое внимание на сведении их негативных последствий до минимума и автоматическом восстановлении системы. Гораздо лучше иметь несколько изолированных ошибок, чем одну-единственную, выводящую из строя всю систему.

Вы удивитесь, но основным инструментом обработки ошибок в Elixir является конкурентность. Два конкурентных процесса полностью изолированы: они не имеют общей памяти, а аварийное завершение одного процесса никак не влияет на выполнение другого. Благодаря изолированности процессов негативные последствия ошибки распространяются только на один процесс или группу связанных между собой процессов, в то время как все остальные части системы продолжают нормально функционировать.

Когда процесс внезапно завершается, желательно об этом узнать и предпринять какие-либо действия. В этой главе вы изучите основные техники обнаружения и обработки ошибок в конкурентных системах.

Затем в следующей главе вы получите дополнительные знания и сможете реализовать изолирование ошибок на уровне отдельных процессов (fine-grained error isolation).

8.1. Ошибки времени выполнения

В предыдущих главах вскользь упоминались различные ситуации, приводящие к возникновению ошибок. Один из наиболее ярких примеров – неудавшееся сопоставление с образцом. Ошибку времени выполнения можно получить и при отправке синхронного запроса `GenServer.call`, если ответ не приходит в течение заданного промежутка времени (по умолчанию это 5 секунд). Существует множество других примеров: недопустимые арифметические операции (деление на ноль), вызов несуществующей функции и явные сообщения об ошибках.

Когда во время выполнения программы возникает ошибка, управление передается от стека вызовов к коду обработки исключений. Если этот код не присутствует, то процесс, в котором возникла ошибка, завершается. По умолчанию это никак не влияет на работу всех остальных процессов. Разумеется, в Elixir имеются средства перехвата и обработки ошибок, напоминающие те самые конструкции `try-catch` из других языков.

8.1.1. Типы ошибок

Виртуальная машина BEAM различает три типа ошибок времени выполнения: *ошибки*, *выходы* и *выбрасывания*. Вот несколько типичных примеров ошибок:

```
iex(1)> 1/0    <---------- Недопустимое арифметическое выражение
** (ArithmeticError) bad argument in arithmetic expression

iex(1)> Module.nonexistent_function()   <---------- Вызов несуществующей функции
** (UndefinedFunctionError) function Module.nonexistent_function/0 is
   undefined or private

iex(1)> List.first({1,2,3})   <---------- Неудачное сопоставление с образцом
** (FunctionClauseError) no function clause matching in List.first/1
```

Вы также можете *возбудить* ошибку сами с помощью макроса `raise/1`, передав в него строку с сообщением об ошибке:

```
iex(1)> raise("Something went wrong")
** (RuntimeError) Something went wrong
```

Если вы хотите, чтобы функция возбуждала ошибку в явном виде, добавьте к ее имени восклицательный знак. Это соглашение используется в стандартных библиотеках Elixir. Например, функция `File.open!` возбуждает ошибку, если файл не может быть открыт:

```
iex(1)> File.open!("nonexistent_file")
** (File.Error) could not open non_existing_file: no such file or directory
```

Если же вызвать функцию без восклицательного знака в имени, она просто возвратит информацию о невозможности открытия файла:

```
iex(1)> File.open("nonexistent_file")
{:error, :enoent}
```

Обратите внимание, что это не ошибка времени выполнения, а лишь результат функции `File.open`, который можно впоследствии обработать каким-либо способом.

Второй тип ошибок – *выход* – используется для преднамеренного завершения процесса. Для этого необходимо вызвать функцию *exit/1* с указанием причины выхода:

```
iex(2)> spawn(fn ->
          exit("I'm done")    <-- Выход из текущего процесса
          IO.puts("This doesn't happen")
        end)
```

Функции `exit` передается любой терм, описывающий причину завершения процесса. Далее вы увидите, как другие процессы могут обнаруживать этот выход и получать его причину.

Последний тип ошибок – *выбрасывание*. Для его возникновения необходимо вызвать функцию `throw/1`:

```
iex(3)> throw(:thrown_value)
** (throw) :thrown_value
```

Выбрасывания делают возможным нелокальный возврат. Как вы могли видеть в третьей и четвертой главах, программы на Elixir состоят из множества вложенных вызовов функций. Циклы, в частности, реализованы в виде рекурсий. Вследствие этого таких стандартных для многих языков конструкций, как `break`, `continue` и `return`, в Eixir не существует. Выйти из цикла и возвратить значение не так просто, но это можно сделать с помощью выбрасываний: вы можете выбросить значение и затем поймать его в стеке вызовов. Однако с точки зрения потока управления этот подход напоминает механизм `goto` и считается далеко не самым привлекательным, поэтому вам стоит по мере возможности обходить его стороной.

8.1.2. Обработка ошибок

Основным механизмом перехвата и обработки любого из описанных выше типов ошибок является выражение *try*. Можно запустить код и отловить ошибки следующим способом:

```
try do
  ...
```

```
catch error_type, error_value ->
  ...
end
```

Это происходит примерно так же, как и во многих других языках программирования: выполняется код блока do, и если в нем возникает ошибка, управление передается блоку catch.

В блоке catch указывается два параметра. error_type сообщает тип возникшей ошибки и представлен атомами :error, :exit или :throw, а error_value содержит конкретную информацию об ошибке, например выброшенное значение или описание возникшей ошибки.

Попрактикуемся немного с обработкой ошибок. Сперва создадим вспомогательную лямбда-функцию для упрощения работы с ошибками:

```
iex(1)> try_helper = fn fun ->
          try do
            fun.()
            IO.puts("No error.")
          catch type, value ->
            IO.puts("Error\n  #{inspect(type)}\n  #{inspect(value)}")
          end
        end
```

Эта функция принимает другую функцию в качестве аргумента, вызывает ее в блоке try, после чего возвращает тип ошибки и соответствующее значение:

```
iex(2)> try_helper.(fn -> raise("Something went wrong") end)
Error
  :error      ←—— Тип ошибки
  %RuntimeError{message: "Something went wrong"}   ←—— Значение
```

Обратите внимание, что сообщение обернуто в структуру RuntimeError. Эта характерная для Elixir декорация выполняется с помощью макроса raise/1. Если она вам не нужна, используйте функцию Erlang :erlang.error/1 с указанием необходимого терма, который впоследствии будет значением ошибки.

Если вы попытаетесь выбросить какое-либо значение, тип ошибки будет другим:

```
iex(3)> try_helper.(fn -> throw("Thrown value") end)
Error
  :throw
  "Thrown value"
```

Вызов exit/1 также возбудит ошибку своего типа:

```
iex(4)> try_helper.(fn -> exit("I'm done") end)
Error
  :exit
  "I'm done"
```

Как вы помните, любая конструкция в Elixir является выражением и имеет возвращаемое значение. Что касается try, возвращаемым значением будет результат последнего выполненного выражения либо блока do, либо блока catch при возникновении ошибки:

```
iex(5)> result =
          try do
            throw("Thrown value")
          catch type, value -> {type, value}
          end

iex(6)> result
{:throw, "Thrown value"}
```

Стоит также отметить, что значения type и value, указанные в блоке catch, являются образцами. Если вам необходимо обработать ошибку определенного типа, достаточно просто обозначить соответствующие образцы.

Скажем, вы хотели бы незамедлительно возвратить значение из глубоко вложенного цикла. Вы можете вызвать функцию:

```
throw({:result, some_result})
```

А затем обработать это выбрасывание где-то в стеке вызовов:

```
try do
  ...
catch
  :throw, {:result, x} -> x
end
```

В данном примере производится сопоставление с определенной ошибкой времени выполнения – выбрасыванием вида {:result, x}. Если возникнут какие-либо другие ошибки, они не будут перехвачены, а будут переданы вверх по стеку вызовов. Если они не будут обработаны, процесс завершится.

Поскольку блок catch – это, по сути, сопоставление с образцом, вы можете указывать в нем несколько условий, прямо как в случаях с конструкциями case и receive:

```
try do
  ...

catch
  type_pattern_1, error_value_1 ->
    ...

  type_pattern_2, error_value_2 ->
    ...

  ...
end
```

В данном случае выполняется код, который следует после первого удачного сопоставления с образцом, и возвращается результат его последнего выражения.

Если тип и значение пойманной ошибки не важны, используйте образцы type, value или _, _. Они подходят для перехвата любых типов ошибок.

Вы также можете добавить код, который будет обязательно выполняться после блока try, независимо от того, поймал ли catch ошибку:

```
iex(7)> try do
          raise("Something went wrong")
        catch
```

```
    _,_ -> IO.puts("Error caught")
  after
    IO.puts("Cleanup code")    <-- Этот код выполняется всегда
  end
```

```
Error caught
Cleanup code
```

Так как этот код выполняется в любом случае, очень удобно реализовывать в нем очистку ресурсов, например закрытие файла.

Стоит заметить, что код блока `after` никак не влияет на результат всего выражения `try`. Результатом `try` является результат последнего выражения блока `do` или блока `catch`, если была обнаружена ошибка.

Блок try и хвостовые вызовы

В третьей главе была рассмотрена оптимизация хвостовой рекурсии. Если последним действием функции является вызов другой функции (или ее самой), вместо стандартной операции помещения ее параметров в стек происходит что-то вроде безусловного перехода. Когда вызов функции находится в выражении `try`, оптимизация хвостовой рекурсии невозможна. Это вполне очевидно, поскольку последним действием данной функции является выполнение блока `try`, продолжающееся до тех пор, пока не сработают блоки `do` или `catch`. Соответственно, выражение `try` – не последнее действие вызываемой функции, и возможность оптимизации хвостовой рекурсии исключена.

Это далеко не вся информация по обнаружению и обработке ошибок времени выполнения. Elixir предоставляет несколько абстракций поверх этого базового механизма. Вы можете определять свои собственные ошибки с помощью макроса `defexception` (**https://hexdocs.pm/elixir/Kernel.html#defexception/1**) и обрабатывать их немного более красивым способом. Некоторые возможности специальной формы `try` тоже остались неосвещенными. Вы можете узнать о них подробнее, обратившись к официальной документации try (**https://hexdocs.pm/elixir/Kernel.SpecialForms.html#try/1**) или руководству «Getting Started» (**https://elixir-lang.org/getting-started/try-catch-and-rescue.html**).

В данном разделе были представлены основные принципы работы с ошибками времени выполнения. Расширенные возможности Elixir так или иначе сводятся к этим основным принципам и имеют одни и те же особенности:

- ошибка времени выполнения может быть одного из трех типов: `:error`, `:exit` или `:throw`;
- она также имеет значение в виде произвольного терма Elixir;
- если ошибка не будет обработана, процесс, в котором она возникла, завершится.

По сравнению с языками C++, C#, Java и JavaScript, Elixir не обязывает вас отлавливать ошибки времени выполнения. Для него гораздо более характерно позволить процессу завершиться, после чего предпринять какое-либо действие (как правило, перезапустить его). Такой подход может показаться грубым, но на то есть свои причины. В сложных системах большинство ошибок выявляют на этапе

тестирования. Остальные же ошибки попадают под категорию так называемых *гейзенбагов* – непредвиденных, возникающих нерегулярно при определенных обстоятельствах и крайне тяжело воспроизводимых ошибок. Причина возникновения таких ошибок кроется в нарушении целостности состояния. Именно поэтому разумным решением будет завершение процесса и его последующий перезапуск.

Это поможет избавиться от старого состояния процесса (которое может быть повреждено) и начать все заново с новым. Во многих случаях это позволяет решить возникшую проблему – система восстанавливается после непредвиденной ошибки и продолжает обслуживать клиентов. Разумеется, ошибка должна быть занесена в журнал для дальнейшего анализа и выявления причины ее возникновения.

Не переживайте, если прямо сейчас вам не совсем понятна эта логика. Данный подход к обработке ошибок, также известный как «*let it crash*», будет подробно рассмотрен на протяжении этой и следующей глав. А теперь давайте поговорим об основных принципах обработки ошибок в конкурентных системах.

8.2. Ошибки в конкурентных системах

Конкурентность играет ключевую роль в создании отказоустойчивых систем на основе BEAM благодаря полной изолированности и независимости процессов. Аварийное завершение одного процесса не влияет на работу другого (если только вы сами не зададите это в явном виде).

Рассмотрим небольшой пример:

```
iex(1)> spawn(fn ->                    <-- Запуск 1-го процесса
          spawn(fn ->                  <-- Запуск 2-го процесса
            Process.sleep(1000)
            IO.puts("Process 2 finished")
          end)

          raise("Something went wrong")   <-- Возбуждение ошибки в 1-м процессе
        end)
```

Запустив этот код, вы получите следующий результат:

```
17:36:20.546 [error] Process #PID<0.94.0> raised an exception   <-- Вывод регистратора ошибок
...
Process 2 finished   <-- Результат 2-го процесса
```

Как видите, второй процесс продолжает выполняться после отказа первого. Информация о его завершении выводится на экран, а остальные части системы, включая второй процесс и строку приглашения оболочки IEx, работают в обычном режиме.

Более того, так как процессы не разделяют память, при аварийном завершении одного процесса в памяти не останется мусора, который может негативно повлиять на другой процесс. А значит, запуская независимые операции в отдельных процессах, вы автоматически обеспечиваете их изолированность и защиту.

Разработанная ранее система управления списками дел уже пользуется преимуществами изолированных вычислений. Давайте освежим в памяти текущую архитектуру системы, показанную на рис. 8.1.

Каждый блок этой схемы – это отдельный процесс BEAM. Сбой в работе одного сервера не влияет на выполнение операций с другими списками дел. Отказ `Todo.Database` не останавливает чтение из кеша, производимое в серверных процессах.

Как вы могли догадаться, одной изолированности процессов здесь недостаточно. По рис. 8.1 видно, что процессы часто взаимодействуют друг с другом. Если процесс не работает, он не может выполнить необходимую клиенту задачу. Например, если процесс базы данных отказал, серверные процессы не могут отправлять ему запросы и, что гораздо важнее, внесенные в список изменения не будут сохранены. Очевидно, нужно найти способ обнаружения отказа процесса и возвращения его в рабочее состояние, чтобы не допустить такого поведения.

8.2.1. Установка связей между процессами

Основным механизмом обнаружения отказа процесса в Elixir являются *связи*. Если два процесса связаны между собой и один из них неожиданно завершается, второй процесс получает *сигнал выхода* – уведомление о завершении процесса.

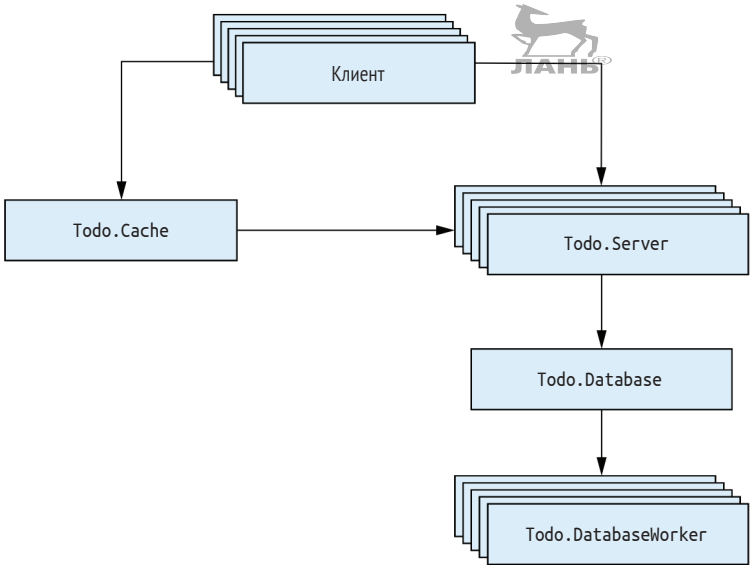


Рис. 8.1 ❖ Изоляция ошибок в системе

Сигнал выхода содержит pid завершившегося процесса и *причину выхода* – терм Elixir, описывающий причину отказа процесса. В случае штатного завершения процесса (окончание работы, породившей его функции) в качестве причины выхода будет выступать атом `:normal`. Если процесс получает от связанного с ним процесса любую другую причину выхода, по умолчанию он тоже завершается.

Связь устанавливается между двумя процессами и всегда работает в обоих направлениях. Чтобы создать связь между текущим процессом и любым другим, используйте функцию `Process.link/1`. Как правило, связывать процессы лучше при их создании при помощи функции *`spawn_link/1`*.

Посмотрим на связи в действии. Создадим два связанных друг с другом процесса и попробуем завершить один из них:

```
iex(1)> spawn(fn ->
          spawn_link(fn ->        <-- Запуск второго процесса и связывание его с первым
            Process.sleep(1000)
            IO.puts("Process 2 finished")
          end)

          raise("Something went wrong")
        end)
```

В результате получается следующее:

```
17:36:20.546 [error] Process #PID<0.96.0> raised an exception
```

Первый процесс завершился аварийно и передал второму сигнал выхода, вследствие чего операция вывода во втором процессе не была выполнена.

Один процесс может быть связан со многими другими процессами, и вы можете создавать сколько угодно таких связей, как показано на рис. 8.2. В таком случае, если один из процессов откажет, он передаст сигналы выхода всем связанным с ним процессам. Если это поведение не переопределить, то данные процессы тоже откажут, и в итоге все дерево связанных процессов перестанет функционировать.

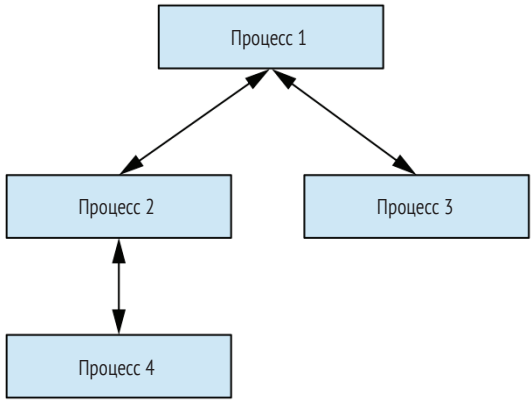


Рис. 8.2 ❖ Связи между несколькими процессами

Перехват сигналов выхода

Сейчас вы наверняка задумались о необходимости такого рода связей. Ранее говорилось о том, как изолированность процессов помогает ограничить распространение негативного эффекта ошибки. Связи исключают эту возможность и прокидывают ошибки за пределы процесса. Связи можно расценивать в качестве коммуникационных каналов для оповещения об отказе процессов.

Как правило, завершение процесса, связанного с отказавшим, крайне нежелательно. Необходимо обнаружить завершение процесса и предпринять определенные действия. Это можно сделать с помощью *перехвата сигналов выхода*. Перехватывая сигнал выхода, процесс не завершается вместе со связанным с ним процессом, а помещает сигнал выхода в свою очередь сообщений в формате стан-

дартного сообщения. Впоследствии это сообщение обрабатывается и выполняются какие-либо предписанные действия.

Чтобы установить ловушку сигналов выхода, используйте вызов `Process.flag(:trap_exit, true)`, как показано в примере ниже:

```
iex(1)> spawn(fn ->
          Process.flag(:trap_exit, true)
          spawn_link(fn -> raise("Something went wrong") end)
          receive do
            msg -> IO.inspect(msg)
          end
        end)
```

Перехват сигналов выхода в текущем процессе

Порождение связанного процесса

Получение и вывод сообщения

Родительский процесс отлавливает сигналы выхода и порождает связанный процесс, обреченный на отказ. Затем он получает сообщение и выводит его на экран. В оболочке вы получите следующий результат:

```
{:EXIT, #PID<0.93.0>,
 {%RuntimeError{message: "Something went wrong"},
  [{:erl_eval, :do_apply, 6, [file: 'erl_eval.erl', line: 668]}]}}
```

`{:EXIT, from_pid, exit_reason}` – это общий формат сообщения сигнала выхода, где `from_pid` – идентификатор завершившегося процесса, а `exit_reason` – терм Elixir, описывающий причину его завершения. Если это произошло из-за выбрасывания или ошибки, причиной выхода будет кортеж `{reason, where}`, элемент where которого – это трассировка стека. Если же процесс завершился вследствие выхода, причиной будет терм, передававшийся функции `exit/1`.

8.2.2. Мониторы

Как отмечалось ранее, связи работают в обоих направлениях. Это играет на руку при реализации очень многих задач, однако в некоторых случаях односторонняя связь может оказаться более эффективной. Иногда необходимо установить такую связь между процессами, чтобы процесс А был оповещен при завершении процесса Б, но не наоборот. Для этого вы можете использовать *монитор* – что-то вроде односторонней связи.

Чтобы организовать мониторинг заданного процесса, используйте функцию *`Process.monitor`*:

```
monitor_ref = Process.monitor(target_pid)
```

Результатом этого выражения будет уникальная ссылка, идентифицирующая монитор. Один процесс может создавать несколько мониторов.

Если отслеживаемый процесс завершается, текущий процесс получает сообщение формата `{:DOWN, monitor_ref, :process, from_pid, exit_reason}`. Одностороннюю связь можно при необходимости разорвать вызовом `Process.demonitor(monitor_ref)`.

Рассмотрим простой пример:

```
iex(1)> target_pid = spawn(fn ->
          Process.sleep(1000)
        end)
```

Порождение процесса, завершающегося через одну секунду

```
iex(2)> Process.monitor(target_pid)
```
Отслеживание порожденного процесса

```
iex(3)> receive do
          msg -> IO.inspect(msg)
        end
```
Ожидание сообщения

```
{:DOWN, #Reference<0.1398266903.3291480065.256365>, :process,
  #PID<0.88.0>, :noproc}
```
Вывод сообщения

Между мониторами и связями можно выделить два основных отличия. Во-первых, мониторы работают только в одном направлении: уведомления получает только создавший монитор процесс. Кроме того, процесс-наблюдатель не завершается при отказе отслеживаемого процесса. Он получает сообщение, которое может проигнорировать или обработать каким-либо образом.

Выходы распространяются при использовании синхронных запросов GenServer

Отправляя синхронный запрос с помощью `GenServer.call`, при отказе серверного процесса вы получите сигнал выхода в клиентском процессе. Это простой, но очень важный пример распространения ошибок на другие процессы. На внутреннем уровне `GenServer` устанавливает временный монитор на серверный процесс. Если во время ожидания ответа от сервера получено сообщение `:DOWN`, `GenServer` обнаруживает отказ процесса и возбуждает соответствующий сигнал выхода в клиентском процессе.

Связи, перехват сигналов выхода и мониторы позволяют обнаружить ошибки в конкурентной системе. Вы можете создать отдельный процесс, единственной задачей которого будет получение уведомлений об аварийном завершении процессов и выполнение предписанных действий. Такие процессы, называемые *супервизорами*, являются основным механизмом восстановления конкурентных систем.

8.3. Супервизоры

Супервизор – это обобщенный процесс, управляющий жизненным циклом процессов системы. Он может запускать другие процессы, которые затем будут являться его потомками. Используя связи, мониторы и перехват выходов, супервизор обнаруживает возможные отказы своих потомков и перезапускает их по мере необходимости.

Процессы, не являющиеся супервизорами, называются *рабочими*. Это все те процессы, которые занимаются обслуживанием клиентов системы. Текущая версия системы управления списками дел реализована на основе одних только рабочих процессов, в числе которых кеш-процесс и несколько серверных процессов.

Если один из рабочих процессов откажет из-за возможной ошибки, определенная часть вашей системы перестанет функционировать. Именно здесь пригодятся супервизоры. Запуская рабочие процессы из супервизора, вы можете обеспечить перезапуск отказавшего процесса и восстановить работоспособность системы.

Чтобы это стало возможным, необходимо добавить в систему хотя бы один процесс-супервизор. В Elixir это делается с помощью модуля *Supervisor* (**https://hexdocs.pm/elixir/Supervisor.html**). Вызвав функцию `Supervisor.start_link/2`, вы запустите супервизор, который будет работать следующим образом.

1. Супервизор отлавливает сигналы выхода и затем запускает дочерние процессы.
2. Если какой-либо из дочерних процессов внезапно завершается, супервизор получает соответствующее сообщение о выходе и выполняет указанные действия (например, перезапускает этот процесс).
3. Если отказывает сам супервизор, его дочерние процессы также завершаются.

Существует два способа запуска супервизора. Первый и основной – вызвать функцию `Supervisor.start_link`, передав ей список всех планируемых потомков и некоторые дополнительные параметры. Второй способ – передать этой функции модуль, определив в нем функцию обратного вызова, возвращающую эту информацию. Пока остановимся на первом, а второй попробуем чуть позже.

Добавим в существующую систему один супервизор. Как показано на рис. 8.3, система состоит из следующих процессов:

- `Todo.Server` – позволяет нескольким клиентам взаимодействовать с одним и тем же списком дел;
- `Todo.Cache` – хранит коллекцию серверных процессов и отвечает за их создание и обнаружение;
- `Todo.DatabaseWorker` – выполняет операции чтения из базы данных и записи в нее;
- `Todo.Database` – управляет пулом рабочих процессов базы данных и направляет к ним запросы.

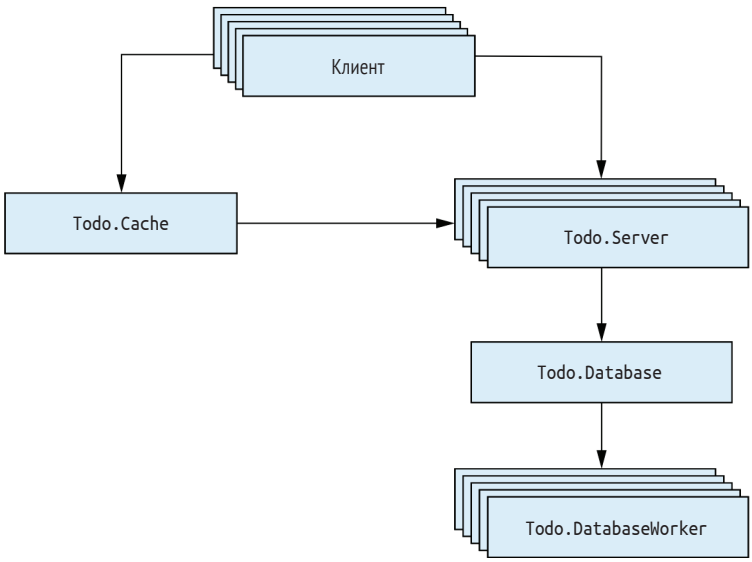


Рис. 8.3 ❖ Процессы текущей системы

Кеш-процесс является точкой входа системы. Когда он запускается, запускаются и все необходимые процессы, поэтому его можно считать главным элементом системы. Новый процесс-супервизор, который мы собираемся добавить, будет осуществлять наблюдение как раз за кеш-процессом.

8.3.1. Подготовка существующего кода

Перед тем как начать работать над супервизором, необходимо внести несколько изменений в текущий код кеш-процесса.

Во-первых, его нужно зарегистрировать. Это позволит обращаться к нему без pid. Во-вторых, для запуска кеш-процесса из супервизора обязательно необходимо добавить связь. Супервизоры используют связи, а не мониторы, потому что они работают в обоих направлениях, а значит, отказ супервизора приведет к отказу всех его дочерних процессов. В свою очередь, это позволяет вам технично вывести из строя определенную часть системы целиком, не оставляя никаких «бесхозных» процессов. В этой и следующей главах вы увидите, как это работает, на реальных примерах.

Создать связь с вызывающим процессом очень просто: необходимо всего лишь заменить функцию `GenServer.start` на `GenServer.start_link`. Также переименуйте соответствующую функцию интерфейса модуля `Todo.Cache` в `start_link`.

И последнее, передайте функции `start_link` один пустой аргумент. Это немного упростит запуск супервизора. Вы узнаете об этом аргументе далее в этой главе, когда речь пойдет о спецификациях потомков.

Все изменения показаны в следующем листинге.

Листинг 8.1 ❖ Изменения в коде кеш-процесса (supervised_todo_cache/lib/todo/cache.ex)

```
defmodule Todo.Cache do
  use GenServer

  def start_link(_) do                                  ← Переименованная функция интерфейса
    IO.puts("Starting to-do cache.")                    ← Отладочное сообщение
    GenServer.start_link(__MODULE__, nil, name: __MODULE__)   ← Регистрация процесса и связывание его с текущим процессом
  end

  def server_process(todo_list_name) do
    GenServer.call(__MODULE__, {:server_process, todo_list_name})   ← Функция интерфейса, использующая зарегистрированное имя
  end

  ...
end
```

Обратите внимание, что в функции `start_link` также вызывается функция `IO.puts/1`. Это делается во всех остальных процессах для получения информации во время отладки.

8.3.2. Запуск процесса-супервизора

После внесения этих изменений вы можете попробовать запустить супервизор, в потомках которого будет только один кеш-процесс. Переименуйте текущую

папку в supervised_todo_cache и запустите оболочку с помощью команды `iex -S mix`. Выполните следующее:

```
iex(1)> Supervisor.start_link([Todo.Cache], strategy: :one_for_one)
Starting to-do cache.
Starting database server.
Starting database worker.
Starting database worker.
Starting database worker.
```

Запуск супервизора и дочернего кеш-процесса

Как можно видеть из полученного результата, вызов функции `Supervisor.start_link/2` запускает кеш-процесс, который затем запускает процессы базы данных.

Рассмотрим функцию `Supervisor.start_link/2` более подробно:

```
Supervisor.start_link(
  [Todo.Cache],              ← Список спецификации потомков
  strategy: :one_for_one     ← Стратегия супервизора
)
```

Функция `Supervisor.start_link/2` запускает процесс-супервизор и связывает его с вызывающим процессом.

Первый аргумент функции – это список планируемых потомков супервизора. Если быть точнее, то каждый элемент этого списка – это спецификация потомка, описывающая принципы его запуска и управления им. Все это будет рассмотрено более детально чуть позже. В данном случае в качестве спецификации потомка указано имя модуля.

После запуска супервизор проходит этот список и запускает каждый дочерний процесс в соответствии с его спецификацией. В текущем примере супервизор вызывает функцию `Todo.Cache.start_link/1`. Как только все потомки будут запущены, функция `Supervisor.start_link/2` возвращает `{:ok, supervisor_pid}`.

Второй аргумент функции `Supervisor.start_link/2` – это список специальных опций супервизора. Опция `:strategy`, называемая *стратегией перезапуска*, обязательна. Она определяет то, каким образом супервизору необходимо перезапускать дочерние процессы. Стратегия `one_for_one` означает, что если один дочерний процесс завершился, он должен быть перезапущен. Существуют и другие стратегии, и они будут рассмотрены далее в главе 9.

> **ПРИМЕЧАНИЕ** Слово «перезапуск» в отношении процессов употреблено с одной оговоркой. Технически процесс не может быть перезапущен: когда он завершается, вместо него запускается новый процесс на основе того же модуля. Он имеет другой pid и не разделяет состояние со старым процессом.

После того как функция `Supervisor.start_link/2` вернет результат, все необходимые процессы системы будут уже запущены, и системой можно пользоваться. Например, можно запустить серверный процесс:

```
iex(2)> bobs_list = Todo.Cache.server_process("Bob's list")
Starting to-do server for Bob's list.
#PID<0.116.0>
```

Кеш-процесс запустился в качестве потомка процесса-супервизора. Это означает, что если кеш-процесс потерпит сбой, супервизор его перезапустит.

Это можно легко проверить, заставив кеш-процесс аварийно завершиться. Для этого нужно узнать его pid. Как известно, кеш-процесс зарегистрирован под определенным именем (именем его модуля), значит, получить его pid можно с помощью функции `Process.whereis/1`:

```
iex(3)> cache_pid = Process.whereis(Todo.Cache)
#PID<0.110.0>
```

Теперь уничтожим процесс, используя функцию `Process.exit/2`, принимающую на вход pid процесса и причину выхода и передающую соответствующий сигнал выхода указанному процессу. Причиной выхода может быть любой терм, но в данном случае будем использовать атом `:kill`, воспринимаемый системой особым образом. Если причиной выхода указать `:kill`, то это будет означать, что процесс должен завершиться в обязательном порядке, даже если в нем реализован механизм перехвата сигналов выхода. Посмотрим, как это работает:

```
iex(4)> Process.exit(cache_pid, :kill)
Starting to-do cache.
Starting database server.
```

Как видите, кеш-процесс сразу же перезапускается. Вы также можете убедиться в том, что новый процесс запущен под другим pid:

```
iex(5)> Process.whereis(Todo.Cache)
#PID<0.119.0>
```

Новый процесс можно использовать точно так же, как и старый:

```
iex(6)> bobs_list = Todo.Cache.server_process("Bob's list")
Starting to-do server for Bob's list. #PID<0.122.0>
```

Наш небольшой эксперимент иллюстрирует некоторые базовые возможности механизма отказоустойчивости Elixir. Система сама восстанавливается после аварии и продолжает выполнять свои функции в полном объеме.

Имена помогают обнаруживать процессы

Важно разъяснить, для чего кеш-процесс был зарегистрирован под локальным именем. Не стоит забывать, что для взаимодействия с процессом так или иначе требуется знать его pid. В предыдущей главе использовался примитивный подход, напоминающий подход, характерный для разработанных на объектно-ориентированных языках систем. Вы создавали процесс и затем передавали функциям его pid. Это работает без нареканий до тех пор, пока в игру не вступят супервизоры.

Проблема в том, что потомки супервизоров не могут быть перезапущены. Вместо упавшего процесса запускается другой процесс со своим pid. А значит, все ссылки на pid старого процесса становятся недействительными, поскольку этого процесса больше не существует.

Именно поэтому такие процессы важно регистрировать. Имена позволяют легко обнаружить процесс и осуществлять взаимодействие с ним, даже если он был перезапущен.

8.3.3. Спецификации потомков

Для управления потомками супервизору требуется информация о том:

- как следует запускать потомков;
- что следует предпринять, если один из потомков аварийно завершится;
- какой терм должен быть использован для уникального определения каждого из потомков.

Эти данные в совокупности называются *спецификациями потомков*. Вспомните, как в функцию `Supervisor.start_link/2` передавался список спецификаций потомков. В общем виде спецификация представляет собой словарь, содержащий несколько полей, определяющих свойства потомка.

Например, вот так будет выглядеть спецификация для кеш-процесса:

```
%{
  id: Todo.Cache,    <-- Идентификатор потомка
  start: {Todo.Cache, :start_link, [nil]},    <-- Спецификация запуска
}
```

Поле `:id` – это терм Elixir, по которому супервизор может отличить этот дочерний процесс от других своих потомков. Поле `:start` представлено кортежем с тремя элементами в виде `{module, start_function, list_of_arguments}`. Запуская дочерний процесс, супервизор вызывает функцию `module.start_function`, передавая ей указанный список аргументов. Эта функция запускает процесс и связывает его с супервизором.

Есть и другие поля, при наличии которых производится более тонкая настройка супервизора. Некоторые из них будут рассмотрены в следующей главе, но вы всегда можете найти информацию о них в официальной документации (**https://hexdocs.pm/elixir/Supervisor.html#module-child-specification**).

Вы можете передать этот словарь со спецификациями напрямую функции `Supervisor.start_link`, как показано в примере:

```
Supervisor.start_link(
  [
    %{
      id: Todo.Cache,
      start: {Todo.Cache, :start_link, [nil]}
    }
  ],
  strategy: :one_for_one
)
```

Для запуска кеш-процесса супервизор вызывает функцию `Todo.Cache.start_link(nil)`. Как вы помните, вы сделали так, чтобы эта функция принимала один аргумент, и в данном случае в нее передается значение `nil`.

Стоит отметить, что данный способ запуска потомков не самый надежный. Если, к примеру, сигнатура запускающей функции кеш-процесса поменяется, необходимо помнить о том, что спецификацию тоже необходимо будет переработать в соответствии с этими изменениями.

Эту проблему можно решить, передав кортеж `{module_name, arg}` в списке спецификаций потомка. В этом случае супервизор первым делом будет вызывать `mod-`

ule_name.child_spec(arg) для получения спецификаций в форме словаря, после чего он запустит дочерний процесс согласно указанным спецификациям.

Модуль Todo.Cache уже содержит определение функции child_spec/1, хотя вы и не добавляли его сами. Оно было внедрено во время выполнения выражения use GenServer. Благодаря этому вы можете запустить супервизор следующим образом:

```
Supervisor.start_link(
  [{Todo.Cache, nil}],
  strategy: :one_for_one
)
```

В результате супервизор вызывает Todo.Cache.child_spec(nil) и запускает дочерний процесс в соответствии с возвращенной спецификацией. Проверить, что именно возвращает автоматически добавленная реализация функции child_spec/1, очень легко:

```
iex(1)> Todo.Cache.child_spec(nil)
%{id: Todo.Cache, start: {Todo.Cache, :start_link, [nil]}}
```

Другими словами, сгенерированная функция child_spec/1 возвращает спецификацию, которая вызывает функцию start_link/1 текущего модуля с аргументом, переданным child_spec/1. Именно для этого вы сделали так, чтобы функция Todo.Cache.start_link принимала один пустой аргумент:

```
defmodule Todo.Cache do
  use GenServer            <-- Генерация функции child_spec/1 по умолчанию

  def start_link(_) do     <-- Приведение в соответствие с child_spec/1
    ...
  end

  ...
end
```

Таким образом, модуль Todo.Cache становится совместимым с генерируемой по умолчанию функцией child_spec/1, что позволяет включить кеш-процесс в список потомков супервизора без каких-либо дополнительных действий.

Если такой подход вам не нравится, вы можете настроить результат, возвращаемый генерируемой функцией child_spec/1, указав несколько опций при вызове use GenServer. Более подробную информацию об этом вы можете найти в официальной документации (**https://hexdocs.pm/elixir/GenServer.html#module-use-genserver-and-callbacks**). Если и этого недостаточно, просто определите child_spec/1 самостоятельно, и эта реализация перекроет стандартную.

Теперь кеш-процесс можно запустить следующим образом:

```
Supervisor.start_link(
  [Todo.Cache],
  strategy: :one_for_one
)
```

Перед тем как перейти к следующей теме, еще раз проговорим, что происходит при запуске супервизора, а именно после вызова Supervisor.start_link(child_specs, options).

1. Запускается новый процесс на основе модуля Supervisor.

2. Процесс-супервизор запускает по очереди все процессы, указанные в списке спецификаций потомков.
3. При необходимости спецификации запрашиваются и применяются с помощью вызова функции `child_spec/1` из соответствующего модуля.
4. Супервизор запускает дочерний процесс в соответствии с информацией, указанной в поле `:start` его спецификации.

8.3.4. Обертка супервизора

До этого момента вы использовали супервизор только из оболочки, но в реальности вам необходимо будет работать с ним в коде. Аналогично `GenServer`, `Supervisor` обычно оборачивают в модуль.

В следующем листинге показано, как должен выглядеть модуль супервизора.

Листинг 8.2 ❖ Модуль супервизора (supervised_todo_cache/lib/todo/system.ex)

```
defmodule Todo.System do
  def start_link do
    Supervisor.start_link(
      [Todo.Cache],
      strategy: :one_for_one
    )
  end
end
```

Всего одна небольшая правка – и систему можно запустить одним простым действием:

```
$ iex -S mix

iex(1)> Todo.System.start_link()

Starting to-do cache.
Starting database server.
Starting database worker.
Starting database worker.
Starting database worker.
```

Название модуля `Todo.System` говорит о назначении модуля. Вызывая функцию `Todo.System.start_link()`, вы запускаете всю систему со всеми необходимыми сервисами, включая кеш и базу данных.

8.3.5. Использование модуля обратного вызова

Еще один способ запустить супервизор – добавить в систему модуль обратного вызова. Это работает так же, как и в случае с `GenServer`: вы должны разработать модуль, содержащий реализацию функции `init/1`, возвращающую список спецификаций потомков и дополнительные опции супервизора.

Чтобы воспользоваться данным способом, перепишите реализацию модуля `Todo.System` следующим образом:

```
defmodule Todo.System do
  use Supervisor  <-------- Внедрение шаблонного кода
```

```
  def start_link do
    Supervisor.start_link(__MODULE__, nil)
  end

  def init(_) do
    Supervisor.init([Todo.Cache], strategy: :one_for_one)
  end
end
```

Запуск супервизора с модулем обратного вызова Todo.System

Реализация необходимой функции обратного вызова

Сначала для добавления в модуль стандартных реализаций некоторых функций выполняется выражение `use Supervisor`. Самое интересное происходит при вызове функции `Supervisor.start_link/2`. Вместо списка спецификаций потомков в нее передается модуль обратного вызова. В этом случае процесс-супервизор вызовет функцию `init/1`, чтобы получить спецификацию супервизора. Аргумент, передаваемый этой функции, также передается вторым аргументом в функцию `Supervisor.start_link/2`. И наконец, в функции `init/1` путем передачи списка потомков и опций в функцию `Supervisor.init/2` определяется сам супервизор.

Предыдущий код является более продуманным аналогом `Supervisor.start_link([Todo.Cache], strategy: :one_for_one)`. Да, в нем больше строк, но зато он предоставляет вам больше контроля. Например, если перед запуском дочерних процессов требуется выполнить дополнительную инициализацию, вы можете сделать это в функции `init/1`. Кроме этого, модуль обратного вызова более приспособлен к горячему обновлению кода: список потомков можно изменять, не перезапуская супервизор.

В большинстве случаев будет вполне достаточно передать список спецификаций потомков супервизору напрямую. Так же, как вы видели из рассмотренных примеров, если обернуть код супервизора в отдельный модуль, можно легко переключиться с этого элементарного подхода на более сложный с модулем обратного вызова. Далее в книге будет использован исключительно первый подход.

8.3.6. Связывание всех процессов

На данный момент супервизор контролирует кеш-процесс, что обеспечивает базовую отказоустойчивость системы: если кеш-процесс аварийно завершается, вместо него запускается новый процесс, и система продолжает работать в штатном режиме.

Однако текущая реализация не без изъяна. Когда супервизор перезапускает кеш-процесс, выстраивается отдельная иерархия процессов и появляется новый набор серверных процессов, никак не связанных с предыдущими. Старые серверные процессы при этом окажутся ненужными, но при этом будут продолжать работать и поглощать память и ресурсы ЦП.

Рассмотрим это на примере. Запустим систему и создадим один серверный процесс:

```
iex(1)> Todo.System.start_link()

iex(2)> Todo.Cache.server_process("Bob's list")
Starting to-do server for Bob's list.
#PID<0.141.0>
```

При отправке такого же запроса снова еще один серверный процесс не запускается:

```
iex(3)> Todo.Cache.server_process("Bob's list")
#PID<0.141.0>
```

Проверим число запущенных процессов:

```
iex(4)> :erlang.system_info(:process_count)
60
```

Теперь завершим кеш-процесс:

```
iex(5)> Process.exit(Process.whereis(Todo.Cache), :kill)
Starting to-do cache.
Starting database server.
```

И наконец, запустим серверный процесс для списка Bob:

```
iex(6)> Todo.Cache.server_process("Bob's list")
Starting to-do server for Bob's list.
#PID<0.147.0>
```

Как видите, после перезапуска кеш-процесса тот же запрос создает новый серверный процесс, и это неудивительно, поскольку предыдущий кеш-процесс был уничтожен, а вместе с ним было уничтожено и его состояние.

Когда процесс завершается, его состояние удаляется, а новый процесс запускается уже с новым состоянием. Если вам необходимо сохранить старое состояние, то придется делать это вручную. Вы узнаете об этом подробнее в главе 9.

После перезапуска кеш-процесса вы получаете совершенно новый процесс, не имеющий информации о данных предыдущего процесса. В то же время структура старого кеш-процесса (его серверные процессы) продолжает существовать. В этом можно удостовериться, запросив количество запущенных процессов:

```
iex(7)> :erlang.system_info(:process_count)
61
```

Один лишний процесс – это ранее запущенный серверный процесс для списка Bob. Очевидно, эту проблему необходимо решить и при завершении кеш-процесса удалять не только его состояние, но и завершать все существующие серверные процессы.

Для этого процессы необходимо связать между собой. Необходимо обеспечить связи: каждого серверного процесса с кешем, сервера базы данных с кешем, рабочих процессов с процессом базы данных. Таким образом, все компоненты системы окажутся взаимосвязанными, как показано на рис. 8.4.

Так выглядит основной метод обеспечения согласованности процессов в Elixir. Связывая группу взаимозависимых процессов, вы можете быть уверены в том, что при завершении одного процесса все связанные с ним процессы так же будут завершены, независимо от того, какой из процессов откажет. Это непременно приведет к отказу кеш-процесса, оповещению супервизора и запуску новой версии системы.

Именно так наиболее правильно обеспечивать восстановление работоспособности системы после ошибки. Когда в какой-либо части системы обнаруживается

ошибка, эта часть целиком перезапускается, а старая ее версия и ее данные уничтожаются. Недостаток данного подхода в слишком большой цене ошибки – сбой в одном-единственном рабочем процессе базы данных или серверном процессе приведет к отказу всей иерархии процессов. Такая схема работы далека от совершенства, и в главе 9 вы сможете это исправить.

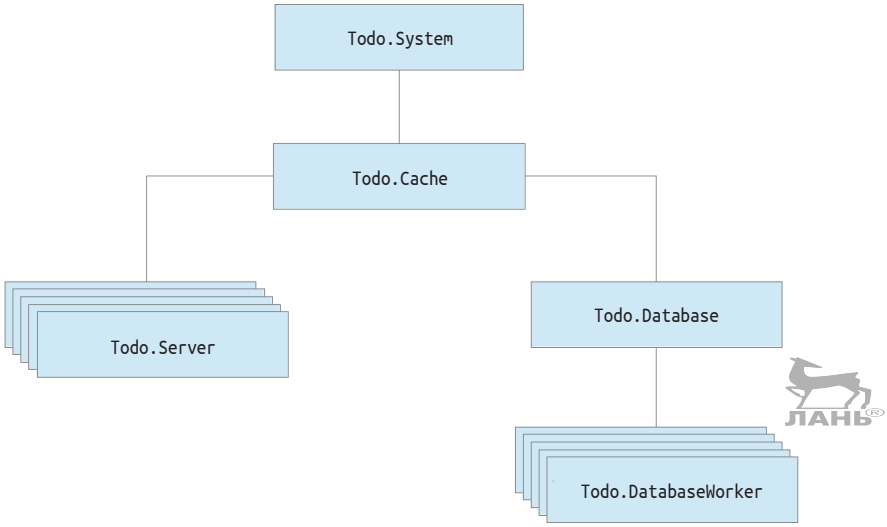


Рис. 8.4 ❖ Связи между всеми процессами системы

А пока возьмем за основу описанный выше метод и реализуем необходимый код. В текущей системе имеется супервизор, запускающий кеш-процесс и осуществляющий слежение за ним. Необходимо прямо или косвенно связать кеш-процесс со всеми остальными рабочими процессами.

Это делается очень просто. Замените функцию `start` на `start_link` во всех процессах системы. Что у вас есть:

```
def start(...) do
  GenServer.start(...)
end
```

Что должно получиться:

```
def start_link(...) do
  GenServer.start_link(...)
end
```

Разумеется, все функции `Module.start` следует переименовать в `Module.start_link`. Это чисто механические правки, и нет смысла приводить здесь весь код. Вы можете найти его в папке todo_link.

Давайте посмотрим, как работает новая система:

```
iex(1)> Todo.System.start_link()

iex(2)> Todo.Cache.server_process("Bob's list")
Starting to-do server for Bob's list.
```

```
iex(3)> :erlang.system_info(:process_count)
60

iex(4)> Process.exit(Process.whereis(Todo.Cache), :kill) ◄— Завершение всей иерархии процессов

iex(5)> bobs_list = Todo.Cache.server_process("Bob's list")
Starting to-do server for Bob's list.

iex(6)> :erlang.system_info(:process_count)
60 ◄———————— Число процессов остается прежним
```

Теперь, когда один процесс отказывает, благодаря установленным связям все его зависимые части также завершаются, и вместо него запускается новый процесс. Это обеспечивает согласованность системы.

8.3.7. Частота перезапусков

Важно понимать, что супервизор не будет перезапускать дочерние процессы постоянно. Существует так называемая максимальная *частота перезапусков*, определяющая максимально возможное количество перезапусков в единицу времени. По умолчанию эта величина составляет три перезапуска в пять секунд. Изменить эти значения можно, передав опции `:max_restarts` и `:max_seconds` в вызов функции `Supervisor.start_link/2`. Если эта частота будет превышена, супервизор и все его потомки завершатся.

Посмотрим, как это выглядит в оболочке. Для начала запустите супервизор:

```
iex(1)> Todo.System.start_link()
Starting the to-do cache.
```

Теперь сымитируем частые перезапуски кеш-процесса:

```
iex(1)> for _ <- 1..4 do
          Process.exit(Process.whereis(Todo.Cache), :kill)
          Process.sleep(200)
        end
```

В данном коде запускается цикл, в котором кеш-процесс завершается, а после ожидания в 2 секунды он перезапускается супервизором. Это делается 4 раза подряд, и при этом стандартное значение максимальной частоты перезапусков превышается.

Вот что получится в результате:

```
Starting the to-do cache.
Starting database server.   | Повторяется три раза
...

** (EXIT from #PID<0.107.0>) :shutdown ◄——— Завершение супервизора
```

Как только максимальное значение частоты перезапусков превышается, супервизор прекращает работу и завершается вместе со всеми своими потомками.

В чем же смысл данного механизма? Когда один из основных процессов системы падает, супервизор пытается вернуть его к жизни с помощью запуска нового процесса. Если это не помогло, то нет смысла продолжать эти попытки бесконечно – очевидно, таким способом проблема не может быть решена. В этом случае

единственным разумным решением является остановка работы супервизора и ее завершение.

Этот механизм является неотъемлемой частью *деревьев супервизоров* – более сложной иерархии супервизоров и рабочих процессов, позволяющей вам контролировать ход восстановления системы после ошибок. Это будет рассмотрено во всех подробностях в следующей главе.

Выводы

- Существует три типа ошибок времени выполнения: выбрасывания, ошибки и выходы.
- Когда возникает ошибка времени выполнения, управление передается блоку `try`. Если ошибку не обработать, процесс завершится.
- Аварийное завершение процесса можно обнаружить с помощью другого процесса. Для этого используются связи и мониторы.
- Связи работают в обоих направлениях: отказ одного из процессов также приводит к отказу связанного с ним процесса.
- По умолчанию все процессы, связанные с отказавшим, также завершаются. С помощью перехвата сигналов выхода вы можете обнаружить отказ процесса и выполнить определенные действия.
- Супервизор – это процесс, управляющий жизненными циклами других процессов. Он может запускать процессы, наблюдать за ними, а также перезапускать их после аварии.
- Для запуска супервизоров и работы с ними используется модуль `Supervisor`.
- При определении супервизора указывается список спецификаций его потомков и стратегия наблюдения. Эти параметры можно передать функции `Supervisor.start_link/2` или реализовать модуль обратного вызова.

Глава 9

Изолирование последствий ошибок

В главе рассматриваются:

- деревья супервизоров;
- динамический запуск рабочих процессов:
- стратегия Let it crash.

В предыдущей главе вы познакомились с теоретической информацией касательно подхода к обработке ошибок в конкурентных системах на основе супервизоров. Основная его идея заключается в наличии процесса, единственной задачей которого является наблюдение за остальными процессами и их перезапуск после отказа. Это позволяет вам разобраться с непредвиденными ошибками в системе: по какой бы причине рабочий процесс ни завершился, супервизор обнаружит ошибку и перезапустит его.

Супервизоры не только предоставляют стандартный механизм обнаружения ошибок и восстановления системы, но и играют важную роль в изолировании последствий ошибок. Запуская независимые рабочие процессы из одного супервизора, вы ограничиваете распространение негативного эффекта ошибки в пределах одного рабочего процесса, что положительным образом сказывается на доступности системы. Как бы вы ни пытались предотвратить непредвиденные ошибки, шанс их возникновения все равно остается. Изолирование последствий таких ошибок позволяет другим составляющим системы сохранить работоспособность, пока отказавшие части восстанавливаются.

Если говорить о главном примере этой книги, то ошибка в процессе базы данных не должна привести к отказу кеш-процеса. Пока часть, отвечающая за базу данных, восстанавливается, система должна продолжать хотя бы частично функционировать, предоставляя существующие в кеше данные. Более того, ошибки в отдельных рабочих процессах не должны оказывать влияние на другие операции с базой данных. Если вам удастся «заточить» последствия ошибки в одной небольшой части, ваша система будет способна обслуживать клиентов беспрерывно.

Изолирование ошибок и сведение к минимуму их негативных последствий – тема текущей главы. Чтобы этого добиться, необходимо организовать запуск каждого рабочего процесса в отдельном супервизоре, и тогда каждый из них будет

перезапускаться независимо от других. Вы увидите, как это работает, уже в следующем разделе, когда начнете разрабатывать мелкоструктурное дерево супервизоров.

9.1. Деревья супервизоров

В данном разделе мы поговорим о том, как снизить негативное влияние ошибки на целую систему. Основными инструментами для этого послужат процессы, связи и супервизоры, да и сам подход довольно прост. Необходимо всегда учитывать, что случится с системой при отказе одного из процессов, и принимать соответствующие меры, в случае если последствия ошибки слишком значительны (задет не один, а несколько процессов).

9.1.1. Разделение слабо связанных частей

Рассмотрим распространение ошибок в текущей модели системы. Связи между процессами показаны на рис. 9.1.

Как видите, все процессы взаимосвязаны. Не важно, какой из них потерпит аварию, сигнал выхода будет передан всем связанным с ним процессам. В конечном счете завершится и кеш-процесс, `Todo.Supervisor` заметит это и перезапустит его.

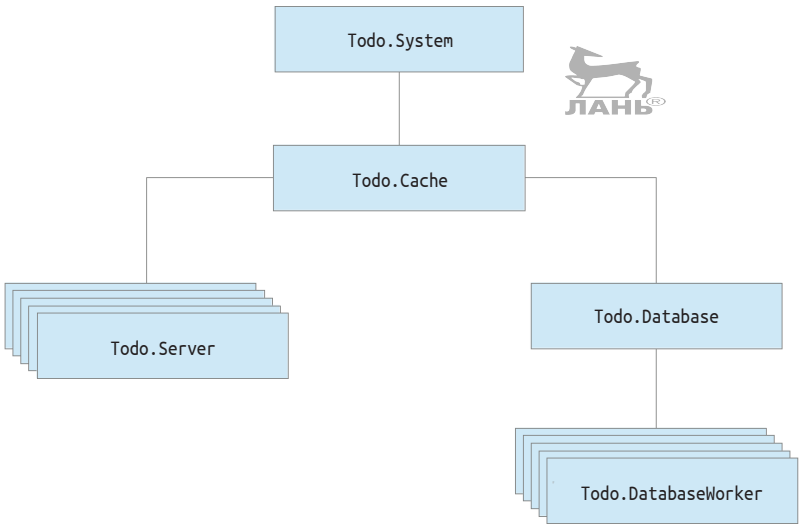


Рис. 9.1 ❖ Связи между процессами в текущей системе

Так выглядит наиболее правильный подход к обработке ошибок: система перезапускается, а старая ее версия и все ее данные удаляются. Однако этот подход слишком грубый, так как всего одна ошибка приводит к останову и перезапуску всей системы. В текущей системе если ошибка возникнет в одном из рабочих процессов базы данных, то кеш-процесс аварийно завершится. Аналогичным образом ошибка одного из серверных процессов выведет из строя все рабочие процессы базы данных.

Так происходит, потому что рабочие процессы запускаются другими рабочими процессами, например сервер базы данных запускается кеш-процессом. Чтобы уменьшить последствия ошибок, необходимо запускать каждый рабочий процесс в отдельном супервизоре, и тогда они будут перезапускаться независимо друг от друга.

Попробуем это реализовать. Первым делом необходимо сделать так, чтобы сервер базы данных запускался из супервизора, что позволит отделить ошибки базы данных от ошибок кеша. Сделать это довольно просто. Удалите вызов функции `Todo.Database.start_link` из `Todo.Cache.init/1` и добавьте еще одну спецификацию потомка при вызове `Supervisor.start_link/2`, как показано в следующем листинге.

Листинг 9.1 ❖ Запуск процесса базы данных из супервизора (supervise_database/lib/todo/system.ex)

```
defmodule Todo.System do
  def start_link do
    Supervisor.start_link(
      [
        Todo.Database,
        Todo.Cache
      ],
      strategy: :one_for_one
    )
  end
end
```

Добавление процесса базы данных в список спецификаций

Необходимо сделать еще кое-что. Как и в случае с `Todo.Cache`, перепишите функцию `Todo.Database.start_link` таким образом, чтобы она принимала один пустой аргумент. Так автоматически сгенерированная после выполнения выражения `use GenServer` функция `Todo.Database.child_spec/1` сможет вызывать ее со своими данными.

Листинг 9.2 ❖ Переработка функции `start_link` (supervise_database/lib/todo/system.ex)

```
defmodule Todo.Database do
  ...

  def start_link(_) do
    ...
  end

  ...
end
```

Эти правки позволят отделить кеш-процесс и сервер базы данных, как показано на рис. 9.2. Запуск двух этих процессов из супервизора обеспечит их независимый перезапуск. Ошибка в рабочем процессе базы данных приведет к отказу всего сервера базы данных, однако кеш-процесс это никак не затронет, и клиенты смогут пользоваться системой, пока сервер перезапускается.

Проверим, как работает наш код. Откройте папку supervise_database и выполните команду `iex -S mix`. Запустите систему:

```
iex(1)> Todo.System.start_link()
Starting database server.
Starting database worker.
Starting database worker.
Starting database worker.
Starting to-do cache.
```

Остановите сервер:

```
iex(2)> Process.exit(Process.whereis(Todo.Database), :kill)
Starting database server.
Starting database worker.
Starting database worker.
Starting database worker.
```

Как видите, перезапускаются только процессы, относящиеся к базе данных. Если остановить кеш-процесс, произойдут аналогичные действия. Помещая оба процесса под один супервизор, вы ограничиваете распространение негативных последствий ошибки. Ошибка в кеш-процессе никак не скажется на работоспособности базы данных, и наоборот.

В предыдущей главе мы говорили об изолировании процессов. Поскольку каждая из этих двух частей реализована в отдельном процессе, эти части изолированы и не влияют друг на друга. Конечно, эти два процесса неявно связаны между собой через супервизор, но супервизор перехватывает сигналы выхода, предотвращая распространение ошибки. Так, в частности, работают все супервизоры со стратегией `one_for_one` – они ограничивают влияние ошибки рамками одного рабочего процесса и принимают соответствующие меры (перезапуск) только в отношении этого процесса.

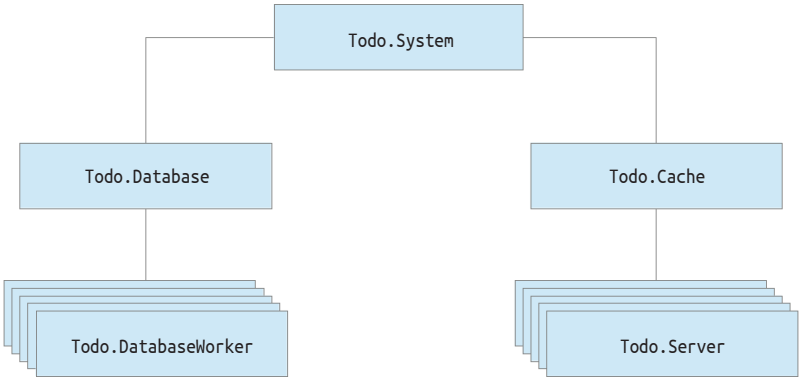


Рис. 9.2 ❖ Изолирование процессов базы данных и кеша

Дочерние процессы запускаются синхронно

В данном примере супервизор запускает два дочерних процесса. Важно понимать, что эти процессы запускаются синхронно в указанном порядке. Супервизор запускает один процесс, ожидает его полноценного запуска и переходит к следующему процессу. Если одним из дочерних процессов является `GenServer`, то следующий

дочерний процесс запускается только после того, как функция обратного вызова `init/1` завершит работу.

Как объяснялось в главе 7, функция `init/1` не должна выполняться слишком долго, и вот почему. Если бы процесс `Todo.Database` запускался, скажем, в течение пяти минут, все это время кеш-процесс был бы недоступен. Всегда следите за тем, чтобы функция `init/1` работала как можно быстрее, и используйте описанный в седьмой главе подход (отправка процессом самому себе во время инициализации) при более сложной инициализации.

9.1.2. Усовершенствованное обнаружение процессов

Итак, вы добавили изолирование ошибок на базовом уровне, но текущая реализация оставляет желать лучшего. Ошибка в одном рабочем процессе базы данных приведет к отказу всей базы данных и остановит все выполняющиеся с ней операции. В идеале ошибку следует изолировать в одном рабочем процессе, а значит, каждый рабочий процесс базы данных должен находиться под наблюдением супервизора.

Однако при использовании этого подхода возникает одна проблема. В текущей системе сервер базы данных запускает рабочие процессы и сохраняет их pid в список. Но если процесс будет запущен из супервизора, его pid будет неизвестен. Для супервизоров не характерно долгое хранение идентификаторов процессов, потому что всегда есть шанс, что тот или иной процесс будет перезапущен, и у его замены будет уже другой pid.

Поэтому каждому из наблюдаемых процессов необходимо дать символьное имя, и обращаться к нему следует тоже по этом имени. Когда такой процесс будет перезапущен, его новая версия будет зарегистрирована под этим же именем, что позволит обнаружить данный процесс даже после нескольких перезапусков.

Для этих целей вы можете использовать зарегистрированные имена. Однако они могут быть представлены только атомами, а в данном случае нужно что-то более сложное для детального описания именуемого процесса, например `{:database_worker, 1}`, `{:database_worker, 2}` и т. д. Необходим своеобразный реестр процессов, поддерживающий структуру ключ/значение (словарь), где ключами будут имена, а значениями – pid процессов. В отличие от стандартного механизма регистрации процессов, реестр позволит использовать более сложные имена.

Как только процесс будет создан, он может быть зарегистрирован в реестре под определенным именем. Если процесс аварийно завершился и был перезапущен, новый процесс будет зарегистрирован повторно. Использование реестра позволяет легко обнаружить процессы (их pid). Основная идея показана на рис. 9.3.

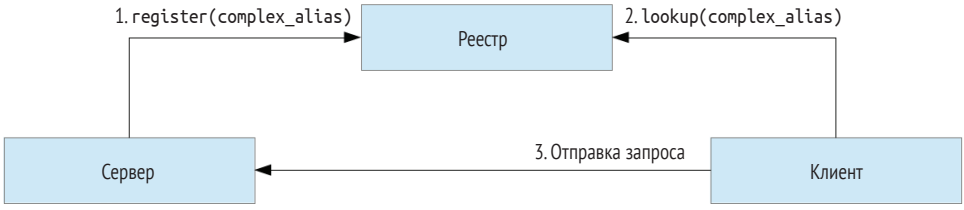


Рис. 9.3 ❖ Обнаружение процессов с помощью реестра

Первый шаг – регистрация рабочего процесса во время его инициализации. Немного позже клиентский процесс запросит у реестра pid требуемого рабочего процесса, после чего сможет отправить запрос серверному процессу.

Стандартная библиотека Elixir предоставляет реализацию *реестра процессов* в модуле *Registry*. С помощью этого модуля можно связать процесс с одним или несколькими ключами произвольной сложности и впоследствии получить pid этого процесса по этому ключу.

Рассмотрим пару примеров. Реестр процессов – это тоже процесс. Запустить его можно с помощью функции *Registry.start_link/1*:

```
iex(1)> Registry.start_link(name: :my_registry, keys: :unique)
```

Единственный аргумент данной функции – это ключевой список параметров регистрации. Обязательными параметрами являются `:name` и `:keys`.

Параметр `:name` представлен атомом и определяет имя процесса реестра, которое будет использовано для взаимодействия с ним.

Параметр `:keys` может быть либо `:unique`, либо `:duplicate`. В первом случае имена процессов уникальны, то есть под одним ключом может быть зарегистрирован только один процесс. Если вы хотите, чтобы процесс выполнял определенную роль, то это очень удобно. Например, только один процесс может соответствовать ключу `{:database_worker, 1}`. Второй вид реестра предполагает, что несколько процессов могут иметь одно и то же имя. Такой реестр может понадобиться, например, когда один процесс-издатель должен посылать уведомления неопределенному числу процессов-подписчиков, меняющемуся в течение времени.

Запустив процесс-реестр, вы можете зарегистрировать новый процесс под определенным ключом. Попробуем сделать это. Породим процесс `{:database_worker, 1}`, задачей которого будет ожидание сообщения и вывод его на консоль:

```
iex(2)> spawn(fn ->
          Registry.register(:my_registry, {:database_worker, 1}, nil)
          receive do
            msg -> IO.puts("got message #{inspect(msg)}")
          end
        end)
```

Регистрация процесса в реестре

Самая важная строка этого кода – вызов функции *Registry.register/3*. В нее передается имя реестра (`:my_registry`), имя, которое вы хотите дать порождаемому процессу (`{:database_worker, 1}`), и произвольное значение. После этого в реестре сохранится соответствие имени, указанного значения и pid вызывающего процесса.

Теперь другие процессы могут с легкостью обнаружить зарегистрированный процесс. В следующем примере pid рабочего процесса базы данных не указывается, потому что он больше не нужен. Его можно получить с помощью функции *Registry.lookup/2*:

```
iex(3)> [{db_worker_pid, _value}] =
          Registry.lookup(
            :my_registry,
            {:database_worker, 1}
          )
```

Функция `Registry.lookup/2` принимает на вход имя реестра и ключ (имя процесса), а возвращает список кортежей вида `{pid, value}`. Если реестр содержит уникальные ключи, то этот список либо пустой (ни один процесс не зарегистрирован под заданным именем), либо содержит всего один элемент. В противном случае список может содержать любое количество записей. Элемент `pid` каждого кортежа – это идентификатор зарегистрированного процесса, а `value` – значение, переданное функции `Registry.register/3`.

Обнаружив рабочий процесс базы данных, можно отправить ему сообщение:

```
iex(4)> send(db_worker_pid, :some_message)
got message :some_message
```

Очень интересным свойством реестра является то, что он связан со всеми зарегистрированными процессами, а значит, он сможет зафиксировать факт аварийного завершения этих процессов и удалить соответствующие им записи.

Проверим, так ли это на самом деле. Порожденный рабочий процесс является одноразовым: он получает сообщение, выводит его и затем завершается. Попробуем снова его обнаружить:

```
iex(5)> Registry.lookup(:my_registry, {:database_worker, 1})
[]
```

Как видите, по заданному ключу не найдено ни одной записи, что означает, что данного процесса больше не существует.

> **ПРИМЕЧАНИЕ** Стоит отметить, что модуль `Registry` полностью написан на Elixir. Можно сказать, что `Registry` – это своеобразный `GenServer`, хранящий соответствия между именами и pid в качестве своего состояния. Разумеется, в реальности реализация имеет гораздо более сложную структуру. Она основана на таблицах ETS, о которых вы узнаете из главы 10. Таблицы ETS позволяют `Registry` добиться эффективной работы и способности к масштабированию. Операции записи в реестр и поиска по нему выполняются очень быстро и в большинстве случаев не блокируют друг друга, благодаря чему несколько операций с одним и тем же реестром может выполняться параллельно.

Модуль `Registry` обладает другими свойствами и функциональными возможностями, которые не будут рассмотрены в этой книге. Вы можете найти их подробное описание в официальной документации по адресу: **https://hexdocs.pm/elixir/Registry.html**. Но все же далее мы рассмотрим одну из важнейших особенностей OTP-процессов под названием «*via-кортеж*».

9.1.3. Via-кортежи

Via-кортеж – это механизм, позволяющий использовать произвольный сторонний реестр для регистрации совместимых с OTP процессов, таких как `GenServer` и `Supervisor`. Вы должны помнить, что при запуске `GenServer` вы можете указать параметр `:name`:

```
GenServer.start_link(callback_module, some_arg, name: some_name)
```

До этого момента в параметре `:name` вы указывали только атомы, и запускаемый процесс локально регистрировался. Но параметр `:name` может быть представлен и в виде кортежа `{:via, some_module, some_arg}`, называемого via-кортежем.

Если в параметре :name указать via-кортеж, то для регистрации процесса GenServer вызовет функцию модуля some_module. Аналогичным образом можно передать via-кортеж первым аргументом функциям GenServer.cast и GenServer.call, и GenServer отыщет pid с помощью some_module. В этом смысле some_module выступает в качестве стороннего реестра процессов, а via-кортеж – способ подключения этого реестра к GenServer и подобным OTP-абстракциям.

Третий элемент via-кортежа (some_arg) – данные, передаваемые функциям модуля some_module. Конкретный формат этих данных должен быть определен в модуле реестра. Как минимум, они должны содержать имя, под которым процесс должен быть зарегистрирован и позже обнаружен.

В случае с модулем Registry третий аргумент должен содержать пару значений – {registry_name, process_key}, а via-кортеж целиком будет выглядеть так: {:via, Registry, {registry_name, process_key}}.

Рассмотрим пример. Вернемся к реализации EchoServer, приведенной в главе 6, – простому GenServer, обрабатывающему запрос типа call посредством возвращения содержимого запроса. Попробуем добавить регистрацию эхо-сервера. При запуске сервера укажем его ID – произвольный терм, идентифицирующий данный сервер уникальным образом. При отправке серверу запроса будем использовать этот ID вместо pid.

Реализация примет следующий вид:

```
defmodule EchoServer do
  use GenServer

  def start_link(id) do
    GenServer.start_link(__MODULE__, nil, name: via_tuple(id))
  end

  def call(id, some_request) do
    GenServer.call(via_tuple(id), some_request)
  end

  defp via_tuple(id) do
    {:via, Registry, {:my_registry, {__MODULE__, id}}}
  end

  def handle_call(some_request, _, state) do
    {:reply, some_request, state}
  end
end
```

Регистрация сервера с помощью via-кортежа

Обнаружение сервера с помощью via-кортежа

Via-кортеж, совместимый с реестром

Внешний вид via-кортежа определяется во вспомогательной функции via_tuple/1. Зарегистрированное имя будет представлено в формате {__MODULE__, id}, или {EchoServer, id} в данном случае.

Попробуем выполнить код. Запустите сеанс оболочки IEx, скопируйте определение модуля и запустите процесс реестра :my_registry:

```
iex(1)> defmodule EchoServer do ... end

iex(2)> Registry.start_link(name: :my_registry, keys: :unique)
```

Теперь вы можете запустить несколько эхо-серверов и работать с ними, не зная их pid:

```
iex(3)> EchoServer.start_link("server one")
iex(4)> EchoServer.start_link("server two")

iex(5)> EchoServer.call("server one", :some_request)
:some_request

iex(6)> EchoServer.call("server two", :another_request)
:another_request
```

Обратите внимание, что ID серверов представлены строками и что зарегистрированное имя имеет вид {EchoServer, some_id}. Это доказывает, что теперь для регистрации и обнаружения процессов используются сложные термы.

9.1.4. Регистрация рабочих процессов базы данных

Теперь, имея общее представление о возможностях Registry, вы можете реализовать регистрацию и обнаружение рабочих процессов базы данных. Во-первых, добавьте модуль Todo.ProcessRegistry, код которого приведен в листинге 9.3.

Листинг 9.3 ❖ Реестр процессов системы (pool_supervision/lib/todo/process_registry.ex)

```
defmodule Todo.ProcessRegistry do
  def start_link do
    Registry.start_link(keys: :unique, name: __MODULE__)
  end

  def via_tuple(key) do
    {:via, Registry, {__MODULE__, key}}
  end

  def child_spec(_) do
    Supervisor.child_spec(
      Registry,
      id: __MODULE__,
      start: {__MODULE__, :start_link, []}
    )
  end
end
```

Спецификация потомка

Функции интерфейса довольно просты. Функция start_link ссылается на модуль Registry для запуска реестра с уникальными ключами. Функция via_tuple/1 может быть использована другими модулями (например, Todo.DatabaseWorker) для создания подходящего via-кортежа, регистрирующего процесс в заданном реестре.

Поскольку реестр сам тоже является процессом, он должен находиться под наблюдением супервизора. Поэтому в модуль необходимо включить определение функции child_spec/1. В данном случае для изменения стандартной спецификации модуля Registry используется функция Supervisor.child_spec/2. По сути, этим вызовом вы обозначаете, что будет использована любая спецификация, предоставленная модулем Registry, с измененными полями :id и :start. Таким образом, подробности реализации Registry остаются в стороне, и вам не важно, является процесс реестра рабочим процессом или супервизором.

Теперь можно поручить запуск процесса реестра супервизору Todo.System. Реализация приведена в следующем листинге.

Листинг 9.4 ❖ Запуск реестра из супервизора (pool_supervision/lib/todo/system.ex)

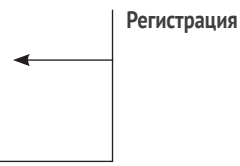
```
defmodule Todo.System do
  def start_link do
    Supervisor.start_link(
      [
        Todo.ProcessRegistry,  <-- Запуск процесса реестра
        Todo.Database,
        Todo.Cache
      ],
      strategy: :one_for_one
    )
  end
end
```

Не забывайте, что процессы в супервизоре запускаются синхронно в заданном вами порядке. Порядок следования спецификаций потомков должен быть строго определен: сначала указываются главные, а потом зависящие от них потомки. В данном случае реестр должен быть запущен первым, потому что рабочие процессы базы данных зависят от него.

Модуль `Todo.ProcessRegistry` готов, и теперь перейдем к доработке рабочих процессов базы данных. Необходимые изменения показаны в следующем листинге.

Листинг 9.5 ❖ Регистрация рабочих процессов (pool_supervision/lib/todo/database_worker.ex)

```
defmodule Todo.DatabaseWorker do
  use GenServer

  def start_link({db_folder, worker_id}) do
    IO.puts("Starting database worker #{worker_id}")

    GenServer.start_link(
      __MODULE__,
      db_folder,
      name: via_tuple(worker_id)    <-- Обнаружение
    )
  end

  def store(worker_id, key, data) do
    GenServer.cast(via_tuple(worker_id), {:store, key, data})
  end

  def get(worker_id, key) do
    GenServer.call(via_tuple(worker_id), {:get, key})
  end

  defp via_tuple(worker_id) do
    Todo.ProcessRegistry.via_tuple({__MODULE__, worker_id})
  end

  ...
end
```



В данном коде введено понятие идентификатора рабочего процесса (`worker_id`), находящегося в пределах от 1 до размера пула (`pool_size`). Функция `start_link`

теперь принимает этот параметр в качестве аргумента наряду с db_folder в виде одного кортежа {db_folder, worker_id}. Это опять же делается для согласования с автоматически генерируемой функцией child_spec/1, вызывающей функцию start_link/1. Для управления рабочим процессом в супервизоре теперь можно использовать спецификацию потомка {Todo.DatabaseWorker, {db_folder, worker_id}}.

При вызове функции GenServer.start_link в качестве параметра имени указывается via-кортеж. Сам кортеж оборачивается во внутреннюю функцию via_tuple/1, принимающую на вход ID рабочего процесса и возвращающую соответствующий via-кортеж. Эта функция делегирует модулю Todo.ProcessRegistry, передавая ему будущее имя в виде кортежа {__MODULE__, worker_id}, после чего рабочий процесс регистрируется под ключом {Todo.DatabaseWorker, worker_id}. Такой формат имени позволяет избежать конфликтов с другими процессами, зарегистрированными в том же реестре.

Вспомогательная функция via_tuple/1 точно так же используется для обнаружения процессов при вызовах GenServer.call и GenServer.cast. Функциям store/3 и get/2 теперь передается ID рабочего процесса в качестве первого аргумента, а значит, клиентам, использующим эти функции, больше нет необходимости в отслеживании pid процессов.

9.1.5. Наблюдение за рабочими процессами

Для управления пулом рабочих процессов необходимо создать отдельный супервизор. Теоретически все рабочие процессы можно без проблем запускать и из супервизора Todo.System. Однако, как уже говорилось в предыдущей главе, если процессы будут перезапускаться слишком часто, супервизор остановится и завершит все дочерние процессы. Чем больше у супервизора потомков, тем выше шанс достижения максимального значения частоты перезапусков, и всего один проблемный процесс может привести к останову всей системы.

С учетом вышесказанного было принято решение поместить часть системы (базу данных) под отдельный супервизор. Если какой-либо из рабочих процессов базы данных не удастся запустить, новый супервизор завершится, и супервизор-родитель попытается перезапустить только относящиеся к базе данных процессы, не затрагивая остальные процессы системы.

Так или иначе, вам больше не понадобится GenServer базы данных. В его задачи входили запуск пула рабочих процессов и приведение в соответствие ID рабочих процессов и их pid. Теперь рабочие процессы будут запускаться супервизором, а соответствия будут храниться в реестре.

Модуль Todo.Database можно оставить и реализовать в нем супервизор рабочих процессов базы данных и те же функции интерфейса, что и прежде. Благодаря этому код клиентского модуля Todo.Server переписывать не придется, а Todo.Database можно будет оставить в списке потомков супервизора Todo.System.

Давайте проделаем это. Готовый код обновленного модуля Todo.Database приведен в листинге 9.6.

Листинг 9.6 ❖ Супервизор рабочих процессов (pool_supervision/lib/todo/database.ex)

```
defmodule Todo.Database do
  @pool_size 3
  @db_folder "./persist"
```

```
  def start_link do
    File.mkdir_p!(@db_folder)

    children = Enum.map(1..@pool_size, &worker_spec/1)
    Supervisor.start_link(children, strategy: :one_for_one)
  end

  defp worker_spec(worker_id) do
    default_worker_spec = {Todo.DatabaseWorker, {@db_folder, worker_id}}
    Supervisor.child_spec(default_worker_spec, id: worker_id)
  end

  ...
end
```


В данном коде сначала создается список из трех спецификаций потомков, каждая из которых описывает рабочий процесс. Затем этот список передается функции `Supervisor.start_link/2`.

Спецификация для каждого потомка создается в функции `worker_spec/1`. Берется стандартная спецификация рабочего процесса базы данных в виде `{Todo.DatabaseWorker, {@db_folder, worker_id}}`, и в нее добавляется уникальный идентификатор рабочего процесса.

Если этого не сделать, то несколько дочерних процессов будут иметь один и тот же ID. Как вам известно, стандартная функция `child_spec/1`, генерируемая после выполнения выражения `use GenServer`, предоставляет имя модуля в поле `:id`. Соответственно, если использовать стандартную спецификацию и попробовать запустить два рабочих процесса, они получат одинаковые ID `Todo.DatabaseWorker`. Модулю `Supervisor` это не понравится, и он возбудит ошибку.

Необходимо также реализовать функцию `Todo.Database.child_spec/1`, как показано в листинге 9.7. Модуль базы данных теперь является супервизором и не содержит строки `use GenServer`, а значит, функция `child_spec/1` больше не генерируется автоматически.

Листинг 9.7 ❖ Операции с базой данных (pool_supervision/lib/todo/database.ex)

```
defmodule Todo.Database do
  ...

  def child_spec(_) do
    %{
      id: __MODULE__,
      start: {__MODULE__, :start_link, []},
      type: :supervisor
    }
  end

  ...
end
```

Как можно видеть, спецификация содержит новое поле `:type`. Оно используется для обозначения типа запускаемого процесса. Допустимыми значениями этого поля могут быть атомы `:supervisor` (если дочерний процесс является супервизором) или `:worker` (для любых других процессов). Если это поле не указать, то по умолчанию будет использовано значение `:worker`.

В листинге 9.7 функция child_spec/1 определяет, что Todo.Database является супервизором и может быть запущен функцией Todo.Database.start_link/0.

Это был отличный пример того, как child_spec/1 позволяет не выносить подробности реализации за пределы модуля процесса. Вы только что превратили процесс базы данных в супервизор и изменили арность его функции start_link (теперь она равна 0), однако в модуле Todo.System менять ничего не требуется.

Теперь немного перепишем функции store/2 и get/1.

Листинг 9.8 ❖ Операции с базой данных (pool_supervision/lib/todo/database.ex)

```
defmodule Todo.Database do
  ...

  def store(key, data) do
    key
    |> choose_worker()
    |> Todo.DatabaseWorker.store(key, data)
  end

  def get(key) do
    key
    |> choose_worker()
    |> Todo.DatabaseWorker.get(key)
  end

  defp choose_worker(key) do
    :erlang.phash2(key, @pool_size) + 1
  end

  ...
end
```

Данная реализация отличается от предыдущей только функцией choose_worker/1. Ранее эта функция отправляла синхронный запрос к серверному процессу базы данных. Теперь же она выбирает ID рабочего процесса из диапазона [1..@pool_size]. Этот ID затем передается функциям Todo.DatabaseWorker, выполняющим поиск по реестру и передающим запрос соответствующему рабочему процессу базы данных.

Настало время проверить, как все это работает. Запустите систему:

```
iex(1)> Todo.System.start_link()
Starting database worker 1
Starting database worker 2
Starting database worker 3
Starting to-do cache.
```

Убедимся в том, что отдельные рабочие процессы перезапускаются корректно. Для этого нужно узнать их pid. Так как вы хорошо знакомы с внутренним устройством системы, их можно легко найти в реестре. Имея pid рабочего процесса, его можно остановить вручную:

```
iex(2)> [{worker_pid, _}] =
          Registry.lookup(
            Todo.ProcessRegistry,
            {Todo.DatabaseWorker, 2}
          )
```

```
iex(3)> Process.exit(worker_pid, :kill)
Starting database worker 2
```

Как и ожидалось, рабочий процесс перезапущен, и остальные составляющие системы не затронуты.

Стоит еще раз поговорить о том, как реестр обеспечивает правильное поведение в системе относительно перезапущенных процессов. После перезапуска рабочий процесс получит новый pid. Благодаря наличию реестра для клиента этот факт неважен. Получение идентификатора происходит в самый последний момент с помощью операции поиска по регистру перед передачей запроса рабочему процессу базы данных. В большинстве ситуаций поиск будет успешен, и запрос будет направлен соответствующему процессу.

В некоторых случаях, например когда рабочий процесс завершился после получения клиентом его pid, но перед отправкой запроса, рабочий процесс не может быть обнаружен. Тогда клиенту будет предоставлен устаревший pid, и запрос не удастся. То же самое актуально, если клиент пытается обнаружить только что отказавший рабочий процесс. Перезапуск и регистрация процессов выполняются конкурентно относительно действий клиента, поэтому pid запрашиваемого процесса может отсутствовать в реестре на момент поиска.

Результат обоих сценариев один: клиентский процесс (сервер списка дел) аварийно завершится, и ошибка будет передана конечному пользователю. Такова сама сущность систем с высокой степенью конкурентности. Восстановление системы после ошибки выполняется конкурентно в процессе супервизора, и отдельные части системы могут на короткий срок потерять согласованность.

9.1.6. Построение дерева супервизоров

Остановимся на минуточку и посмотрим, что мы имеем на данный момент. На рис. 9.4 показаны связи между процессами в текущей системе.

Вы можете наблюдать пример простого дерева супервизоров – вложенной структуры супервизоров и рабочих процессов. Оно описывает организацию системы в иерархию сервисов. В данном случае система состоит из трех сервисов: реестр процессов, база данных и кеш.

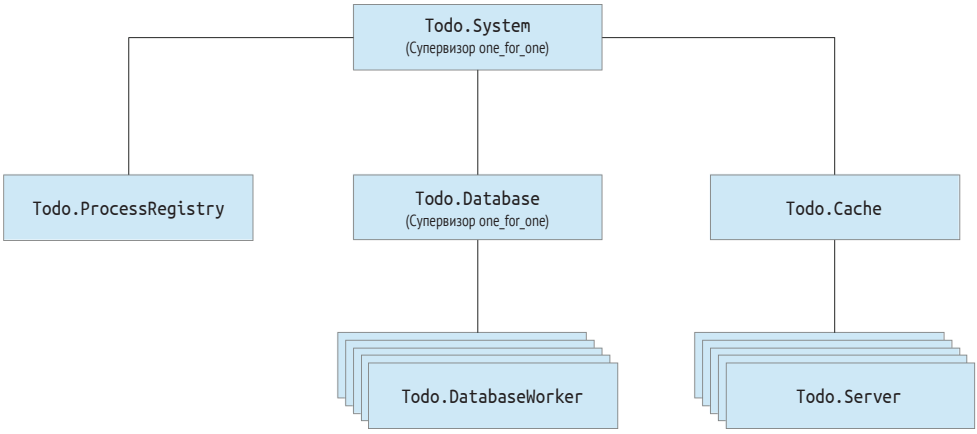


Рис. 9.4 ❖ Дерево супервизоров

Каждый сервис может, в свою очередь, состоять из более мелких сервисов. Например, база данных строится на основе нескольких рабочих процессов, а кеш – на основе серверных процессов. Даже реестр разделяется на некоторое количество процессов, но это уже особенности реализации модуля `Registry`, и на рисунке они опущены.

Несмотря на то что супервизоры часто упоминаются в контексте отказоустойчивости и восстановления системы после ошибок, их главной задачей является определение соответствующего порядка запуска процессов системы. Дерево супервизоров показывает, как система должна быть запущена и остановлена.

Более мелкоструктурное дерево позволяет отключить определенную часть системы незаметно для остальных ее составляющих. Чтобы остановить процесс базы данных в текущей версии системы, нужно всего лишь попросить его предка (`Todo.System`) завершить дочерний процесс `Todo.Database` с помощью функции `Supervisor.terminate_child/2`. Процесс базы данных и все его потомки прекратят работу.

Раз уж рабочие процессы – это мелкие сервисы системы, то супервизоры можно назвать своеобразными менеджерами сервисов вроде systemd и Windows Service Manager. Они не только управляют сервисами, но и отвечают за их жизненный цикл. Если какой-либо важный сервис откажет, предок попытается перезапустить его.

Взглянув на дерево супервизоров, можно представить, как в системе распространяются и обрабатываются ошибки. Если рабочий процесс базы данных откажет, супервизор базы данных перезапустит его, не затрагивая остальные части системы. А если это не поможет, максимальное значение частоты перезапусков будет превышено, и супервизор остановит все рабочие процессы и остановится сам.

Главный супервизор системы обязательно это заметит и запустит пул новых рабочих процессов в надежде решить проблему. Что это дает? Перезапуская целую группу рабочих процессов, вы эффективно завершаете все текущие операции с базой данных и начинаете с чистого листа. Если это не помогает, то больше ничего сделать нельзя, и ошибка передается выше по дереву (в данном случае вся система останавливается). Именно так происходит восстановление системы в дереве супервизоров – сначала это делается локально, затрагивая при этом как можно меньше процессов. Если это не сработало, то управление передается выше, и перезапускается более объемная часть системы.

Совместимость процессов с OTP

Все процессы, запускаемые из супервизора, должны быть *совместимы с OTP*. Чтобы реализовать такой процесс, недостаточно просто породить процесс или добавить с ним связь. Необходимо также обрабатывать характерные для OTP сообщения особым способом. Более подробную информацию об этом можно найти в разделе «sys and proc_lib» официальной документации Erlang по адресу: **http://erlang.org/doc/design_principles/spec_proc.html#id80464**.

К счастью, в большинстве случаев вам не придется писать совместимые с OTP процессы с нуля. Вы можете использовать готовые высокоуровневые абстракции вроде `GenServer`, `Supervisor` и `Registry`. Запущенные с помощью этих модулей процессы всегда будут совместимы с OTP. Elixir также предоставляет модули `Task` и `Agent`, которые можно использовать для запуска совместимых с OTP процессов. Эти модули будут подробно рассмотрены в следующей главе.

Обычные процессы, порождаемые функцией `spawn_link`, не совместимы с OTP, и их не стоит запускать из супервизора. Вы можете без проблем запускать обычные процессы из рабочих процессов вроде `GenServer`, но все же более правильным решением будет по максимуму использовать совместимые с OTP процессы. При аварийном завершении таких процессов в журнал будет занесена более подробная информация об ошибке.

Мягкое завершение процессов

Важным преимуществом деревьев супервизоров является возможность останова всей системы и полного завершения всех процессов. При завершении супервизора завершаются и все его потомки первого уровня. Если все остальные процессы системы прямо или косвенно связаны с этими потомками, они тоже неизбежно будут завершены. Соответственно, остановив главный процесс-супервизор, можно остановить всю систему.

Чаще всего для завершения поддерева супервизоров после ошибки используется *мягкий выход*. Процесс-супервизор останавливает своих потомков должным образом, предоставляя им возможность очистить за собой память. Если какие-то из этих потомков сами являются супервизорами, они точно так же завершат свои деревья. Мягкий выход с освобождением занимаемой памяти при использовании рабочих процессов `GenServer` можно осуществить с помощью функции обратного вызова `terminate/2`, но только если в рабочем процессе в `init/1` реализован перехват выходов.

Поскольку мягкий выход предполагает возможное задействование кода, отвечающего за очистку памяти, он может занять больше времени, чем вам бы хотелось. Параметр `:shutdown` в спецификации потомков позволяет задать максимальное значение времени, в течение которого супервизор будет ожидать мягкого завершения потомка. Если он не успеет сделать все необходимое за отведенное время, то будет завершен принудительно. Чтобы установить время ожидания, укажите параметр `shutdown: shutdown_strategy` в функции `child_spec/1` и передайте целое число, соответствующее времени в миллисекундах. Как вариант вы можете передать атом `:infinity`, и супервизор будет ожидать полного завершения потомка столько, сколько потребуется. Вы также можете использовать атом `:brutal_kill`, и тогда супервизор незамедлительно остановит дочерний процесс. Такое же принудительное завершение выполняется и при передаче процессу сигнала выхода `:kill`, как вы уже делали ранее в вызовах `Process.exit(pid, :kill)`.

Стандартное значение параметра `:shutdown` равно 5000 для рабочих процессов и `:infinity` для супервизоров.

Предотвращение перезапуска процесса

По умолчанию супервизор перезапускает упавший процесс независимо от причины выхода. Даже если процесс завершится с причиной `:normal`, он все равно будет перезапущен, и это не всегда уместно.

Для примера рассмотрим процесс, обрабатывающий HTTP-запрос или подключение по TCP. Если такой процесс откажет, то сокет закроется, и перезапускать этот процесс будет незачем (связь с удаленным сервером будет утеряна). Подобные процессы все же стоит оставить под наблюдением супервизора, чтобы при необходимости они завершались вместе со всей остальной веткой супервизора. В таком случае вы можете создать *временный* рабочий процесс, указав в функции

child_spec/1 параметр restart: :temporary. Временные рабочие процессы не перезапускаются после завершения.

Еще один вариант – использование *переходных* рабочих процессов, перезапускаемых только после непредвиденного завершения. Процессы, которые по плану должны рано или поздно завершиться, можно сделать переходными. Например, если вам необходимо выполнить какую-то разовую задачу при запуске системы, вы можете запустить соответствующий процесс (на основе модуля Task) в дереве супервизоров и обозначить его как переходный, включив опцию restart: :transient в child_spec/1.

Стратегии перезапуска

До этого момента во всех примерах использовалась только стратегия :one_for_one, при которой супервизор перезапускает лишь отказавший дочерний процесс, не затрагивая других своих потомков. Существуют также и иные стратегии перезапуска:

- :one_for_all – завершение всех потомков при отказе одного из них и последующий их перезапуск;
- :rest_for_one – завершение всех потомков отказавшего процесса и их перезапуск.

Использовать эти стратегии имеет смысл, если между родственными процессами имеется тесная связь и один процесс не может выполнить свои задачи без помощи других процессов. Например, если процесс хранит pid родственного процесса в своем внутреннем состоянии, то эти два процесса связаны между собой, и при завершении одного из них другой тоже должен быть завершен.

Стратегии :one_for_all и :rest_for_one очень пригодятся в таких случаях. Первая используется, если все родственные процессы взаимосвязаны, а вторая – если дочерние процессы зависят от родительских.

Например, в текущей системе можно было бы использовать стратегию :rest_for_one, чтобы рабочие процессы базы данных завершались при отказе процесса реестра. Без реестра эти процессы не смогут выполнять поставленные задачи, и вполне разумным будет избавиться от них. Однако в нашем случае это не требуется, поскольку модуль Registry связывает каждый зарегистрированный процесс с процессом реестра, и в результате завершение процесса реестра приводит к завершению всех зарегистрированных процессов. Те из них, которые перехватывают сигналы выхода, получат уведомление, а остальные будут автоматически остановлены.

На этом мы закончим разговор об изоляции ошибок на уровне отдельных процессов с помощью супервизоров. Вы внесли в код системы некоторое количество изменений, позволяющих минимизировать последствия ошибок, но он все еще далек от идеала. Мы продолжим работать над ним в следующем разделе, посвященном динамическому запуску рабочих процессов.

9.2. Динамический запуск рабочих процессов

Благодаря изменениям, проделанным в предыдущей главе, ошибки в рабочих процессах базы данных не выходят за пределы одного рабочего процесса. Теперь нам предстоит сделать то же самое для серверных процессов. Грубо говоря, будем

использовать тот же подход: реализуем запуск каждого серверного процесса из супервизора и зарегистрируем их в реестре.

9.2.1. Регистрация серверных процессов

Начнем с регистрации серверных процессов – внесите одну небольшую правку, как показано в следующем листинге.

Листинг 9.9 ❖ Регистрация серверных процессов (dynamic_workers/lib/todo/server.ex)

```
defmodule Todo.Server do
  use GenServer, restart: :temporary

  def start_link(name) do
    IO.puts("Starting to-do server for #{name}")
    GenServer.start_link(Todo.Server, name, name: via_tuple(name))   ← Регистрация сервера
  end

  defp via_tuple(name) do
    Todo.ProcessRegistry.via_tuple({__MODULE__, name})
  end

  ...
end
```

Как видите, здесь используется тот же подход, что и с рабочими процессами базы данных: в опции имени передается via-кортеж. Он четко обозначает, что серверный процесс должен быть зарегистрирован в реестре под ключом {__MODULE__, name}. Более сложный формат ключа позволяет избежать возможных конфликтов с ключами других серверных и рабочих процессов.

Функции add_entry/2 и entries/2 остаются без изменений, по-прежнему принимают на вход pid и используются так же, как и раньше. Клиентский процесс сначала получает pid сервера с помощью Todo.Cache.server_process/1, после чего вызывает функции модуля Todo.Server.

9.2.2. Динамические супервизоры

Теперь необходимо реализовать запуск серверных процессов из супервизора. Вот тут есть небольшая загвоздка. В отличие от рабочих процессов, серверные процессы создаются динамически, когда в них появляется необходимость. Изначально в системе нет ни одного серверного процесса; каждый из них создается по запросу вызовом Todo.Cache.server_process/1. Это означает, что указать точное количество потомков в супервизоре не удастся, потому что их число заранее неизвестно.

Для таких случаев необходим *динамический супервизор*, который бы запускал дочерние процессы по запросу. В Elixir за это отвечает модуль *DynamicSupervisor*. Он похож на Supervisor, разве что при запуске динамического супервизора список потомков не указывается, и в итоге запускается только процесс супервизора. В любой момент после этого вы можете породить дочерний процесс с помощью функции DynamicSupervisor.start_child/2.

Рассмотрим это на примере. Превратим процесс Todo.Cache в динамический супервизор, как показано в следующем листинге.

Листинг 9.10 ❖ Превращение кеш-процесса в супервизор (dynamic_workers/lib/todo/cache.ex)

```
defmodule Todo.Cache do
  def start_link() do
    IO.puts("Starting to-do cache")

    DynamicSupervisor.start_link(                    | Запуск динамического супервизора
      name: __MODULE__,
      strategy: :one_for_one
    )
  end

  ...
end
```

Можно видеть, что супервизор запускается с помощью функции `DynamicSupervisor.start_link/1`. Она запустит только процесс супервизора, поскольку потомки на этом этапе не указываются. Обратите внимание, что при запуске супервизора также передается параметр `:name`, вследствие чего супервизор будет зарегистрирован под локальным именем.

Благодаря локальной регистрации взаимодействие с супервизором и запуск его потомков заметно упрощаются. Добавьте определение функции `start_child/1`, которая будет запускать серверный процесс для заданного списка дел:

```
defmodule Todo.Cache do
  ...

  defp start_child(todo_list_name) do
    DynamicSupervisor.start_child(
      __MODULE__,
      {Todo.Server, todo_list_name}
    )
  end

  ...
```

В данной реализации вызывается функция `DynamicSupervisor.start_child/2`, и ей передается имя супервизора и спецификация потомка, которого она должна запустить. Спецификация `{Todo.Server, todo_list_name}` вызовет `Todo.Server.start_link(todo_list_name)`, и серверный процесс будет запущен как потомок супервизора `Todo.Cache`.

Стоит заметить, что `DynamicSupervisor.start_child/2` – это синхронный вызов. Процессу супервизора отправляется запрос, и он запускает дочерний процесс. Если несколько клиентов одновременно захотят запустить потомка одного и того же супервизора, запросы будут выполняться по очереди.

Более подробную информацию о динамических супервизорах можно найти в официальной документации по адресу: **https://hexdocs.pm/elixir/DynamicSupervisor.html**.

Осталось лишь добавить реализацию функции `child_spec/1`:

```
defmodule Todo.Cache do
  ...

  def child_spec(_arg) do
```

```
    %{
      id: __MODULE__,
      start: {__MODULE__, :start_link, []},
      type: :supervisor
    }
  end

  ...
end
```

Вот и все, теперь кеш-процесс является динамическим супервизором.

9.2.3. Обнаружение серверных процессов

Последнее, что нужно исправить на данном этапе, – это реализация функции обнаружения `Todo.Cache.server_process/1`. Она принимает имя серверного процесса, возвращает его pid и запускает его, если он еще не запущен. Обновленная реализация приведена в следующем листинге.

Листинг 9.11 ❖ Обнаружение серверного процесса (dynamic_workers/lib/todo/cache.ex)

```
defmodule Todo.Cache do
  ...

  def server_process(todo_list_name) do
    case start_child(todo_list_name) do
      {:ok, pid} -> pid          <-- Запуск нового сервера
      {:error, {:already_started, pid}} -> pid   <-- Сервер уже запущен
    end
  end

  defp start_child(todo_list_name) do
    DynamicSupervisor.start_child(
      __MODULE__,
      {Todo.Server, todo_list_name}
    )
  end
end
```

Данная функция сначала вызывает локальную функцию `start_child/1`, которую вы реализовали в предыдущем разделе и которая является оберткой над функцией `DynamicSupervisor.start_child/2`.

Этот вызов имеет два возможных положительных результата. Наиболее очевидный из них – кортеж `{:ok, pid}`, где pid – это идентификатор только что запущенного серверного процесса.

Второй результат, `{:error, {:already_started, pid}}`, гораздо интереснее. Он означает, что серверный процесс зарегистрировать не удалось, потому что с таким именем уже запущен. В нашем случае подобный результат также является успешным: вы пытаетесь запустить сервер, но он и так уже работает. Это нормально. У вас есть его pid, и можно без проблем посылать ему запросы.

Результат `{:error, {:already_started, pid}}` возвращается в таком виде из-за внутреннего устройства механизма регистрации `GenServer`. Если функции `GenServer.start_link` передается параметр `:name`, регистрация выполняется в запущенном

процессе перед вызовом функции `init/1`. Регистрация не удастся, если под таким же ключом уже зарегистрирован другой процесс. В данном случае функция `GenServer.start_link` вместо запуска цикла сервера возвращает результат `{:error, {:already_started, pid}}`, в котором pid идентифицирует процесс, зарегистрированный под таким же ключом. Этот же результат затем возвращает и функция `DynamicSupervisor.start_child`.

Стоит кратко проанализировать, как `server_process/1` поведет себя в конкурентном сценарии. Допустим, два процесса попытаются одновременно вызвать эту функцию. `Server_process/1` передаст управление функции `DynamicSupervisor.start_child/2`, и тогда вы получите два одновременных запроса `start_child` к одному и тому же супервизору. Как вы помните, дочерние процессы запускаются в процессе супервизора, а потому вызовы `start_child` будут упорядочены, и состояния гонки не возникнет.

Если посмотреть на это под другим углом, такой способ использования `start_child` далеко не эффективен. Каждый раз, когда вам необходимо проделать какие-либо действия со списком, вы отправляете запрос одному и тому же супервизору, и он легко может стать узким местом системы. Даже если сервер уже запущен, супервизор запустит нового потомка и сразу же его остановит. Все это можно легко исправить, но пока это не требуется. Мы вернемся к этой проблеме в главе 12, когда будем говорить о распределенной регистрации.

9.2.4. Использование временных рабочих процессов

Осталось еще кое-что. Сделаем серверные процессы временными, чтобы они не перезапускались после аварийного завершения.

Зачем это нужно? Серверные процессы запускаются по запросу. Когда пользователь пытается выполнить операцию со списком, запускается серверный процесс, если его еще не существует. Если он упадет, то при следующем запросе пользователя он будет снова запущен, поэтому необходимости в его перезапуске просто нет.

Выбор стратегии `:temporary` также означает, что супервизор не будет перезапускаться из-за многочисленных отказов его потомков. Даже если один серверный процесс будет постоянно останавливаться, скажем, из-за поврежденной структуры его состояния, кеш-процесс будет продолжать работать, что повышает доступность системы в целом.

Стратегию перезапуска можно легко изменить, указав параметр `:restart` в строке `use GenServer`:

```
defmodule Todo.Server do
  use GenServer, restart: :temporary

  ...
end
```

Значение `:temporary` будет автоматически подставлено после ключа `:restart` в результате функции `child_spec/1`, и родительский супервизор считает эти потомки временными. Если такой потомок завершится, супервизор не станет его перезапускать.

Здесь вы можете задуматься, для чего тогда вообще помещать таких потомков под наблюдение супервизора. На то есть две весомые причины. Во-первых, при

использовании такой структуры отказ одного серверного процесса не влияет на остальные процессы системы. Во-вторых, как объяснялось в разделе 9.1.6, это позволяет завершить всю систему должным образом и очистить занимаемую память. Чтобы остановить все серверные процессы, нужно всего лишь остановить супервизор `Todo.Cache`. Другими словами, в функции супервизоров входит не только перезапуск упавших процессов, но и ограничение влияния отказов отдельных частей на всю систему и осуществление корректного завершения всех процессов.

9.2.5. Тестирование системы

Вы сделали так, чтобы серверные процессы отслеживались супервизором, и теперь самое время для небольшого теста. Заметьте, супервизор `Todo.System` остался без изменений. `Todo.Cache` уже был указан в списке его потомков, и вы лишь поменяли его реализацию. Давайте проверим, как это работает.

Запустите всю систему в оболочке:

```
iex(1)> Todo.System.start_link()
Starting database worker 1
Starting database worker 2
Starting database worker 3
Starting to-do cache
```

Создайте один серверный процесс:

```
iex(2)> bobs_list = Todo.Cache.server_process("Bob's list")
Starting to-do server for Bob's list
#PID<0.118.0>
```

Повторите запрос и убедитесь, что он не запускает еще один сервер:

```
iex(3)> bobs_list = Todo.Cache.server_process("Bob's list")
#PID<0.118.0>
```

Повторите запрос для другого списка и проверьте, создается ли новый серверный процесс:

```
iex(4)> alices_list = Todo.Cache.server_process("Alice's list")
Starting to-do server for Alice's list
#PID<0.121.0>
```

Остановите один из серверов:

```
iex(5)> Process.exit(bobs_list, :kill)
```

Убедитесь, что соответствующий вызов `Todo.Cache.server_process/1` возвращает новый pid:

```
iex(6)> Todo.Cache.server_process("Bob's list")
Starting to-do server for Bob's list
#PID<0.124.0>
```

Серверный процесс для списка Alice должен остаться прежним:

```
iex(7)> Todo.Cache.server_process("Alice's list")
#PID<0.121.0>
```

Дерево супервизоров для данной реализации представлено на рис. 9.5. На рисунке показано, как отслеживание каждого процесса позволяет ограничить распространение непредвиденных ошибок.

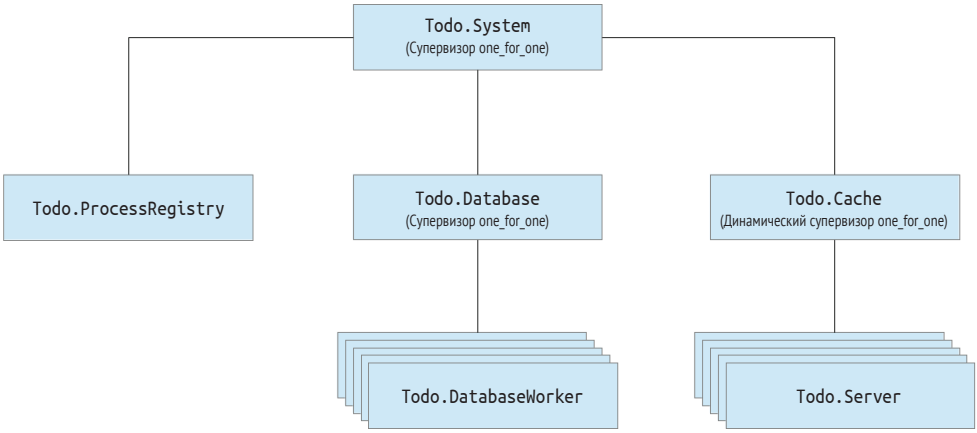


Рис. 9.5 ❖ Отслеживание процессов системы

Теперь можно сказать, что ваша система отказоустойчива. Вы добавили в нее несколько дополнительных супервизоров, а также упростили кое-какие ее части (избавились от кеша и серверных процессов базы данных). Впереди еще много работы, но пока ненадолго переключимся на изучение некоторых практических рекомендаций.

9.3. Let it crash

Обычно при разработке сложных систем для обработки ошибок и восстановления системы используются супервизоры. Если дерево супервизоров построено правильно, то последствия непредвиденных ошибок будут ограничены, и система сможет восстанавливаться после них. В моей практике были случаи, когда супервизоры помогли справиться с нестандартными проблемами во время промышленной эксплуатации, сохраняя систему в стабильном рабочем состоянии, а меня – в стабильном психическом. Также стоит заметить, что благодаря OTP при каждом отказе в журнал добавляется соответствующая запись, и вы можете даже добавить обработчик события, который будет срабатывать при каждом отказе и выполнять определенные вами действия, такие как отправка сообщения на электронную почту или передача отчета внешней системе.

Важнейшим следствием такого подхода к обработке ошибок является отсутствие в коде надоедливых защитных конструкций `try/catch`. Как правило, они попросту не нужны, так как супервизоры полностью берут на себя восстановление системы после ошибок. Один из создателей Erlang, Джо Армстронг (Joe Armstrong), в своей докторской диссертации («Making reliable distributed systems in the presence of software errors», **http://erlang.org/download/armstrong_thesis_2003.pdf**) назвал этот стиль *интенциональным программированием*. При использовании этого стиля код

не захламлен многочисленными защитными конструкциями и четко демонстрирует намерения разработчика.

Этот стиль также принято называть *Let it crash*. Он не только делает код более чистым и собранным, но и позволяет системе после восстановления начинать «с чистого листа». Как вы уже знаете, новый процесс всегда запускается с новым согласованным состоянием, а очередь сообщений (почтовый ящик) старого процесса уничтожается. В результате этого некоторые запросы не смогут быть выполнены, но зато новые процессы начинают с нуля, что дает им больше шансов выполнить работу должным образом.

Принцип Let it crash не означает игнорирование абсолютно всех возникающих в системе ошибок. Есть две ситуации, в которых ошибку требуется обрабатывать вручную:

- основные процессы, отказа которых допускать нельзя;
- ожидаемые ошибки, которые можно эффективно обработать.

Разберем каждый из этих случаев более подробно.

9.3.1. Процессы, отказа которых допускать нельзя

Неофициально такие процессы называются *ядром ошибок* системы. Это процессы, обеспечивающие основную функциональность системы, состояние которых не так просто восстановить и привести в соответствие. Эти процессы – сердце любой системы, и если они завершатся, система не сможет выполнить ни одной задачи.

Код основных процессов следует делать как можно проще: чем меньше в нем выполняется логики, тем меньше шанс возникновения ошибки. Если вы чувствуете, что код получился слишком длинным, попробуйте разделить его на два процесса – одному поручите хранение состояния, а другому – выполнение необходимых операций. В этом случае первый процесс будет совсем простым и вряд ли когда-либо откажет, а второй, рабочий процесс уже не будет относиться к ядру ошибок, поскольку он больше не хранит жизненно важное состояние.

Для усиления основных процессов и предотвращения их аварийного завершения можно добавить защитные конструкции `try/catch` в каждую функцию обратного вызова `handle_*`. Это выглядит примерно следующим образом:

```
def handle_call(message, _, state) do
  try
    new_state =
      state
      |> transformation_1()
      |> transformation_2()
      ...

    {:reply, response, new_state}
  catch _, _ ->
    {:reply, {:error, reason}, state}
  end
end
```

Отлов всех ошибок и использование исходного состояния

Данный пример показывает, как неизменяемые структуры данных позволяют реализовать отказоустойчивый сервер. Пока запрос обрабатывается, над состоянием выполняется ряд преобразований. Если что-то пошло не так, используется

исходное состояние, и происходит откат всех изменений. Это обеспечивает согласованность состояния и постоянно поддерживает процесс на плаву.

Имейте в виду, что данный подход не гарантирует полной защиты от аварий. Например, процесс может быть уничтожен вызовом `Process.exit(pid, :kill)`, так как причина выхода `:kill` не распознается при перехвате выходов. Поэтому стоит продумать план восстановления системы при отказе основного процесса. Создайте дерево супервизоров и позаботьтесь о том, чтобы все связанные с отказавшим процессы тоже завершались.

9.3.2. Обработка ожидаемых ошибок

Основная идея стратегии Let it crash в том, чтобы поручить восстановление системы после непредвиденных ошибок супервизорам. Но если вы можете заранее предвидеть ошибку и знаете, как с ней стоит разобраться, нет смысла допускать аварийного завершения процесса из-за нее.

Рассмотрим небольшой пример запроса `:get` к рабочему процессу базы данных:

```
def handle_call({:get, key}, _, db_folder) do
  data =
    case File.read(file_name(db_folder, key)) do
      {:ok, contents} -> :erlang.binary_to_term(contents)
      _ -> nil    <-------- Обработка ошибки чтения файла
    end

  {:reply, data, db_folder}
end
```


При обработке get-запроса происходит чтение данных из файла и покрывается случай, если файл прочитать не удастся. Тогда возвратится значение `nil`, и это будет означать, что записи под заданным ключом в базе данных не существует.

Но можно сделать еще лучше. Будем возбуждать ошибку `{:error, :enoent}`, только если файл недоступен. Это будет выглядеть следующим образом:

```
case File.read(...) do
  {:ok, contents} -> do_something_with(contents)
  {:error, :enoent} -> nil
end
```

Обратите внимание, что в данном случае используется сопоставление с образцом. Если ни одна из ожидаемых ситуаций не выполняется, сопоставление не удастся, и процесс с этим кодом завершится. В этом и состоит идея стратегии Let it crash. Все ожидаемые случаи (файл либо есть, либо его нет) обрабатываются, а все остальное (например, отсутствие прав для чтения файла) игнорируется. Лично я бы даже не называл это обработкой ошибок. Есть определенные ситуации, которые вы можете предвидеть, и с ними нужно что-то делать. Они не должны вызывать проблем.

Другой пример – для занесения данных в базу используется функция `File.write!/2` (обратите внимание на восклицательный знак), которая может выбросить исключение и остановить процесс. Если сохранить данные не удастся, рабочий процесс завершится, и скрывать этот факт бессмысленно. Чем быстрее это случится, тем быстрее ошибка будет занесена в журнал, замечена и исправлена.

Конечно, перезапуск процесса не всегда помогает решить возникшую проблему. Например, если в результате ошибки несколько рабочих процессов одновременно записали данные в один и тот же файл, то после перезапуска процесса все снова встанет на свои места. Некоторые запросы не выполнятся, но система продолжит обслуживать клиентов. В случае отсутствия прав на выполнение операций с файлом перезапуск ничего не изменит. Супервизор остановится и завершится, и вся система довольно быстро свернется, что, в общем-то, правильно. Какой смысл в ее работе, если данные не сохраняются.

Общее правило таково, что если вы знаете, что делать с той или иной ошибкой, то лучше ее обработать. А в случае непредвиденной ошибки позвольте процессу завершиться и обеспечьте изолирование ошибок и восстановление с помощью супервизоров.

9.3.3. Сохранение состояния

Не забывайте, что состояние после перезапуска не сохраняется. Как объяснялось в пятой главе, состояние процесса – это его личное дело. Когда процесс завершается, занимаемая им память очищается, и запускается новый процесс с новым состоянием. Благодаря этому, в случае если процесс завершился из-за нарушения целостности его состояния, перезапуск процесса с новым состоянием поможет решить проблему.

Тем не менее существуют ситуации, когда состояние упавшего процесса необходимо сохранить. Встроенного механизма на этот счет в Elixir не существует, поэтому вам придется реализовать его самим. Для этого необходимо сохранить состояние где-то за пределами процесса (например, в другом процессе или в базе данных) и восстановить его после запуска новой версии отказавшего процесса.

Этот функционал уже имеется в сервере списка дел. У вас имеется простая система базы данных, сохраняющая списки дел на диске. Когда серверный процесс запускается, первое, что он делает, – это восстановление данных. Именно на этом моменте он и может унаследовать состояние предыдущего процесса.

Сохраняя состояние, помните о том, что любое изменение функциональной абстракции данных выполняется в виде цепочки преобразований:

```
new_state =
 state
  |> transformation_1(...)
  ...
  |> transformation_n(...)
```

Операции сохранения состояния следует выполнять по завершении этих преобразований. Только тогда можно наверняка знать, что состояние является согласованным. Например, в кеш-процессе это можно сделать после изменения внутренней абстракции данных:

```
def handle_cast({:add_entry, new_entry}, {name, todo_list}) do
  new_list = TodoList.add_entry(todo_list, new_entry)
  Todo.Database.store(name, new_list)      <-- Сохранение состояния
  {:noreply, {name, new_list}}
end
```

СОВЕТ Долгосрочное состояние может оказать негативное влияние на перезапуски процесса. Скажем, из-за некорректного состояния (ставшего таким из-за бага) возникла ошибка. Если это состояние будет сохраняться, его процесс не сможет нормально работать: при каждом его перезапуске он будет наследовать некорректное состояние и снова аварийно завершаться либо сразу же, либо при обработке запроса. Все же лучше перезапускать процесс с обновленным состоянием и завершать все связанные с ним процессы, если это возможно.

Выводы

- Благодаря супервизорам непредвиденные ошибки в процессах не влияют на не связанные с ними части системы.
- Реестр позволяет обнаруживать процессы, не зная их pid. Это очень удобно, особенно если процессы были перезапущены.
- Каждый процесс должен быть частью дерева супервизоров. В таком случае всю систему (или отдельную ее часть) можно будет свернуть, завершив всего один супервизор.
- Для динамического запуска супервизоров используется модуль `DynamicSupervisor`.
- Когда процесс завершается, его состояние удаляется. Вы можете сохранять его где-то за пределами процесса, но это далеко не всегда правильно.
- Непредвиденные ошибки следует обрабатывать с помощью правильно организованной иерархии супервизоров. Обработка в явном виде с использованием конструкции `try` используется только для тех ошибок, которые вы можете продумать заранее.

Глава 10

За пределами GenServer

В главе рассматривается:

- модуль Task;
- модуль Agent;
- таблицы ETS.

В главах 8 и 9 была показана разница между рабочими процессами и процессами-супервизорами. Первые отвечают за выполнение функциональных задач системы, а вторые организуют рабочие процессы в единое дерево. Благодаря этому у вас есть возможность запускать и останавливать процессы в любом порядке, а также перезапускать критически важные процессы после их отказа.

Как уже отмечалось в разделе 9.1.6, все процессы, запускаемые из супервизора, должны быть совместимы с OTP. Процессы, порождаемые с помощью функций `spawn` и `spawn_link`, не удовлетворяют этому критерию, поэтому стоит избегать их использования во время промышленной эксплуатации. Модули `Supervisor`, `GenServer` и `Registry` позволяют запускать совместимые с OTP процессы в составе дерева супервизоров.

В этой главе вы познакомитесь с двумя дополнительными модулями – *Task* и *Agent*, процессы которых также совместимы с OTP. Модуль `Task` очень пригодится для запуска одноразовых задач, а `Agent` – для управления состоянием и предоставления к нему конкурентного доступа. Кроме этих двух модулей, мы коснемся связанной с ними функциональной возможности – таблиц ETS, которые при определенных обстоятельствах могут выступать более эффективными альтернативами `GenServer` и `Agent`.

Впереди много всего интересного, и начнем мы с модуля `Task`.

10.1. Задачи

Модуль `Task` можно использовать для запуска конкурентных задач – процессов, которые принимают на вход некоторые данные, выполняют с ними определенные вычисления и затем останавливаются. В отличие от процессов `GenServer`, представляющих собой долгоживущие серверы, процессы модуля `Task` сразу же приступают к выполнению предписанных им операций, не обрабатывают запросы и завершаются по окончании работы.

В зависимости от того, требуется ли от запущенного процесса ответ, модуль `Task` можно использовать двумя различными способами. В первом случае задачи на-

зывают *задачами с ожиданием ответа*, так как запускающий задачу процесс находится в ожидании результата. Рассмотрим этот случай более подробно.

10.1.1. Задачи с ожиданием ответа

Задача с ожиданием ответа – это процесс, который выполняет какую-либо функцию, отправляет ее результат запустившему его процессу, после чего завершается.

Рассмотрим простой пример. Предположим, вам необходимо запустить конкурентную, возможно, долго выполняющуюся задачу и получить ее результат. Длительное время выполнения можно сымитировать с помощью следующей функции:

```
iex(1)> long_job =
          fn ->
            Process.sleep(2000)
            :some_result
          end
```

Данная анонимная функция останавливается на две секунды, после чего возвращает `:some_result`.

Чтобы запустить эту функцию конкурентно, используйте функцию *`Task.async/1`*:

```
iex(2)> task = Task.async(long_job)
```

Функция `Task.async/1` принимает на вход анонимную функцию с нулевой арностью, порождает отдельный процесс и вызывает в нем эту функцию. Возвращенное функцией значение отправляется запускающему процессу в виде сообщения.

Так как вычисления производятся в отдельном процессе, функция `Task.async/1` возвращает результат незамедлительно, даже если анонимная функция выполняется достаточно долгое время. Это означает, что запускающий процесс не блокируется и может параллельно заняться выполнением других задач.

Возвращаемое функцией `Task.async/1` значение представляет собой структуру, описывающую запущенные задачи. Эту структуру можно передать функции *`Task.await/1`*, ожидающей результат задачи:

```
iex(3)> Task.await(task)
:some_result
```

Функция `Task.await/1` ожидает ответного сообщения с результатом анонимной функции от процесса, выполняющего задачу. Когда это сообщение приходит, `Task.await/1` возвращает результат анонимной функции. Если сообщение не будет получено в течение пяти секунд, `Task.await/1` возбудит исключение. Вы также можете установить желаемое максимальное время ожидания, передав его вторым аргументом функции `Task.await/2`.

Задачи с ожиданием ответа особенно удобно использовать, когда необходимо произвести несколько независимых разовых вычислений одновременно и получить результат каждого из них. Чтобы разобрать это на практике, вспомним пример из раздела 5.2.2, в котором выполнялось несколько независимых запросов и сбор их результатов. Поскольку запросы не зависят друг от друга, можно сократить время их выполнения, запустив каждый из них в отдельном процессе и направив результат в виде сообщения запускающему процессу, который будет находиться в ожидании всех результатов.

В пятой главе вы реализовали это с помощью функций spawn, send и receive, а сейчас попробуем использовать Task.async/1 и Task.await/1.

Для начала определите вспомогательную лямбда-функцию, имитирующую длительное выполнение запроса:

```
iex(1)> run_query =
          fn query_def ->
            Process.sleep(2000)
            "#{query_def} result"
          end
```

Теперь запустите пять запросов, каждый – в виде отдельной задачи:

```
iex(2)> queries = 1..5

iex(3)> tasks =
          Enum.map(
            queries,
            &Task.async(fn -> run_query.("query #{&1}") end)
          )
```

Результат, хранимый в переменной tasks, представляет собой список из пяти структур %Task{}, каждая из которых описывает задачу, выполняющую соответствующий запрос.

Для объединения результатов каждую из этих структур необходимо передать функции Task.await/1:

```
iex(4)> Enum.map(tasks, &Task.await/1)
["query 1 result", "query 2 result", "query 3 result", "query 4 result", "query 5 result"]
```

Этот код можно немного сократить, используя оператор конвейера:

```
iex(5)> 1..5 |>
          Enum.map(&Task.async(fn -> run_query.("query #{&1}") end)) |>
          Enum.map(&Task.await/1)
["query 1 result", "query 2 result", "query 3 result", "query 4 result",
 "query 5 result"]    <-------- Возвращает результат через 2 секунды
```

Все результаты возвращаются через 2 секунды, что доказывает, что каждый запрос выполняется в отдельном процессе.

Стоит отметить, что функция Task.async/1 связывает новый процесс с запускающим. Поэтому, если любой из процессов задач откажет, запускающий процесс также завершится (если он не отлавливает выходы). Отказ запускающего процесса, в свою очередь, приведет к отказу всех остальных запущенных им процессов задач.

Если вам хотелось бы обработать ошибки отдельных процессов задач, необходимо будет реализовать перехват сигналов выхода и обработку соответствующих сообщений выхода в запускающем процессе. С этим вам помогут некоторые функции модуля Task, в особенности функция *Task.async_stream/3*. Для получения более подробной информации вы можете обратиться к официальной документации по адресу: **https://hexdocs.pm/elixir/Task.html**.

А сейчас рассмотрим случай, когда запускающему процессу не обязательно ожидать результата выполнения задач.

10.1.2. Задачи без ожидания ответа

В некоторых случаях посылать ответное сообщение запускающему процессу не требуется. Допустим, во время обработки веб-запроса запускается более длительная задача, взаимодействующая с системой оплаты. Можно запустить задачу и сразу же сообщить пользователю, что запрос принят, а когда задача выполнится, сервер отправит уведомление о ее результате с помощью WebSocket или по электронной почте. Или, к примеру, задача должна произвести побочный эффект – обновить базу данных, не оповещая об этом запускающий процесс. В любом из этих двух сценариев запускающий процесс не должен обязательно быть уведомлен о результате выполнения задачи.

Кроме того, иногда нет необходимости связывать процесс задачи с запускающим процессом. Обычно это делается для того, чтобы процесс задачи продолжал работать, даже если запустивший его процесс завершится. В таком случае используется функция *Task.start_link/1*.

Функцию `Task.start_link/1` можно рассматривать в качестве совместимой с OTP обертки над функцией `spawn_link`. Она запускает отдельный процесс и связывает его с вызывающим процессом. После этого в запускающем процессе выполняется указанная анонимная функция. По окончании ее работы процесс завершается с причиной `:normal`. В отличие от функции `Task.async/1`, `Task.start_link/1` не отправляет сообщение запустившему ее процессу. Взгляните на простой пример:

```
iex(1)> Task.start_link(fn ->
          Process.sleep(1000)
          IO.puts("Hello from task")
        end)
{:ok, #PID<0.89.0>}  <-------- Результат функции Task.start_link/1

Hello from task!     <-------- Вывод через секунду
```


Рассмотрим более конкретный пример задачи без ожидания ответа. Предположим, вы хотели бы получить некоторые метрики своей системы и выводить их через каждый определенный промежуток времени. `GenServer` здесь не понадобится, так как запросы от других клиентских процессов обрабатывать не требуется. Необходим такой процесс, который бы останавливался на какое-то время, а затем собирал необходимую информацию и выводил ее.

Попробуем реализовать это для текущей системы. Для начала создадим последовательный цикл, который будет периодически собирать метрики и выводить их на экран. Код данного цикла приведен в следующем листинге.

Листинг 10.1 ❖ Вывод метрик системы (todo_metrics/lib/todo/metrics.ex)

```
defmodule Todo.Metrics do
  ...

  defp loop() do
    Process.sleep(:timer.seconds(10))
    IO.inspect(collect_metrics())
    loop()
  end

  defp collect_metrics() do
    [
```

```
      memory_usage: :erlang.memory(:total),
      process_count: :erlang.system_info(:process_count)
    ]
  end
end
```

В действительности вам, скорее всего, понадобится гораздо больше данных. и их нужно будет передать какому-либо внешнему сервису, но для наглядности оставим пример простым.

Чтобы добавить этот цикл в систему, необходимо запустить задачу.

Листинг 10.2 ❖ Задача для вывода метрик (todo_metrics/lib/todo/metrics.ex)

```
defmodule Todo.Metrics do
  use Task

  def start_link(_arg), do: Task.start_link(&loop/0)

  ...
end
```

В данном примере сначала указывается выражение `use Task`, автоматически добавляющее функцию `child_spec/1` в модуль `Todo.Metrics`. Как и в случае с `GenServer`, внедренная спецификация вызывает функцию `start_link/1`, поэтому вам придется ее определить, даже если вы не собираетесь использовать ее аргумент. Реализация `start_link/1` просто должна вызывать функцию `Task.start_link/1`, запускающую задачу, в которой и будет выполняться цикл.

С помощью этих двух строк кода теперь вы можете добавить модуль `Todo.Metrics` в дерево супервизоров, как показано в следующем листинге.

Листинг 10.3 ❖ Запуск задачи для вывода метрик из супервизора (todo_metrics/lib/todo/system.ex)

```
defmodule Todo.System do
  def start_link do
    Supervisor.start_link(
      [
        Todo.Metrics,
        ...
      ],
      strategy: :one_for_one
    )
  end
end
```

Основное назначение функции `Task.start_link/1` заключается в том, что она позволяет запускать совместимые с OTP процессы как потомков любого супервизора.

Попробуйте выполнить следующее:

```
$ iex -S mix

iex(1)> Todo.System.start_link()

[memory_usage: 29662600, process_count: 63]  <-------- Вывод через 10 секунд
[memory_usage: 29662600, process_count: 63]  <-------- Вывод через 20 секунд
```

Это был простой способ реализации периодической задачи в системе, при котором нет необходимости запускать несколько процессов ОС и использовать внешние планировщики вроде `cron`.

В более сложных сценариях следует отделять планирование от логики задачи. Основная идея состоит в том, чтобы использовать один процесс для периодического планирования и запускать каждый экземпляр задачи в отдельном одноразовом процессе. Подобный подход позволит повысить отказоустойчивость системы благодаря тому, что аварийное завершение одного процесса задачи никак не повлияет на процесс-планировщик. Попробуйте реализовать это в качестве упражнения, но имейте в виду, что для промышленного кода лучше всего использовать проверенные сторонние библиотеки, такие как Quantum (**https://github.com/quantum-elixir/quantum-core**).

На этом и закончим краткий обзор задач. Для более полной картины советую изучить официальную документацию модуля `Task`, находящуюся по адресу: **https://hexdocs.pm/elixir/Task.html**. А теперь поговорим об агентах.

10.2. Агенты

Модуль *Agent* предоставляет схожую с `GenServer` абстракцию. В работе с агентами чуть меньше формальностей, и они позволяют в некоторых местах исключить шаблонность, присущую `GenServer`. С другой стороны, модуль `Agent` не поддерживает абсолютно все сценарии, поддерживаемые `GenServer`. Как правило, если модуль на основе `GenServer` содержит реализации функций `init/1`, `handle_cast/2` и `handle_call/3`, `GenServer` можно заменить на `Agent`. Однако если вам необходимы функции `handle_info/2` или `terminate/1`, придется все же сделать выбор в пользу `GenServer`.

Рассмотрим агенты более подробно. Первым делом поговорим о том, как их использовать.

10.2.1. Использование агентов

Запустить агента можно с помощью функции *Agent.start_link/1*:

```
iex(1)> {:ok, pid} = Agent.start_link(fn -> %{name: "Bob", age: 30} end)
{:ok, #PID<0.86.0>}
```

Функция `Agent.start_link/1` запускает новый процесс и выполняет в нем указанную анонимную функцию. В отличие от процесса задачи, процесс агента не завершается после выполнения анонимной функции. Вместо этого он использует возвращаемое ею значение в качестве своего внутреннего состояния. Другие процессы могут получать доступ к этому состоянию и изменять его с помощью различных функций модуля `Agent`.

Чтобы получить состояние агента или какую-то его часть, можно использовать функцию *Agent.get/2*:

```
iex(2)> Agent.get(pid, fn state -> state.name end)
"Bob"
```

Функция `Agent.get/2` принимает в качестве аргументов pid агента и анонимную функцию. Лямбда вызывается в процессе агента и получает на вход состояние это-

го процесса. Возвращаемое этой функцией значение отправляется обратно вызывающему процессу в виде сообщения. Это сообщение получает функция Agent.get/2, после чего она возвращает результат вызывающему ее процессу.

Для изменения состояния агента можно использовать функцию *Agent.update/2*:

```
iex(3)> Agent.update(pid, fn state -> %{state | age: state.age + 1} end)
:ok
```

С помощью функции Agent.get/2 можно легко проверить, что состояние действительно изменилось:

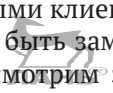
```
iex(2)> Agent.get(pid, fn state -> state end)
%{age: 31, name: "Bob"}
```

Стоит отметить, что вызов Agent.update/2 является синхронным: функция возвращает результат только после успешного изменения состояния. Если вам необходимо асинхронное обновление, используйте функцию *Agent.cast/2*.

Модуль Agent предлагает и другие функции, о которых вы можете узнать из официальной документации (**https://hexdocs.pm/elixir/Agent.html**). А теперь давайте поговорим о том, как агенты ведут себя в конкурентной системе.

10.2.2. Агенты и конкурентность

Будучи процессом, агент может быть использован многочисленными клиентскими процессами. Изменение, внесенное одним процессом, может быть замечено другими процессами в последующих операциях с агентом. Рассмотрим это на примере.



Запустим агента, которого будем использовать в качестве счетчика:

```
iex(1)> {:ok, counter} = Agent.start_link(fn -> 0 end)
```

Начальное состояние агента равно 0. Попробуем изменить его из другого процесса:

```
iex(2)> spawn(fn -> Agent.update(counter, fn count -> count + 1 end) end)
```

Теперь запросим состояние агента из процесса оболочки:

```
iex(3)> Agent.get(counter, fn count -> count end)
1
```

Данный пример показывает, что состояние связано с процессом агента. Когда один клиентский процесс изменяет это состояние, в последующих операциях других процессов будет участвовать обновленное состояние.

Процесс агента работает точно так же, как GenServer. Если несколько клиентов попытаются одновременно взаимодействовать с одним и тем же агентом, их операции будут упорядочены и выполнены по очереди. На самом деле модуль Agent реализован на Elixir на основе GenServer. Чтобы это увидеть, попробуем создать примитивную реализацию модуля наподобие Agent.

Инициализацию состояния можно реализовать следующим способом:

```
defmodule MyAgent do
  use GenServer    <-------- Модуль реализован на основе GenServer
```

```
  def start_link(init_fun) do
    GenServer.start_link(__MODULE__, init_fun)
  end

  def init(init_fun) do
    {:ok, init_fun.()}
  end

  ...
end
```

Передача анонимной функции серверу в качестве аргумента

Вызов анонимной функции и использование ее результата в качестве состояния сервера

Как вы помните из пятой главы, посылаемое сообщение может содержать любой терм. Анонимные функции, активно использующиеся в реализации модуля `Agent`, – это тоже термы. Функции интерфейса данного модуля принимают на вход анонимную функцию и передают ее серверному процессу, который, в свою очередь, вызывает эту функцию и выполняет какие-либо действия с ее результатом.

Операции `get` и `update` реализованы таким же способом:

```
defmodule MyAgent do
  ...

  def get(pid, fun) do
    GenServer.call(pid, {:get, fun})
  end

  def update(pid, fun) do
    GenServer.call(pid, {:update, fun})
  end

  def handle_call({:get, fun}, _from, state) do
    response = fun.(state)
    {:reply, response, state}
  end

  def handle_call({:update, fun}, _from, state) do
    new_state = fun.(state)
    {:reply, :ok, new_state}
  end

  ...
end
```

В действительности реализация модуля `Agent` куда более сложная и имеет более богатый функционал, но основная идея ровно та же, что и в предыдущем примере. Модуль `Agent` – это обычный `GenServer`, которым можно управлять с помощью передачи анонимных функций серверному процессу. Поэтому в конкурентных сценариях агенты ведут себя точно так же, как и процессы на основе `GenServer`.

10.2.3. Сервер списка дел на основе модуля Agent

Так как `Agent` можно использовать для управления конкурентным состоянием, он отлично подойдет для обеспечения работы сервера списка дел. Преобразование `GenServer` в агента – довольно простая задача. Необходимо всего лишь заменить пару функций интерфейса и соответствующее выражение с обратным вызовом на одну функцию, использующую API модуля `Agent`.

Полный код агента Todo.Server приведен в следующем листинге.

Листинг 10.4 ❖ Сервер списка дел на основе модуля Agent (todo_agent/lib/todo/server.ex)

```
defmodule Todo.Server do
  use Agent, restart: :temporary

  def start_link(name) do
    Agent.start_link(
      fn ->
        IO.puts("Starting to-do server for #{name}")
        {name, Todo.Database.get(name) || Todo.List.new()}
      end,
      name: via_tuple(name)
    )
  end

  def add_entry(todo_server, new_entry) do
    Agent.cast(todo_server, fn {name, todo_list} ->
      new_list = Todo.List.add_entry(todo_list, new_entry)
      Todo.Database.store(name, new_list)
      {name, new_list}
    end)
  end

  def entries(todo_server, date) do
    Agent.get(
      todo_server,
      fn {_name, todo_list} -> Todo.List.entries(todo_list, date) end
    )
  end

  defp via_tuple(name) do
    Todo.ProcessRegistry.via_tuple({__MODULE__, name})
  end
end
```

Стоит заметить, что интерфейс модуля остался прежним, а значит, менять что-либо в коде других модулей не придется.

В этом коде следует обратить особое внимание на две вещи. Первая – это выражение use Agent в самом начале модуля. Как и в случаях с GenServer и Task, это выражение автоматически добавит реализацию функции child_spec/1, позволяющую указать данный модуль в списке спецификаций потомков. Вторая – реализация функции add_entry/2 использует функцию Agent.cast/2 – асинхронную версию функции Agent.update/2, и это означает, что данная функция будет немедленно возвращать результат, и обновление будет производиться конкурентно. Agent.cast/2 используется для имитации такого же поведения, что и в предыдущей версии сервера на основе GenServer.

Всегда оборачивайте код агента в модуль

Проблема агентов состоит в том, что они делают состояние процесса общедоступным. Как вы помните, при использовании `GenServer` сервер имеет приватное состояние, изменить которое можно только с помощью четко определенных сообщений. Состояние агента может быть изменено произвольным способом через передаваемые ему анонимные функции, а значит, целостность такого состояния может быть легко нарушена. Чтобы этого избежать, следует всегда оборачивать код агента в отдельный модуль и взаимодействовать с ним только с помощью функций этого модуля. Именно это вы и сделали, когда превратили `Todo.Server` в агента.

Новая версия `Todo.Server` содержит всего 29 строк кода, что на 12 строк короче предыдущей версии с `GenServer`. В этом смысле агенты можно назвать довольно привлекательной альтернативой `GenServer`.

Однако агенты не могут сравниться с `GenServer` по числу поддерживаемых сценариев. В следующем разделе мы рассмотрим случаи, когда использование агентов неуместно.

10.2.4. Пределы возможностей агентов

Если вам необходимо реализовать обработку сообщений или добавить какие-либо действия по завершении процесса, возможностей модуля `Agent` окажется недостаточно. В этом случае следует сделать выбор в пользу `GenServer`.

Рассмотрим пример. В текущей версии системы срок хранения данных в кеше никак не ограничен. Это означает, что измененный пользователем список дел останется в памяти до тех пор, пока вся система не остановится. Очевидно, в этом нет ничего хорошего. Пользователи будут взаимодействовать с разными списками, система будет потреблять все больше и больше памяти, а в какой-то момент доступная память закончится, и произойдет коллапс.

Введем серверам определенный срок действия: будем завершать те из них, которые простаивали какое-то время.

Первый способ это реализовать – создать отдельный процесс, который будет завершать неиспользуемые серверные процессы. При этом каждому серверу необходимо будет оповещать этот процесс каждый раз при его использовании, что, возможно, превратит завершающий процесс в узкое место. Одному процессу необходимо будет обрабатывать большое количество сообщений от других процессов, и он может с этим не справиться.

Более правильным решением будет предоставить каждому серверу возможность самому решать, когда ему нужно завершиться. Так вы сможете упростить логику и избежать появления узких мест. Пример, приведенный далее, реализован на основе `GenServer`, но его можно переписать и с использованием `Agent`.

Определить время простоя серверного процесса можно несколькими способами, и мы воспользуемся самым простым из них. Добавим один дополнительный элемент в кортеж, возвращаемый функциями обратного вызова `GenServer`. Этот элемент, представленный целым числом, будет обозначать максимальное время простоя, по прошествии которого процессу `GenServer` будет отправлено сообщение.

К примеру, функция init/1 будет возвращать не {:ok, initial_state}, а {:ok, initial_state, 1000}. Последний элемент данного кортежа означает, что если в течение 1000 миллисекунд серверному процессу не придет сообщение, запрос call или cast, то будет вызвана функция handle_info/2, значение первого аргумента которой будет равно :timeout.

То же самое актуально и для функций обратного вызова handle_cast/2 и handle_call/3, которые будут возвращать кортежи {:noreply, new_state, timeout} и {:reply, response, new_state, timeout} соответственно.

Итак, чтобы серверный процесс останавливал сам себя по прошествии определенного времени бездействия, необходимо сделать следующее:

1) откатить реализацию сервера списка дел до предыдущей версии на основе GenServer;
2) добавить время простоя, выраженное целым числом, в возвращаемые функциями обратного вызова кортежи;
3) добавить функцию handle_info/2, останавливающую сервер при получении сообщения со значением :timeout.

Соответствующий код модуля Todo.Server приведен в следующем листинге.

Листинг 10.5 ❖ Задание времени простоя (todo_cache_expiry/lib/todo/server.ex)

```
defmodule Todo.Server do
  ...
  @expiry_idle_timeout :timer.seconds(10)  <-- Задание времени простоя

  def init(name) do
    IO.puts("Starting to-do server for #{name}")
    {
      :ok,
      {name, Todo.Database.get(name) || Todo.List.new()},
      @expiry_idle_timeout
    }
  end

  def handle_cast({:add_entry, new_entry}, {name, todo_list}) do
    new_list = Todo.List.add_entry(todo_list, new_entry)
    Todo.Database.store(name, new_list)
    {:noreply, {name, new_list}, @expiry_idle_timeout}
  end

  def handle_call({:entries, date}, _, {name, todo_list}) do
    {
      :reply,
      Todo.List.entries(todo_list, date),
      {name, todo_list},
      @expiry_idle_timeout
    }
  end
                                  Добавление времени простоя в ответ
  ...
end
```

Сначала объявляется атрибут модуля @expiry_idle_timeout, содержащий значение 10 000 (полученное после вызова :timer.seconds(10)). Атрибут – это константа на уровне модуля, которую вы прописываете на месте последнего элемента

возвращаемого кортежа каждой из функций обратного вызова. Благодаря этому, если сервер будет неактивен в течение 10 секунд, будет вызвана функция `handle_info(:timeout, state)`.

Теперь необходимо добавить обработку сообщения со значением `:timeout` и останов сервера, как показано в следующем листинге.

Листинг 10.6 ❖ Завершение неактивного сервера (todo_cache_expiry/lib/todo/server.ex)

```
defmodule Todo.Server do
  ...

  def handle_info(:timeout, {name, todo_list}) do
    IO.puts("Stopping to-do server for #{name}")
    {:stop, :normal, {name, todo_list}}   <-- Завершение процесса
  end
end
```

Посмотрим, как это работает. Откройте папку todo_cache_expiry, запустите систему и один серверный процесс:

```
$ iex -S mix

iex(1)> Todo.System.start_link()
iex(2)> pid = Todo.Cache.server_process("bobs_list")
```

Подождав немного, вы увидите следующее сообщение отладки:

```
Stopping to-do server for bobs_list
```

Проверим, действительно ли процесс завершился:

```
iex(3)> Process.alive?(pid)
false
```

В подобных сценариях возможностей агентов будет недостаточно, и вам придется использовать `GenServer`. `Agent` подошел бы так же хорошо, как и `GenServer`, ровно до того момента, как мы решили задать процессам срок действия. Если сценарий включает в себя обработку сообщений или действия по завершении процесса в функции `terminate/1`, то следует выбирать `GenServer`.

Лично я практически никогда не использую `Agent`. Мне не хочется тратить время на преобразование `Agent` в `GenServer`, поэтому я всегда сразу же начинаю работать с `GenServer`. Это также позволяет добиться единообразия кода, ведь все серверные процессы реализованы на основе одной и той же абстракции. Если вы находитесь в раздумьях и не можете решить, какой из двух модулей выбрать для конкретной ситуации, мой совет – всегда используйте `GenServer`, поскольку он ненамного сложнее `Agent` и покрывает больше случаев использования.

Это все, что я хотел вам рассказать об агентах. Далее мы поговорим о таблицах ETS.

10.3. Таблицы ETS

Таблицы ETS (Erlang Term Storage – хранилище термов Erlang) – это эффективный механизм, позволяющий нескольким процессам разделять общее состояние. Его можно считать своеобразным средством оптимизации. Что бы вы ни реализовали

на основе таблиц ETS, все то же самое можно сделать и на основе абстракций Agent и GenServer, однако версия с таблицами, скорее всего, будет иметь более высокую производительность. Таблицы ETS покрывают ограниченное число сценариев использования, а потому не смогут в большинстве случаев заменить GenServer.

Как правило, таблицы ETS используются для реализации хранилищ ключ/значение с общим доступом или счетчиков. Для этого можно использовать и GenServer, но тогда вы можете столкнуться с проблемами по части производительности и масштабируемости.

Чтобы это продемонстрировать, давайте реализуем конкурентное хранилище ключ/значение на основе GenServer. Для начала взгляните на пример использования такого хранилища:

```
iex(1)> KeyValue.start_link()
{:ok, #PID<0.118.0>}

iex(2)> KeyValue.put(:some_key, :some_value)
:ok

iex(3)> KeyValue.get(:some_key)
:some_value
```

Полный код модуля KeyValue приведен в следующем листинге.

Листинг 10.7 ❖ Хранилище ключ/значение на основе GenServer (key_value/lib/key_value.ex)

```
defmodule KeyValue do
  use GenServer

  def start_link do
    GenServer.start_link(__MODULE__, [], name: __MODULE__)
  end

  def put(key, value) do
    GenServer.cast(__MODULE__, {:put, key, value})
  end

  def get(key) do
    GenServer.call(__MODULE__, {:get, key})
  end

  def init(_) do
    {:ok, %{}}
  end

  def handle_cast({:put, key, value}, store) do
    {:noreply, Map.put(store, key, value)}
  end

  def handle_call({:get, key}, _, store) do
    {:reply, Map.get(store, key), store}
  end
end
```

Все довольно стандартно. Модуль KeyValue – это простой GenServer, хранящий в своем состоянии словарь. Запросы put и get сводятся к вызовам функций Map.put/3 и Map.get/2 в серверном процессе.

Теперь попробуем замерить производительность быстрым и не особо показательным способом. Откройте папку key_value и выполните следующую команду:

```
mix run -e "Bench.run(KeyValue)"
```

Команда *mix run* компилирует проект, запускает экземпляр BEAM и затем выполняет выражение, указанное после аргумента `-e`, то есть вызывает функцию `Bench.run/1`. Как только функция выполнится, экземпляр BEAM завершится.

Модуль `Bench`, находящийся в файле key_value/lib/bench.ex, выполняет простой нагрузочный тест. Он запускает сервер `KeyValue` и выполняет 10 операций `put` для каждого из одного миллиона ключей. После каждой из этих операций следует операция `get`. В сумме программа выполняет 20 млн операций.

По завершении теста функция выводит полученный показатель производительности:

```
mix run -e "Bench.run(KeyValue)"
621003 operations/sec
```

620 тыс. операций в секунду – довольно хорошая производительность. Но давайте проверим, как быстро сервер ключ/значение работает при обслуживании нескольких клиентских процессов. Это можно сделать, указав опцию `:concurrency` при вызове функции `Bench.run`:

```
mix run -e "Bench.run(KeyValue, concurrency: 1000)"
325055 operations/sec
```

Как видите, 1000 клиентских процессов сервер обслуживает почти в 2 раза медленнее. Почему так происходит? Главной проблемой является то, что все операции выполняются в порядке очереди одним и тем же серверным процессом, как показано на рис. 10.1.

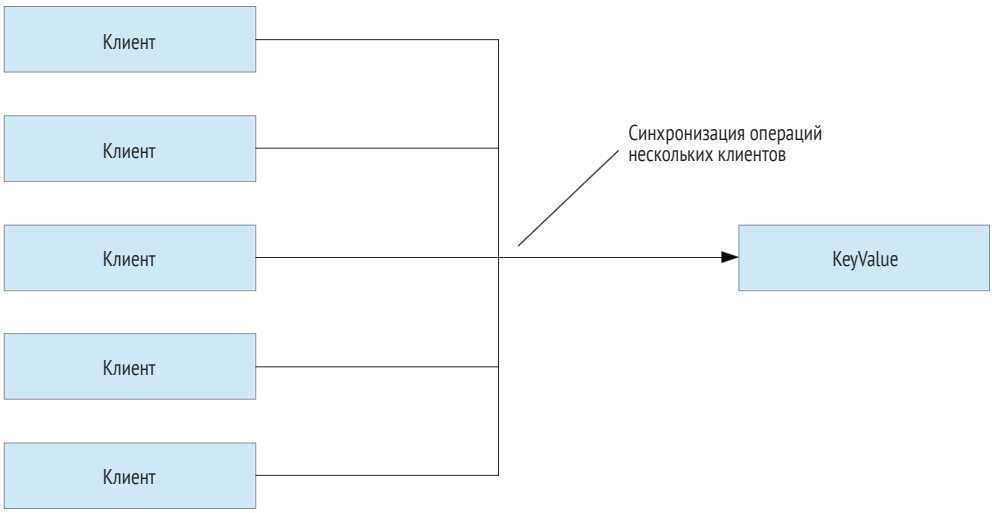


Рис. 10.1 ❖ Узкое место программы

Таким образом, серверный процесс становится узким местом программы, что отрицательно сказывается на масштабируемости. Система не способна эффек-

тивным образом задействовать все аппаратные ресурсы. Все запросы от 1000 конкурентных клиентских процессов выполняются по одному за раз.

Кроме того, имейте в виду, что даже в умеренно конкурентных системах запущенных процессов в разы больше, чем доступных ядер ЦП. Соответственно, все процессы не могут работать одновременно; каким-то из них придется ждать своей очереди.

Как объяснялось в пятой главе, виртуальная машина старается по максимуму использовать доступные ей ресурсы, но факт остается фактом – большое количество процессов гонится за ограниченными ресурсами. В результате серверный процесс не получает в свое распоряжение ни одного целого ядра ЦП. Время от времени, если планировщики BEAM выполняют другие процессы системы, ему приходится ждать своей очереди. При таком сценарии сервер ключ/значение располагает меньшим количеством ресурсов ЦП и, как следствие, работает медленнее.

Это не значит, что от использования процессов нужно отказаться. Чтобы улучшить масштабируемость и отказоустойчивость системы, необходимо обеспечить конкурентное выполнение независимых задач. Процессы также отлично подходят для хранения изменяющегося состояния. В данном случае проблема заключается не в наличии большого количества работающих процессов, а в том, что все они зависят от одного-единственного процесса.

Приведенный в примере код будет работать гораздо быстрее, если его реализовать с помощью таблиц ETS. В следующем разделе вы узнаете, что они из себя представляют и как с ними работать.

10.3.1. Основные операции

Таблица ETS – это особая структура данных для хранения термов Erlang в памяти. Они позволяют организовать общий доступ к какому-либо состоянию системы без создания специального серверного процесса. Данные помещаются в таблицу ETS – динамическую структуру, в которой можно хранить кортежи.

По сравнению с другими структурами, таблицы ETS имеют следующие отличия:

1) специального типа данных для ETS не существует. Таблицы идентифицируются по ID (ссылке) или глобальному имени (атому);
2) таблицы ETS являются изменяемыми. Операция записи в таблицу окажет влияние на последующие операции чтения;
3) несколько процессов могут производить операции записи и чтения данных с одной и той же таблицей. Эти операции выполняются конкурентно;
4) обеспечивается минимальный уровень изоляции. Несколько процессов могут пытаться записать данные в одну и ту же строку одной и той же таблицы. Удается это тому из них, который по очереди делает это последним;
5) таблица ETS находится в отдельной области памяти. При помещении данных в таблицу и их извлечении создается глубокая копия этих данных;
6) таблицы самостоятельно выполняют сбор мусора: память, занимаемая перезаписанными или удаленными данными, немедленно освобождается;
7) любая таблица ETS тесно связана с процессом-владельцем (по умолчанию с тем процессом, который ее создал). Если этот процесс завершится, принадлежащая ему таблица уничтожается;

8) требуемая для работы таблицы память освобождается только при завершении процесса-владельца. Даже если вы не пользуетесь этой таблицей, она все равно занимает память.

Перечисленные выше особенности дают понять, что таблицы ETS чем-то напоминают процессы. На деле можно сказать, что они имеют ту же семантику. Таблицу ETS можно реализовать и на основе процесса, но она окажется менее эффективной. В виртуальной машине BEAM таблицы ETS представлены кодом на языке C, что обеспечивает более высокую скорость работы.

Пятый пункт списка особенностей представляет особый интерес. Создание глубоких копий данных при их записи в таблицу и извлечении из нее позволяет избежать стандартной проблемы при работе с изменяемыми данными. Когда данные считываются с таблицы, создается их экземпляр, который не может быть изменен. Другие процессы могут редактировать строки с этими данными в таблице ETS, но считанная копия остается нетронутой.

Рассмотрим несколько примеров. Все функции, относящиеся к таблицам ETS, находятся в Erlang-модуле `:ets` (**http://erlang.org/doc/man/ets.html**). Создать таблицу можно с помощью функции *`:ets.new/2`*:

```
iex(1)> table = :ets.new(:my_table, [])
#Reference<0.970221231.4117102596.53103>
```

Первый аргумент этой функции – это имя таблицы, которое понадобится вам, только если вы захотите ее зарегистрировать (об этом чуть позже). Также можно передавать функции различные опции, информацию о которых настоятельно советую изучить в официальной документации.

Результатом функции `:ets.new/2` является ссылка, уникальный непрозрачный терм, представляющий таблицу ETS в текущей системе.

В таблицу можно помещать несколько строк, каждая из которых – это кортеж с произвольным количеством элементов (минимум с одним). Каждый элемент может содержать любой терм Erlang, включая глубоко вложенные списки, кортежи, словари и любые другие данные, которые можно помещать в переменные.

Чтобы добавить данные в таблицу, используйте функцию *`:ets.insert/2`*:

```
iex(2)> :ets.insert(table, {:key_1, 1})
true
iex(3)> :ets.insert(table, {:key_2, 2})
true
iex(4)> :ets.insert(table, {:key_1, 3})
true
```

Вставка новой строки

Перезапись существующей строки

Первый элемент кортежа – это ключ, по которому можно осуществлять быстрый поиск записи в таблице. По умолчанию таблицы ETS организованы как множества, что означает, что вы не можете хранить в них несколько кортежей с одинаковыми ключами. Соответственно, при осуществлении записи кортежа с уже имеющимся ключом существующая строка будет заменена новой.

Проверить это можно с помощью функции *`:ets_lookup`*, возвращающей список записей по указанному ключу:

```
iex(5)> :ets.lookup(table, :key_1)
[key_1: 3]
```

```
iex(6)> :ets.lookup(table, :key_2)
[key_2: 2]
```

Зачем же данные возвращаются в виде списка, если на один уникальный ключ может быть только одна запись? На самом деле таблицы ETS поддерживают и другие типы организации данных, и некоторые из них позволяют хранение дубликатов записей. Существуют следующие виды таблиц:

- `:set` (множество) – значение по умолчанию, одна запись на уникальный ключ;
- `:ordered_set` (упорядоченное множество) – то же, что и :set, но записи упорядочены по ключам (используются операторы сравнения < и >);
- `:bag` (мультимножество) – возможно хранение нескольких записей с одинаковыми ключами, но разными данными;
- `:duplicate_bag` (мультимножество с дублированием) – то же, что и `:bag`, но возможно хранение полностью идентичных записей.

Кроме этого, процессы могут иметь различные права доступа к таблицам ETS:

- `:protected` – значение по умолчанию, при котором чтение данных доступно всем процессам, а запись – только владельцу;
- `:public` – всем процессам доступны операции чтения и записи;
- `:private` – доступ к таблице есть только у процесса-владельца.

Чтобы создать таблицу определенного типа и изменить права доступа, просто включите соответствующие опции в список опций, передаваемых функции `:ets.new/2`. Например, создать общедоступное мультимножество с дублированием можно вызовом функции со следующим набором параметров:

```
:ets.new(:some_table, [:public, :duplicate_bag])
```

И наконец, поговорим об имени таблицы. Этот аргумент может быть выражен атомом, и по умолчанию он нигде не используется (но, как ни странно, указывать его обязательно). Вы можете создать несколько таблиц с одинаковыми именами, но они все равно будут считаться разными таблицами.

Однако если при создании таблицы указать параметр `:named_table`, к ней можно будет обращаться по имени:

```
iex(1)> :ets.new(:my_table, [:named_table])
:my_table
iex(2)> :ets.insert(:my_table, {:key_1, 3})
iex(3)> :ets.lookup(:my_table, :key_2)
[]
```

Создание именованной таблицы

Обращение к таблице по имени

В этом смысле имя таблицы похоже на локально зарегистрированное имя процесса. Это символьное имя, избавляющее вас от необходимости повсеместного использования ссылки на таблицу.

Попытка создания еще одной таблицы с таким же именем возбудит ошибку:

```
iex(4)> :ets.new(:my_table, [:named_table])
** (ArgumentError) argument error
    (stdlib) :ets.new(:my_table, [:named_table])
```

СОВЕТ Таблицы ETS – это ограниченный ресурс. Вы можете создать не более 1400 таблиц на один экземпляр BEAM. Этот предел можно увеличить, установив переменную окружения `ERL_MAX_ETS_TABLES` при запуске системы (например, `iex --erl '-env ERL_MAX_ETS_TABLES 10000'`). Ограничение введено потому, что каждая таблица при создании занимает несколько килобайтов памяти, поэтому в целом не следует особо увлекаться использованием таблиц ETS в системах.

10.3.2. Хранилище ключ/значение на основе таблицы ETS

Вооружившись новыми знаниями о таблицах ETS, попробуем реализовать то же хранилище ключ/значение другим способом. Идея проста и заключается в следующем. Будем по-прежнему использовать серверный процесс `GenServer`, а в функции `init/1` этот процесс будет создавать именованную таблицу ETS с общим доступом. Функции `put` и `get` будут напрямую работать с таблицей без отправки запроса серверу.

Код этого решения следует разместить в том же проекте, что и код предыдущего. Для начала реализуйте запуск и инициализацию процесса-владельца, как показано в следующем листинге.

Листинг 10.8 ❖ Создание таблицы ETS (key_value/lib/ets_key_value.ex)

```
defmodule EtsKeyValue do
  use GenServer

  def start_link do
    GenServer.start_link(__MODULE__, nil, name: __MODULE__)   <-- Запуск процесса-владельца
  end

  def init(_) do
    :ets.new(                                                  <-- Создание таблицы
      __MODULE__,
      [:named_table, :public, write_concurrency: true]
    )

    {:ok, nil}
  end

  ...
end
```

В функции `start_link` запускается `GenServer`, после чего в функции обратного вызова `init/1` создается новая таблица ETS. Указанный параметр `:named_table` означает, что клиентские процессы могут обращаться к этой таблице по ее имени (имени модуля). К созданной таблице имеется общий доступ, что позволяет клиентам производить в нее запись данных. Тип таблицы здесь не указан, соответственно, по умолчанию данная таблица организована в виде множества.

Обратите внимание на параметр `:write_concurrency`, указанный в функции `:ets.new`. Он разрешает выполнять операции записи в таблицу конкурентно, что нам как раз и нужно. Существует также опция `:read_concurrency`, позволяющая увеличить производительность системы в ряде случаев. В данном примере она не используется, потому что модуль `Bench` выполняет множество чередуемых операций

чтения и записи, и подключение этой опции сделает только хуже. Не стоит включать их везде с закрытыми глазами. Всегда старайтесь оценивать их возможные положительные и отрицательные эффекты.

Теперь добавьте реализации следующих операций:

```
defmodule EtsKeyValue do
  ...

  def put(key, value) do
    :ets.insert(__MODULE__, {key, value})  <-- Добавление пары ключ/значение
  end

  def get(key) do
    case :ets.lookup(__MODULE__, key) do  <-- Поиск по таблице
      [{^key, value}] -> value  <-- Данные найдены
      [] -> nil  <-- Данные не найдены
    end
  end

  ...
end
```

В данном коде показаны простейшие способы применения функций модуля `:ets`. Функция `ets.insert/2` добавляет запись в таблицу, а функция `:ets.lookup/2` осуществляет поиск по ней. Поскольку тип данной таблицы – множество, получаемый в результате поиска список может содержать максимум один элемент – пару ключ/значение для указанного ключа. Если запись с таким ключом отсутствует, список будет пустым.

Самое важное отличие этой реализации от предыдущей в том, что операции `get` и `put` теперь не проходят через серверный процесс, а следовательно, несколько клиентских процессов могут одновременно работать с хранилищем, не блокируя друг друга, как показано на рис. 10.2.

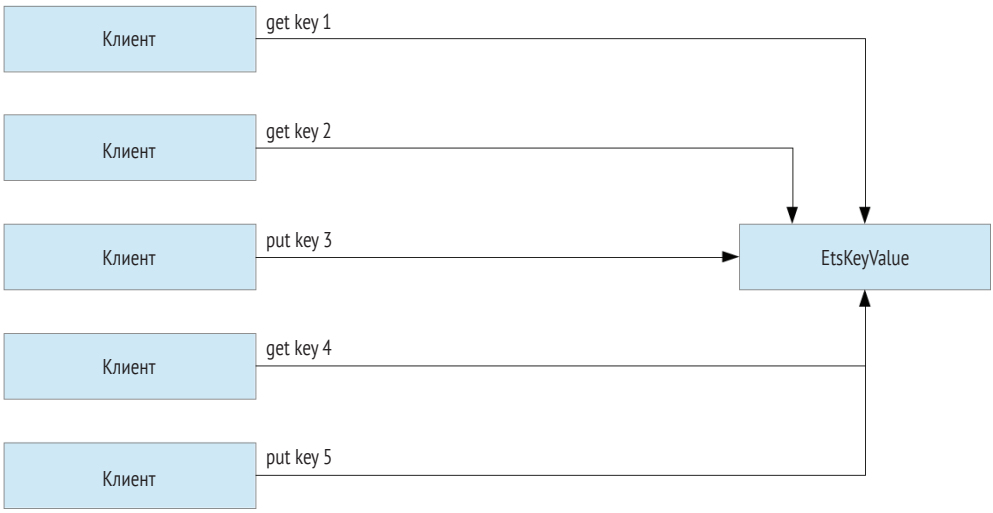


Рис. 10.2 ❖ Конкурентность в хранилище ключ/значение на основе таблицы ETS

Как видите, операции по разным ключам могут выполняться параллельно, а операции по одному ключу синхронизируются во избежание состояния гонки. Если в вашей таблице имеется большое множество различных ключей, шанс конфликтов очень мал, планировщик будет использоваться эффективнее, а значит, удастся достичь большего масштабирования.

Проверим, как работает новая версия хранилища:

```
iex(1)> EtsKeyValue.start_link()
{:ok, #PID<0.109.0>}

iex(2)> EtsKeyValue.put(:some_key, :some_value)
true

iex(3)> EtsKeyValue.get(:some_key)
:some_value
```

Кажется, все в порядке. Теперь измерим производительность следующим образом:


```
mix run -e "Bench.run(EtsKeyValue)"
3576332 operations/sec
```

На своем компьютере я получил вполне ожидаемую цифру в 3,6 млн запросов в секунду, что практически в 6 раз превышает производительность хранилища на основе `GenServer`, способного обрабатывать 620 000 запросов в секунду!

На то есть несколько причин. Во-первых, операции с ETS обрабатываются прямо в клиентском процессе. В примере с `GenServer` время тратится на помещение сообщения в почтовый ящик процесса-получателя, ожидание этим процессом своей очереди и обработку запроса. Если запрос синхронный, к этому списку еще добавляется ожидание ответного сообщения. Во-вторых, изменения, вносимые в таблицы ETS, деструктивны. Если значение под каким-либо определенным ключом заменить, то старое значение будет сразу же уничтожено, и сборщику мусора впоследствии не придется ничего подчищать. Преобразование стандартной иммутабельной структуры данных, напротив, оставляет за собой мусор. При работе хранилища, реализованного на основе `GenServer`, частые операции записи в таблицу оставят в памяти большое количество ненужных данных, и серверный процесс будет заблокирован на время их очистки сборщиком мусора.

Как видите, даже в таком простом последовательном сценарии вы получаете значительное преимущество при использовании таблиц ETS. Но давайте проверим, как обстоят дела с обслуживанием множества клиентов:

```
mix run -e "Bench.run(EtsKeyValue, concurrency: 1000, num_updates: 100)"
16993927 operations/sec
```

Обратите внимание на опцию `num_updates: 100`. Она указана для проведения более длительного теста, так как реализация на основе ETS работает в разы быстрее. В данном случае выполняется 100 операций `put` и 100 `get` по каждому ключу.

При наличии 1000 клиентских процессов система справилась с тестом в 5 раз быстрее, чем в прошлый раз. По сравнению с решением на основе `GenServer`, производительность увеличилась в 52 раза (17 млн против 325 тыс. запросов в секунду). С увеличением общего количества запущенных в системе процессов серверный процесс замедляется, в то время как кеш на основе ETS масштабируется гораздо лучше.

Главным образом это происходит потому, что операции с кешем выполняются в клиентском процессе, и упорядочивать их, как это делается с GenServer, нет необходимости. Элементарные операции, реализованные с помощью функций модуля :ets, синхронизованы должным образом и могут выполняться одновременно в нескольких процессах. Операции по разным ключам и даже несколько операций чтения по одному и тому же ключу могут выполняться параллельно. Только операции записи по одному ключу блокируют друг друга.

С другой стороны, возможности операций записи ограничены. Вы можете вставить в таблицу пару ключ/значение с помощью функции :ets.insert/2, удалить (:ets.delete/2) и изменить (:ets.update_element/3) запись, а также обновить целое число в записи (:ets.update_counter/4). В более сложных сценариях, скорее всего, этих функций будет недостаточно, и вам придется использовать GenServer. Таблицы ETS – это всего лишь средство оптимизации: они невероятно эффективны в простых задачах, но не настолько мощные и гибкие, как серверные процессы.

Если вас одолевают сомнения по поводу того, какое решение выбрать, советую вам все же начать с GenServer. Это довольно простое решение, обеспечивающее достаточную производительность в большинстве случаев. Если вам очевидно, что определенный серверный процесс становится узким местом системы, попробуйте перейти на таблицу ETS. Чаще всего для этого вам понадобится всего лишь изменить реализацию. Например, если сравнить модули KeyValue и EtsKeyValue, можно видеть, что они имеют один и тот же публичный интерфейс. Именно поэтому обобщенный модуль Bench работает и с тем, и с другим решением.

Вы можете задуматься о том, для чего GenServer присутствует в версии хранилища с таблицей ETS. Единственная причина – сохранить активность таблицы. Как вы помните, таблица ETS удаляется из памяти после завершения процесса-владельца, поэтому для создания и сохранения таблицы нужен отдельный процесс с длительным временем жизни.

10.3.3. Прочие операции ETS

В прошлом разделе были рассмотрены только операции добавления данных и поиска по ключу. Пожалуй, эти две операции, наряду с операцией удаления всех записей по заданному ключу (:ets.delete/2), являются самыми часто используемыми на практике.

При создании таблиц необходимо учитывать, что операции по ключу выполняются очень быстро. Вам нужно добиться выполнения максимально возможного количества операций в единицу времени, а для этого код, относящийся к таблице ETS, должен работать как можно быстрее.

В некоторых случаях вам может понадобиться выполнить поиск или изменение данных не по ключу и возвратить список записей, соответствующих определенному критерию. Это можно сделать несколькими способами.

Самое простое и одновременно менее эффективное решение – преобразовать таблицу в список с помощью функции *:ets.tab2list/1*. Затем вы сможете перебрать этот список и отфильтровать результаты, используя функции модулей Enum и Stream.

Еще один вариант – функции *:ets.first/1* и :ets.next/2, позволяющие произвести итерационный обход таблицы. Однако имейте в виду, что этот обход будет

неизолированным, то есть во время его выполнения клиентские процессы могут изменять данные в таблице. Чтобы этого избежать, все операции записи и обхода необходимо выполнять по порядку в одном и том же процессе. Альтернативой выступает функция *:ets.safe_fixtable/2*, обеспечивающая слабую защиту при обходе записей: итерации выполняются без ошибок, и каждый элемент анализируется только один раз. Но не факт, что вставленные в таблицу во время выполнения итерации строки будут включены в работу.

Все описанные выше способы не отличаются высокой производительностью. Учитывая, что данные при выполнении каждой итерации копируются из области памяти таблицы в процесс, в конечном счете будет скопирована вся таблица, а если вам необходимо получить всего пару записей по определенному критерию, то это пустая трата ресурсов.

Гораздо удобнее работать с шаблонами сопоставления – специальными конструкциями, позволяющими описывать запрашиваемые данные.

Шаблоны сопоставления

Получить отдельные записи довольно просто с помощью сопоставления с образцом. Пусть имеется таблица ETS, в которой хранится список дел:

```
iex(1)> todo_list = :ets.new(:todo_list, [:bag])
iex(2)> :ets.insert(todo_list, {~D[2018-05-24], "Dentist"})
iex(3)> :ets.insert(todo_list, {~D[2018-05-24], "Shopping"})
iex(4)> :ets.insert(todo_list, {~D[2018-05-30], "Dentist"})
```

В данном случае таблица организована как мультимножество, что позволяет хранить в ней несколько записей с одинаковыми ключами (датой).

Чаще всего пользователям интересен запрос по ключу. Например, запрос на получение всех событий на определенную дату:

```
iex(5)> :ets.lookup(todo_list, ~D[2018-05-24])
[{~D[2018-05-24], "Dentist"}, {~D[2018-05-24], "Shopping"}]
```

Также может понадобиться вывести все даты на определенное событие. Вот как это можно сделать с помощью шаблонов:

```
iex(6)> :ets.match_object(todo_list, {:_, "Dentist"})
[{~D[2018-05-24], "Dentist"}, {~D[2018-05-30], "Dentist"}]
```

Функция *:ets.match_object/2* принимает *шаблон сопоставления* – кортеж, определяющий вид искомой записи. Атом `:_` означает, что на его месте может стоять любое значение, а шаблону `{:_, "Dentist"}` соответствуют любые записи, вторым элементом которых является `Dentist`.

Обратите внимание, что это не обычное сопоставление с образцом. Кортеж передается функции `:ets.match_object/2`, перебирающей все записи и возвращающей только те из них, которые соответствуют шаблону. Именно поэтому вместо привычной универсальной анонимной переменной (_) на месте не имеющего значение элемента следует передавать атом (:_). Существует также функция *:ets.match_delete/2*, которую можно использовать для удаления нескольких элементов одним вызовом.

Шаблоны сопоставления не только выглядят более элегантно по сравнению с обычными операциями обхода, но и имеют значительное преимущество по ча-

сти производительности. Как вы помните, данные всегда копируются из таблицы в выбранный процесс. При использовании функции `:ets.tab2list/1` или реализации простого обхода в процесс будут скопированы абсолютно все записи таблицы. Функция `:ets.match_object/2`, напротив, копирует только выбранные по какому-либо критерию записи, что гораздо эффективнее.

Помимо шаблонов сопоставления можно также создавать более сложные запросы, задавая составные фильтры, и даже делать выборку отдельных полей. Это реализуется с помощью *спецификации сопоставления*, состоящей из следующих элементов:

- голова – шаблон, описывающий необходимые к получению записи;
- охранное выражение – дополнительные фильтры;
- результат – вид возвращаемых данных.

Эти спецификации можно передавать функции `:ets.select/2`, и она будет возвращать соответствующий им результат.

Если вы обратитесь к официальной документации функции `:ets.select/2` (**http://erlang.org/doc/man/ets.html#select-2**), то увидите, что спецификации сопоставления довольно легко становятся слишком громоздкими. Чтобы упростить работу с ними, можно использовать стороннюю библиотеку ex2ms (**https://github.com/ericmj/ex2ms**).

Другие случаи использования ETS

Пожалуй, наиболее часто таблицы ETS нужны для управления разделяемым состоянием сервера. Кроме этого, таблицы можно использовать для хранения данных процессов. Как вы помните из глав 8 и 9, после завершения процесса его состояние уничтожается. Если вы хотите сохранить это состояние и использовать его после перезапуска процесса, проще всего сделать это с помощью публичной таблицы ETS. Это решение будет работать довольно быстро и обеспечит восстановление данных после аварий.

Но будьте с ним осторожны. В главах 8 и 9 уже говорилось о том, почему процессы все же следует перезапускать с новым состоянием. Вы также можете попробовать восстановить состояние на основе данных других процессов. В таблице ETS (или где бы то ни было) стоит сохранять только состояние основных процессов, входящих в состав ядра ошибок системы.

Таблицы ETS могут выступать и в качестве более быстрой альтернативы неизменяемых структур данных, а именно словарей. Так как при изменении данных в таблицах ETS старые данные немедленно уничтожаются, не будет тратиться время на сбор мусора, а значит, вы получите более предсказуемые временные показатели.

И все же тут есть одно «но». При выполнении операций данные копируются из таблицы в клиентский процесс, следовательно, выполняя действия с длинной записью со сложной структурой, таблицы ETS могут работать медленнее, чем обычные неизменяемые структуры данных. Еще один недостаток таблиц заключается в том, что их нельзя передавать другим экземплярам BEAM, а значит, сложнее по максимуму использовать возможности распределенных систем (подробнее об этом в главе 12).

Подытожим: старайтесь делать выбор в пользу неизменяемых структур данных везде, где это возможно, а прибегать к использованию таблиц ETS только в тех случаях, в которых они гарантируют существенное улучшение производительности.

За пределами ETS

В состав стандартной поставки Erlang входят еще два средства, тесно связанных с ETS и позволяющих легко реализовать и запустить встроенную базу данных в процессе ОС BEAM. Далее мы лишь кратко пробежимся по ним, и если они вас заинтересуют, вы можете изучить их более подробно самостоятельно.

Механизм *DETS* (Disk-Based Erlang Term Storage, **http://erlang.org/doc/man/dets.html**) – это дисковое хранилище термов Erlang. Так же, как и ETS, DETS основан на концепции таблиц, и управление каждой из них реализуется в отдельном файле. Интерфейс соответствующего модуля `:dets` напоминает интерфейс `:ets`, но с ограниченным функционалом. DETS предоставляет простой способ сохранения данных на диске, а также обеспечивает базовую изолированность процессов: операции записи выполняются конкурентно, даже если они работают с одной и той же строкой таблицы.

Также Erlang поставляется с базой данных *Mnesia* (**http://erlang.org/doc/apps/mnesia/users_guide.html**), построенной на основе механизмов ETS и DETS и обладающей рядом привлекательных функциональных особенностей:

- Mnesia – это встроенная база данных: она работает в том же экземпляре BEAM, что и остальной код на Elixir/Erlang;
- данные представляют собой термы Erlang;
- таблицы могут храниться в оперативной памяти (ETS) или на диске (DETS);
- доступны такие стандартные особенности баз данных, как сложные транзакции, «грязные» операции и быстрый поиск с помощью вторичных индексов;
- поддержка шардинга и репликации.

Все эти особенности делают Mnesia отменным инструментом для хранения данных. Вам нужно лишь инициализировать базу данных из кода, отвечающего за запуск системы, и она готова к использованию. Также вы получаете огромное преимущество, запуская всю систему в одном процессе ОС.

Недостаток Mnesia в том, что она не пользуется популярностью за пределами сообщества Elixir/Erlang. Следовательно, по сравнению с популярными СУБД, у нее не такая большая аудитория пользователей и слабая поддержка на уровне инструментальных средств. Также нужно знать некоторые хитрости для работы с Mnesia в больших системах. Например, таблицы DETS не могут занимать больше 4 Гб памяти, значит, большие таблицы придется разделять на более мелкие.

10.3.4. Упражнение: реестр процессов

А теперь настало время для практики. Классическим примером реального использования ETS является реестр процессов. Модуль `Registry` опирается на оптимальную комбинацию `GenServer` и ETS для достижения максимальной эффективности. В данном упражнении вы реализуете простую версию уникального реестра `:unique`.

Данный реестр должен будет работать следующим образом:

```
iex(1)> SimpleRegistry.start_link()
{:ok, #PID<0.89.0>}

iex(2)> SimpleRegistry.register(:some_name)
:ok
```

Успешная регистрация

```
iex(3)> SimpleRegistry.register(:some_name)
:error
```
Ошибка регистрации уже имеющегося процесса

```
iex(4)> SimpleRegistry.whereis(:some_name)
#PID<0.87.0>
```
Успешный поиск

```
iex(5)> SimpleRegistry.whereis(:unregistered_name)
nil
```
Неудачный поиск

Интерфейс модуля `SimpleRegistry` будет очень простым. Серверный процесс запускается и локально регистрируется. Затем любой процесс может зарегистрировать себя с помощью функции `SimpleRegistry.register/1`, передав в нее имя, выраженное произвольным термом. При удачной регистрации функция возвращает `:ok`, а если данное имя уже занято – `:error`. Поиск выполняет функция `SimpleRegistry.whereis/1`, возвращающая pid по заданному ключу в случае успеха и `nil` в случае отсутствия искомого процесса в реестре.

Кроме того, процесс реестра должен быть способен обнаружить аварийное завершение любого из зарегистрированных процессов и удалить все связанные с ним записи.

`SimpleRegistry` не будет обладать такими особенностями модуля `Registry`, как поддержка via-кортежей, регистрация дубликатов и возможность создавать несколько экземпляров реестра.

Для создания такого реестра выполните описанные ниже действия.

1. Реализуйте первую версию `SimpleRegistry` на основе `GenServer`. Функции `register` и `whereis` сделайте запросами типа call.
2. Состояние `GenServer` должно быть представлено словарем, ключами которого будут зарегистрированные имена процессов, а значениями – их pid.
3. Процесс реестра должен быть связан с вызывающим процессом, чтобы во время обработки запроса `:register` он мог обнаружить завершение процесса и снять его с регистрации. Поэтому серверный процесс реестра должен отлавливать выходы (вызовом `Process.flag(trap_exit: true)` в `init/1`).
4. В функции `handle_info` процесс реестра должен обработать результат типа `{:EXIT, pid, reason}` и удалить из словаря все записи, относящиеся к завершившемуся процессу.

Создав такой реестр, вы можете вынести некоторые его части за пределы серверного процесса. В частности, используя таблицы ETS, можно выполнить регистрацию и поиск по реестру в клиентских процессах. Вот как это можно сделать.

1. Во время инициализации процесс реестра должен создать таблицу ETS с общим доступом. Эта таблица будет приводить в соответствие имена и pid регистрируемых процессов, и процессу реестра не потребуется хранить эти данные в своем состоянии.
2. Регистрацию следует выполнять с помощью функции *`:ets.insert_new/2`* (**http://erlang.org/doc/man/ets.html#insert_new-2**). Эта функция добавляет новую запись, только если записи под таким же ключом не существует. Это означает, что ее можно вызывать одновременно из разных процессов. Функция возвращает true, если запись была добавлена, и false в обратном случае.

3. Перед вызовом функции `:ets.insert_new/2` вызывающий процесс необходимо связать с серверным с помощью функции `Process.link/1`. Этот вызов можно обернуть в функцию `SimpleRegistry.register/1`.
4. Реализация `whereis/1` должна включать вызов функции `:ets.lookup/2` и вывод результата.
5. Серверный процесс должен обрабатывать сообщения типа `:EXIT` и удалять записи, относящиеся к завершенным процессам. Это можно легко реализовать с помощью функции `:ets.match_delete/2`.

Данное упражнение несколько сложнее предыдущих, но в нем грамотно сочетается сразу несколько подходов, изученных вами в последних главах. При возникновении трудностей ищите готовое решение в папке process_registry. Там вы увидите две реализации: простую на основе `GenServer` и более быструю с использованием таблицы ETS.

Выводы

- Задачи используются для запуска совместимых с OTP конкурентных рабочих процессов.
- Агенты упрощают реализацию процессов, хранящих свое состояние, но не предназначенных для обработки простых сообщений.
- Таблицы ETS увеличивают производительность системы в некоторых случаях, например при использовании структур для хранения пар ключ/значение с общим доступом.

Глава 11

Работа с компонентами

В главе рассматривается:
- создание OTP-приложений;
- работа с зависимостями;
- создание веб-сервера;
- настройка приложений.

Пришло время поговорить о том, как подготовить систему к релизу, развернуть и запустить ее. В этой главе вы узнаете об OTP-приложениях, позволяющих формировать систему из нескольких повторно используемых компонентов. OTP-приложения – стандартный способ организации промышленных систем и библиотек на Elixir/Erlang, и их использование дает такие преимущества, как управление зависимостями, упрощенный запуск системы и возможность создания автономных и готовых к развертыванию релизов.

Вы научитесь создавать приложения и работать с зависимостями, а также превратите текущую реализацию системы в полноценное OTP-приложение и добавите HTTP-интерфейс с помощью некоторых сторонних библиотек экосистемы Elixir/Erlang. Что ж, приступим к изучению OTP-приложений, впереди нас ждет много работы.

11.1. OTP-приложения

OTP-приложение – это компонент, который состоит из нескольких модулей и может зависеть от других приложений. Систему довольно легко превратить в такое приложение, и тогда ее со всеми зависимыми компонентами можно будет запускать в одно действие. Ваша текущая система управления списками дел уже является OTP-приложением, но в некоторых местах требует доработки. Прежде чем перейти к практике, давайте посмотрим, из чего состоят OTP-приложения.

11.1.1. Создание приложений с помощью инструмента mix

Приложение имеет характерную для OTP структуру. Описывающий приложение текстовый файл состоит из термов Erlang и называется *ресурсным файлом приложения* (создавать его вручную не придется – `mix` сделает это за вас). Данный файл содержит следующую информацию:

- имя, версия приложения и его описание;
- список модулей приложения;
- список зависимостей (которые тоже являются приложениями);
- модуль обратного вызова приложения (опционально).

Создав такой файл, вы можете использовать модуль *Application* для запуска и останова приложения. Соответствующий код этого модуля динамически загружает ресурсный файл (который должен находиться в каталоге загрузки) и запускает приложение. Это делается с помощью функции модуля обратного вызова `start/2`, которая сначала запускает все зависимости приложения, а потом само приложение.

Инструмент `mix` позволяет автоматизировать некоторые задачи, связанные с генерацией ресурсных файлов приложений. Например, при использовании `mix` список модулей приложения генерируется автоматически на основании модулей, определенных в исходном коде.

Имя приложения, его версия и описание вы, разумеется, должны указывать вручную. Используя эти данные и исходный код, инструмент mix сможет сгенерировать соответствующие ресурсные файлы на этапе компиляции проекта.

Рассмотрим все это на примере. Откройте временную папку и выполните команду `mix new hello_world --sup`. После этого создастся папка hello_world с минимально необходимым каркасом проекта `mix`. Параметр `--sup` дает инструменту `mix` указание сгенерировать модуль обратного вызова приложения и запустить в нем пустой супервизор (без потомков).

Теперь откройте папку hello_world и выполните команду `iex -S mix`. Вы не заметите ничего необычного. Однако на самом деле `mix` автоматически запустит ваше приложение, что можно легко проверить, вызвав функцию `Application.started_applications/0`:

```
iex(1)> Application.started_applications()
[
  {:hello_world, 'hello_world', '0.1.0'},   <-- Это приложение запущено
  {:logger, 'logger', '1.7.3'},
  {:mix, 'mix', '1.7.3'},
  {:iex, 'iex', '1.7.3'},
  {:elixir, 'elixir', '1.7.3'},
  {:compiler, 'ERTS  CXC 138 10', '7.2.4'},
  {:stdlib, 'ERTS  CXC 138 10', '3.5.1'},
  {:kernel, 'ERTS  CXC 138 10', '6.0.1'}
]
```

Можно видеть, что приложение `hello_world` запущено, а вместе с ним запущены и такие дополнительные приложения, как `mix`, `iex`, `elixir` со стороны Elixir, а также `stdlib` и `kernel` со стороны Erlang.

Совсем скоро вы поймете, для чего все это нужно, а сейчас поговорим об описании приложений. Обычно приложения описывают в файле mix.exs. Вот так выглядит содержимое сгенерированного файла (комментарии опущены):

```
defmodule HelloWorld.MixProject do
  use Mix.Project
```

```
  def project do
    [
      app: :hello_world,
      version: "0.1.0",
      elixir: "~> 1.7-rc",
      start_permanent: Mix.env() == :prod,
      deps: deps()
    ]
  end

  def application do
    [
      extra_applications: [:logger],
      mod: {HelloWorld.Application, []}
    ]
  end

  defp deps do
   []
  end
end
```

Описание проекта

Описание приложения

Список зависимостей

В функции `project/0`, в которой описывается проект `mix`, происходит кое-что интересное. В выражении `app: :hello_world`: приложению дается имя. Оно может быть выражено только атомом, и вы можете использовать его во время выполнения приложения для его останова.

Само приложение описывается в функции *`application/0`*. Здесь необходимо указать несколько опций, которые затем попадут в ресурсный файл приложения. В данном случае описание включает список других приложений на Elixir и Erlang, от которых зависит текущее приложение, а также запускающий приложение модуль. По умолчанию указывается Elixir-приложение `:logger` (**https://hexdocs.pm/logger/Logger.html**).

И последняя функция `deps` возвращает список сторонних библиотек, используемых в данном проекте. По умолчанию этот список пуст, а как использовать зависимости, вы узнаете чуть позже.

OTP-приложения и проекты mix

Стоит отметить, что OTP-приложение – это конструкция времени выполнения, ресурсный файл, интерпретируемый динамически соответствующим OTP-кодом. Во время использования mix какие-то особенности этого файла вы указываете вручную, а какие-то инструмент автоматически получает из вашего кода. Но само приложение приобретает смысл только на этапе выполнения.

Проект `mix` же, напротив, является конструкцией этапа компиляции. В файле mix.exs вы описываете свое приложение и реализуете модули, а затем на этапе компиляции создается ресурсный файл приложения.

11.1.2. Поведение приложения

Наиболее важной частью описания приложения является выражение `mod: {HelloWorld, []}`, указываемое в функции `application/0`. В нем задается модуль, который

следует использовать для запуска приложения. Запуская приложение, вы вызываете функцию HelloWorld.Application.start/2 с теми аргументами, которые будут указаны на месте второго элемента кортежа (в данном случае []).

Как вы уже догадались, необходимо реализовать модуль HelloWorld.Application, и это делается с помощью инструмента mix следующим образом:

```
defmodule HelloWorld.Application do
  use Application ◄──────── Используется модуль Application

  def start(_type, _args) do ◄──────── Функция обратного вызова
    children = []
    opts = [strategy: :one_for_one, name: HelloWorld.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```

Запуск супервизора первого уровня

Любое приложение – это поведение OTP, реализованное на основе модуля Application (**https://hexdocs.pm/elixir/Application.html**), являющегося оберткой над модулем Erlang :application (**http://erlang.org/doc/apps/kernel/application.html**). Прежде чем работать с модулем Application, необходимо реализовать модуль обратного вызова и определить несколько функций обратного вызова.

Модуль обратного вызова как минимум должен содержать функцию start/2. Этой функции передаются два аргумента – тип запускаемого приложения (обычно игнорируется) и произвольный аргумент (терм, указанный в файле mix.exs под ключом mod).

Задача функции обратного вызова start/2 – запустить основной процесс системы, который, как правило, является супервизором. Функция возвращает кортеж {:ok, pid} в случае успеха и {:error, reason} в случае неудачи.

11.1.3. Запуск приложения

Запустить приложение в текущем экземпляре BEAM можно с помощью функции *Application.start/1*. Данная функция выполняет поиск ресурсного файла приложения (сгенерированного инструментом mix) и интерпретирует его содержимое. После этого она проверяет, все ли зависимости запущены. И наконец, запускает приложение с помощью функции обратного вызова start/2. Также доступна функция Application.ensure_all_started/2, рекурсивно запускающая все те зависимости, которые еще не были запущены.

Обычно вызывать эти функции не требуется, поскольку mix автоматически запускает приложение, реализованное проектом. Команда iex -S mix автоматически запускает приложение со всеми его зависимостями:

```
$ iex -S mix
iex(1)> Application.start(:hello_world)
{:error, {:already_started, :hello_world}}
```

Имейте в виду, что вы не можете запустить несколько экземпляров одного приложения. В этом смысле в конкретном экземпляре BEAM данное приложение является синглтоном. При попытке запуска уже запущенного приложения вы получите ошибку.

Остановить приложение можно с помощью функции *Application.stop/1*:

```
iex(2)> Application.stop(:hello_world)
:ok
[info] Application hello_world exited: :stopped
```

Останов приложения заключается в завершении его основного процесса. Это отлично сочетается с использованием деревьев супервизоров, рассмотренных в девятой главе. Если все процессы вашей системы находятся под наблюдением супервизоров и основной процесс системы – тоже супервизор, то при завершении основного процесса завершаются и все остальные процессы, созданные приложением.

Функция `Application.stop/1` останавливает только указанное приложение, при этом его зависимости (тоже приложения) остаются запущенными. Остановить всю систему целиком можно с помощью функции `System.stop/0`. Она завершает все OTP-приложения и сам экземпляр BEAM. Обе функции `Application.stop/1` и `System.stop/0` поддерживают мягкий выход: каждому процессу дерева супервизоров дается время на освобождение памяти в его функции обратного вызова `terminate/1`, как объяснялось в разделе 9.1.6.

11.1.4. Библиотечные приложения

Опцию `mod: …` в функции `application/0` файла mix.exs можно не указывать:

```
defmodule HelloWorld.Application do
  ...

  def application do
    []
  end

  ...
end
```

У данного приложения нет модуля обратного вызова, а значит, и нет основного процесса. Как ни странно, это приложение все равно считается полноценным OTP-приложением. Его можно точно так же запускать и останавливать.

Зачем нужны такие приложения? Этот подход обычно используется при работе с *библиотечными приложениями* – компонентами, для которых нет необходимости создавать дерево супервизоров. Очевидно, эти приложения проще обычных. Классическим примером является парсер JSON. Erlang-приложение `stdlib` (**http://erlang.org/doc/apps/stdlib/index.html**) также является библиотечным, поскольку оно предоставляет различные вспомогательные модули, но не требует наличия дерева супервизоров.

Библиотечные приложения можно указывать в списке зависимостей времени выполнения. Это играет важную роль во время сборки готового к развертыванию релиза, и вы увидите это в главе 13.

11.1.5. Создание приложения текущей системы

Вооружившись новыми знаниями, вы можете превратить свою систему в полноценное приложение. Как отмечалось ранее, она уже является OTP-приложением,

пусть и библиотечным, так как не реализует модуль обратного вызова для поведения приложения.

Принимая во внимание, что текущая система запускает некоторое количество процессов в составе дерева супервизоров, будет очень кстати превратить ее в полноценное OTP-приложение. Первым же преимуществом этого решения будет упрощенный запуск системы. Реализовав модуль обратного вызова приложения, вы сможете добиться автоматического запуска системы при выполнении команды `iex -S mix`.

В первую очередь вам понадобится файл mix.exs. Напоминаю, что в главе 7 вы создали свой проект с помощью инструмента `mix`. Соответственно, этот файл у вас уже есть, и вам лишь нужно добавить в него кое-какую информацию. Полное содержимое файла mix.exs приведено в следующем листинге.

Листинг 11.1 ❖ Задание параметров приложения (todo_app/mix.exs)

```
defmodule Todo.MixProject do
  use Mix.Project

  def project do
    [
      app: :todo,
      version: "0.1.0",
      elixir: "~> 1.7",
      start_permanent: Mix.env() == :prod,
      deps: deps()
    ]
  end

  def application do
    [
      extra_applications: [:logger],
      mod: {Todo.Application, []}
    ]
  end

  defp deps do
    []
  end
end
```

Указание модуля обратного вызова

Вам необходимо внести всего одно изменение в функцию `application/0`, указав в ней модуль обратного вызова.

Теперь приступим к реализации этого модуля. Она будет совсем не сложной.

Листинг 11.2 ❖ Реализация модуля приложения (todo_app/lib/todo/application.ex)

```
defmodule Todo.Application do
  use Application

  def start(_, _) do
    Todo.System.start_link()
  end
end
```

Как отмечалось ранее, запуск приложения сводится к запуску супервизора первого уровня. Учитывая, что ваша система уже находится под наблюдением супер-

визора `Todo.System`, она и так является полноценным OTP-приложением, и больше никаких действий не требуется.

Давайте посмотрим на нее в действии:

```
$ iex -S mix

Starting database worker 1
Starting database worker 2
Starting database worker 3
Starting to-do cache
```

Автоматический запуск системы

Превратив систему в OTP-приложение, вы получаете возможность ее автоматического запуска с помощью всего лишь одной команды.

Стоит также отметить, что такая же возможность характерна и во время запуска тестов. Выполнив команду `mix test`, вы запустите все необходимые процессы своей системы. В главе 7 после добавления пары тестов вам приходилось запускать кеш-процесс вручную:

```
defmodule TodoCacheTest do
  use ExUnit.Case

  test "server_process" do
    {:ok, cache} = Todo.Cache.start()
    bob_pid = Todo.Cache.server_process(cache, "bob")

    assert bob_pid != Todo.Cache.server_process("alice")
    assert bob_pid == Todo.Cache.server_process("bob")
  end

  ...
end
```

Кеш запускается вручную

С тех пор код претерпел массу изменений, но в каждой новой версии было что-то от предыдущей. В модуле `TodoCacheTest` вы вручную запускали кеш-процесс, но теперь, когда система представляет собой OTP-приложение, это больше не потребуется. В следующем листинге можно видеть новую реализацию модуля `TodoCacheTest`.

Листинг 11.3 ❖ Тестирование запроса server_process (todo_app/test/todo_cache_test.exs)

```
defmodule TodoCacheTest do
  use ExUnit.Case

  test "server_process" do
    bob_pid = Todo.Cache.server_process("bob")

    assert bob_pid != Todo.Cache.server_process("alice")
    assert bob_pid == Todo.Cache.server_process("bob")
  end

  ...
end
```

Автоматический запуск кеш-процесса

11.1.6. Структура каталогов приложения

Уделим немного времени изучению структуры каталогов скомпилированного приложения. Благодаря инструменту `mix` она создается автоматически, но все же иногда полезно иметь хотя бы общее представление о ней.

Окружения mix

Прежде чем перейти к структуре каталогов приложения, поговорим немного об *окружениях* `mix` – опции этапа компиляции, с помощью которой можно влиять на вид скомпилированного кода.

У проектов mix имеется три возможных окружения: dev, test и prod. Каждое из них вносит определенные изменения в скомпилированный код. Например, в скомпилированную для дальнейшей разработки версию кода (окружение dev) будет нелишним добавить дополнительное журналирование для отладки, тогда как в промышленной версии (окружение prod) оно не требуется. В версии для тестирования (окружение test), для того чтобы данные тестов не перегружали основную базу, может понадобиться отдельная база данных.

Для большинства задач `mix` по умолчанию установлено окружение dev, означающее стадию разработки. Исключение составляют только тесты: после выполнения команды `mix test` устанавливается окружение test.

Задать окружение `mix` можно с помощью переменной окружения ОС `MIX_ENV`. При создании промышленных приложений следует использовать окружение prod. Чтобы скомпилировать код специально для этого окружения, выполните выражение `MIX_ENV=prod mix compile`, а для компиляции и запуска промышленной версии кода – выражение `MIX_ENV=prod iex -S mix`.

Структура скомпилированного кода

После компиляции проекта скомпилированные бинарные файлы помещаются в папку _build/ProjectEnv, где ProjectEnv – это окружение mix, активное на этапе компиляции.

Так как по умолчанию окружением `mix` является dev, то, выполнив команды `mix compile` или `iex -S mix`, вы получите бинарные файлы в папке _build/dev. Для OTP характерна следующая структура каталогов:


```
lib/
  App1/
    ebin/
    priv/

  App2/
    ebin/
    priv/
  ...
```

Обозначения App1 и App2 данной структуры обозначают имена приложений (например, `todo`). Папка ebin содержит скомпилированные бинарные файлы (файлы с расширением .beam и ресурсные файлы приложения), а в папке priv находятся приватные файлы приложения (изображения, скомпилированные бинарные файлы на языке C и т. д.). Эту структуру не обязательно соблюдать, но такое соглашение используется в большинстве проектов Elixir/Erlang, включая сами языки и некоторые инструменты.

К счастью, вам не требуется поддерживать эту структуру, так как инструмент `mix` делает это автоматически. Конечная структура каталогов скомпилированного проекта mix имеет следующий вид:

```
YourProjectFolder
  _build
    dev
      lib
        App1
          ebin
          priv

        App2
        ...
```

Помимо файлов самого приложения, папка lib содержит также и файлы его зависимостей этапа компиляции. Прочие зависимости времени выполнения (стандартные приложения Elixir/Erlang) находятся где-то на диске и подключаются к проекту через пути загрузки.

Как отмечалось ранее, ресурсный файл приложения с именем YourApp.app находится по адресу lib/YourApp/ebin. Путь к файлу текущей системы (относительно корневого каталога) будет следующим: _build/dev/lib/todo/ebin/. Когда вы запускаете приложение, его обобщенное поведение выполняет поиск ресурсного файла по путям загрузки (используются те же пути, что и для поиска скомпилированных бинарных файлов).

Это вся основная информация о приложениях. Теперь у вас достаточно теоретических знаний, чтобы перейти к изучению зависимостей.

Развертываемые системы

Приложения играют важную роль при создании готовых к развертыванию систем. Это будет подробно рассмотрено в главе 13 при обсуждении OTP-релизов. В двух словах: смысл заключается в том, чтобы разработать небольшую автономную систему, включающую только необходимые приложения и среду выполнения Erlang. Для этого необходимо превратить ее в OTP-приложение, потому что только тогда к ней можно будет подключить другие приложения в виде зависимостей. Затем все эти приложения объединяются вместе в один развертываемый релиз.

Поэтому любой повторно используемый код следует реализовывать в виде приложения. Это актуально и для сборок самих языков Elixir/Erlang. В комплект приложения `elixir` входит стандартная библиотека, а также отдельные приложения `iex` и `mix`, а Erlang разделен на несколько приложений, среди которых можно выделить `kernel` и `stdlib` (**http://erlang.org/doc**).

11.2. Работа с зависимостями

Использование сторонних библиотек нельзя оставлять без внимания. Когда вы начнете работать над более сложными проектами, вам, скорее всего, потребуются разного рода библиотеки вроде веб-фреймворков, парсеров JSON и драйверов базы данных. Они помогут решить различные второстепенные проблемы, предоставив вам возможность сконцентрировать свое внимание на основных задачах системы.

Например, в текущей реализации системы имеется небольшой пул рабочих процессов базы данных, позволяющий добиться управляемого параллельного вы-

полнения операций с базой данных. Этот пул вы реализовали сами еще в седьмой главе. Так случилось, что управление пулом процессов довольно часто используется при разработке на Elixir и Erlang, а потому специально для этого есть парочка сторонних библиотек. В данной главе вы замените свою примитивную реализацию пула процессов на проверенную на практике библиотеку.

11.2.1. Добавление зависимости

Добавим в список зависимостей в файле mix.exs стороннюю библиотеку Poolboy (**https://github.com/devinus/poolboy**), предоставляющую полноценную реализацию пула процессов, как показано в следующем листинге.

Листинг 11.4 ❖ Добавление внешней зависимости (todo_poolboy/mix.exs)

```
defmodule Todo.MixProject do
  ...

  defp deps do
    [
      {:poolboy, "~> 1.5"}   <-------- Указание внешней зависимости
    ]
  end
end
```

Внешняя зависимость указывается в виде кортежа. Первый его элемент – это всегда атом, соответствующий имени приложения этой зависимости, а второй – требуемая версия. В данном случае запись `"~> 1.5"` означает, что приложение должно быть версии 1.5 и выше. Если вас интересует более подробная информация о синтаксисе версий, загляните в следующий раздел официальной документации: **https://hexdocs.pm/elixir/Version.html#module-requirements**.

Указав необходимые сторонние библиотеки, их необходимо загрузить, выполнив команду `mix deps.get` из командной строки. Зависимости загружаются с помощью менеджера пакетов Hex (**http://hex.pm**). Зависимости также могут находиться в репозиториях GitHub, Git или в локальной папке. Подробнее об этом читайте в документации (**https://hexdocs.pm/mix/Mix.Tasks.Deps.html**).

Команда `mix deps.get` загружает все зависимости (рекурсивно) и сохраняет ссылки на конкретные версии каждой зависимости в файл mix.lock. Если этот файл уже существует, то старые данные в нем заменяются на актуальные. Это позволяет без проблем выполнять сборку на разных компьютерах, так что убедитесь, что файл mix.lock включен в системе контроля версий наряду с вашим проектом.

Загрузив необходимые зависимости, вы можете осуществить сборку всей системы, выполнив команду `mix compile`. Она скомпилирует все зависимости и сам проект. Несмотря на то что Poolboy – библиотека Erlang, `mix` все равно сможет ее скомпилировать.

11.2.2. Реорганизация пула процессов

Библиотека Poolboy довольно проста в использовании. Пулом рабочих процессов в ней управляет один процесс. При его запуске вы указываете необходимый размер пула (количество рабочих процессов) и модуль каждого из рабочих процессов.

После этого запускается менеджер пула, который, в свою очередь, запускает рабочие процессы в качестве своих потомков.

Другие процессы системы могут запрашивать у менеджера пула pid того или иного рабочего процесса. Эта операция называется *выдачей* (*checkout*). Как только клиентский процесс получит pid рабочего процесса, он сможет отправлять ему запросы. Если клиенту больше не нужна помощь рабочего процесса, он уведомляет об этом менеджера пула. Эта операция называется *возвратом* (*checkin*).

Данная схема немного сложнее, чем в вашей предыдущей реализации пула. Проделывая операции выдачи/возврата, менеджер отслеживает активность рабочих процессов. Если в пуле имеются свободные рабочие процессы, клиент может сразу взять себе один из них, в противном случае ему придется подождать, пока какой-нибудь процесс освободится.

Poolboy также использует мониторы и ссылки для обнаружения отказа клиентских процессов. Если один из клиентов забронирует рабочий процесс и неожиданно завершится, менеджер узнает об этом и вернет этот рабочий процесс обратно в пул. Если же аварийно завершится рабочий процесс, менеджер запустит вместо него новый.

Что ж, теперь можно приступить к реорганизации текущей реализации пула. Первым делом необходимо запустить менеджер пула как часть вашего дерева супервизоров. Это можно сделать с помощью функции *:poolboy.start_link*, но есть и чуть более элегантный способ. Можно вызвать функцию `:poolboy.child_spec/3`, описывающую параметры запуска библиотеки Poolboy, и тогда, чтобы превратить имеющуюся базу данных в пул на основе Poolboy, понадобится всего лишь изменить реализацию функции `Todo.Database.child_spec/1`. Код данного решения приведен в следующем листинге.

Листинг 11.5 ❖ Запуск пула процессов на основе библиотеки Poolboy (todo_poolboy/lib/todo/database.ex)

```
defmodule Todo.Database do
  @db_folder "./persist"

  def child_spec(_) do
    File.mkdir_p!(@db_folder)

    :poolboy.child_spec(
      __MODULE__,    <-------- ID потомка
      [
        name: {:local, __MODULE__},
        worker_module: Todo.DatabaseWorker,    | Параметры пула
        size: 3
      ],

      [@db_folder]    <-------- Аргументы рабочего процесса
    )
  end

  ...
end
```

Первый аргумент, передаваемый функции `:poolboy.child_spec/3`, – это ID потомка, необходимый для работы супервизора. В данном случае в качестве ID выступает имя модуля (`Todo.Database`).

Второй аргумент – параметры пула. Параметр :name обозначает, что процесс менеджера пула должен быть локально зарегистрирован, чтобы с ним можно было взаимодействовать, не зная его id. Опция :worker_module указывает модуль каждого из рабочих процессов, а опция :size – размер пула.

Последним аргументом функции :poolboy.child_spec/3 является список аргументов, передаваемых функции start_link каждого из рабочих процессов при их запуске.

Эти три аргумента указывают, что необходимо запустить три рабочих процесса на основе модуля Todo.DatabaseWorker. Менеджер пула запустит каждый из этих процессов вызовом функции Todo.DatabaseWorker.start_link(@db_folder).

Благодаря этому изменению функция Todo.Database.start_link больше не нужна, поскольку в новой спецификации указано, что база данных должна быть запущена с помощью функции :poolboy.start_link.

Теперь займемся функциями store и get модуля Todo.Database. Ранее эти функции получали ID рабочего процесса и затем передавали его соответствующим функциям модуля Todo.DatabaseWorker. Сделаем так, чтобы они доставали рабочий процесс из пула, выполняли запрос и возвращали этот процесс обратно в пул. Все это легко осуществляется с помощью функции *:poolboy.transaction/2*, как показано на примере в следующем листинге.

Листинг 11.6 ❖ Новые функции store и get (todo_poolboy/lib/todo/database.ex)

```
defmodule Todo.Database do
  ...

  def store(key, data) do
    :poolboy.transaction(                    ← Запрос одного рабочего процесса из пула
      __MODULE__,
      fn worker_pid ->
        Todo.DatabaseWorker.store(worker_pid, key, data)   | Выполнение операции
      end                                                  | с рабочим процессом
    )
  end

  def get(key) do
    :poolboy.transaction(
      __MODULE__,
      fn worker_pid ->
        Todo.DatabaseWorker.get(worker_pid, key)
      end
    )
  end
end
```

В данном примере вызывается функция :poolboy.transaction/2, принимающая зарегистрированное имя менеджера пула. Таким образом, отправляется запрос на выдачу одного рабочего процесса, и, как только он будет доступен, вызывается указанная анонимная функция. Когда функция завершит все свои действия, :poolboy.transaction/2 вернет рабочий процесс в пул.

Анонимная функция получает pid выданного рабочего процесса и выполняет запрос к процессу Todo.DatabaseWorker. Это означает, что текущую реализацию модуля Todo.DatabaseWorker необходимо немного подправить. Сейчас функции данно-

го модуля принимают ID рабочего процесса и выполняют поиск данного процесса в реестре. В новой версии обнаружение pid выполняется пулом, и рабочие процессы больше не требуется регистрировать или искать в реестре. Новая реализация показана в следующем листинге.

Листинг 11.7 ❖ Обновленные функции интерфейса рабочих процессов (todo_poolboy/lib/todo/database.ex)


```
defmodule Todo.DatabaseWorker do
  use GenServer

  def start_link(db_folder) do
    GenServer.start_link(__MODULE__, db_folder)
  end

  def store(pid, key, data) do
    GenServer.cast(pid, {:store, key, data})
  end

  def get(pid, key) do
    GenServer.call(pid, {:get, key})
  end

  ...
end
```

Это все изменения касательно пула процессов. Код клиентских модулей, модулей `Todo.System` и `Todo.Server`, а также код тестов остается прежним благодаря тому, что все подробности реализации «спрятаны» под функциями интерфейса.

11.2.3. Визуализация системы

Имея полноценное OTP-приложение, вы можете визуализировать его с помощью инструмента *observer*, входящего в стандартный пакет Erlang/OTP.

Посмотрим на это в действии. Запустите систему и создайте два сервера:

```
$ iex -S mix

iex(1)> Todo.Cache.server_process("Alice")
iex(2)> Todo.Cache.server_process("Bob")
```

Теперь запустите инструмент `observer`:

```
iex(3)> :observer.start()
```

Должно появиться окно с графическим интерфейсом, предоставляющее основную информацию о системе. Щелкнув на вкладку **Applications**, вы увидите дерево супервизоров своего приложения, как показано на рис. 11.1.

Два процесса первого уровня с pid <0.148.0> и <0.149.0> – это процессы, используемые OTP для управления приложением. Третий процесс с pid <0.150.0> – это супервизор первого уровня `Todo.System`. После него следуют три потомка: `Todo.Cache`, `Todo.Database` и `Todo.ProcessRegistry`.

Кеш-процесс имеет два дочерних процесса – это те самые серверы, что вы запустили из оболочки. Это можно легко проверить, щелкнув дважды на блоке про-

цесса и нажав кнопку **State** в появившемся окне. В последней строке можно видеть состояние процесса, содержащее имя списка дел.

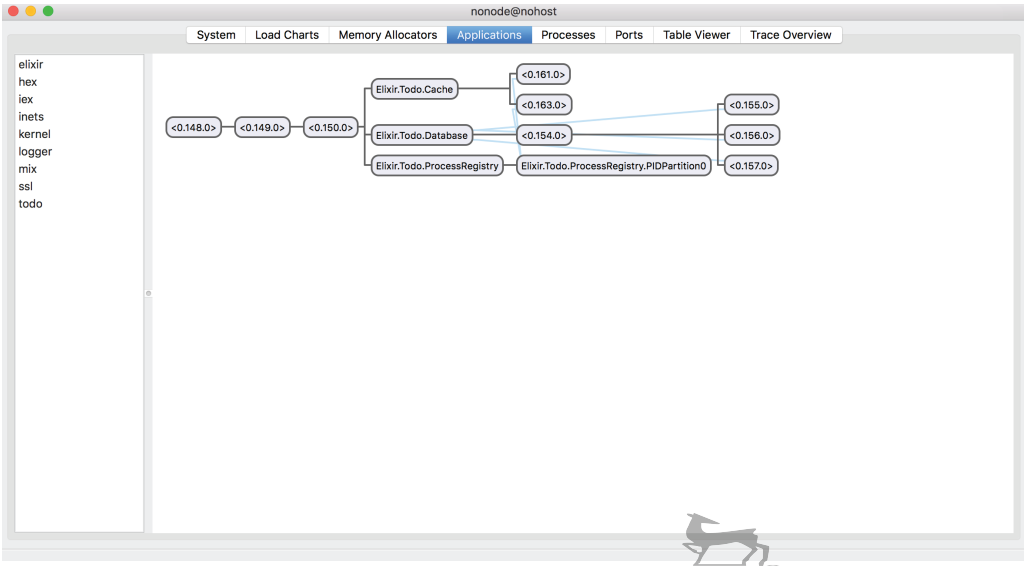


Рис.11.1 ❖ Визуализация приложения

Инструмент `observer` можно использовать и для визуализации поведения работающей системы. Например, на вкладке **Processes** находится список всех запущенных процессов системы, по которому очень легко определить, какие из процессов перегружены или потребляют слишком большое количество памяти. Эта вкладка часто используется для обнаружения узких мест системы. Также `observer` может показать работу системы на этапе промышленной эксплуатации. Это будет рассмотрено в главе 13.

11.3. Создание веб-сервера

Пришло время добавить в текущую систему HTTP-интерфейс. В этом разделе вы разработаете элементарный сервер, обслуживающий только запросы `entries` и `add_entry`.

Мы не будем акцентировать внимание на подробностях реализации веб-серверов и приведении их в идеальный вид. Главное для вас – научиться работать с OTP-приложениями и увидеть, как отдельные части этой простейшей модели соединяются в одну реальную систему.

11.3.1. Выбор зависимостей

Разумеется, вы могли бы разработать все составляющие веб-сервера самостоятельно, но это займет слишком много времени. Чтобы упростить себе жизнь, со-

ветую для этого воспользоваться парочкой существующих библиотек. Прошу заметить, что целью данного раздела является не рассказ об этих библиотеках во всех подробностях, а показ, как с их помощью можно достичь поставленной цели (разработать простейший рабочий HTTP-сервер). Интересующую вас информацию о данных библиотеках вам придется изучить самостоятельно.

Для разработки веб-серверов на Elixir и Erlang существуют различные фреймворки и библиотеки. Если вас интересует масштабная промышленная разработка, то абсолютно точно следует обратить внимание на фреймворк *Phoenix* (**http://phoenixframework.org/**). Это универсальный фреймворк с модульной структурой, идеально подходящий для создания различного рода HTTP-серверов. Работать с Phoenix впервые может быть затруднительно, поэтому для простоты далее мы будем использовать две HTTP-библиотеки более низкого уровня.

Первая – это Erlang-библиотека *Cowboy* (**https://github.com/extend/cowboy**). Это легковесная и эффективная библиотека, часто используемая для разработки HTTP-серверов на Elixir и Erlang. Мы не будем использовать ее напрямую, а подключим ее к проекту с помощью другой библиотеки *Plug* (**https://github.com/elixir-lang/plug**) – проекта команды разработчиков Elixir. Эта библиотека чем-то напоминает библиотеку Rack языка Ruby или Ring языка Closure. Она предоставляет универсальный API, абстрагирующий веб-библиотеку. Кроме того, она вводит понятие *плагов* – связующих модулей или функций, которые можно внедрять в конвейеры обработки запросов для выполнения действий на определенных этапах.

Данные библиотеки необходимо будет указать в списке зависимостей в файле mix.exs, как показано в следующем листинге.

Листинг 11.8 ❖ Внешние зависимости веб-сервера (todo_web/mix.exs)

```
defmodule Todo.Mixfile do
  ...

  defp deps do
    [
      {:poolboy, "~> 1.5"},
      {:cowboy, "~> 1.1"},
      {:plug, "~> 1.4"},
    ]
  end
end
```

Новые зависимости

11.3.2. Запуск сервера

Добавив зависимости, выполним команду `mix deps.get`, после чего приступим к реализации HTTP-интерфейса. Как отмечалось ранее, основным интерфейсом для работы с библиотекой Cowboy будет выступать Plug. Plug – довольно сложная библиотека, но в данном случае все ее особенности нас не интересуют. Необходимо освоить ее основные возможности и понять, как отдельные ее части складываются воедино.

Чтобы запустить сервер на основе Plug и Cowboy, используйте функцию *`Plug.Adapters.Cowboy.child_spec/1`*. Она возвращает спецификацию потомков, описывающую параметры запуска процессов системы, отвечающих за HTTP-сервер. Как

вам уже известно, лучше вынести эту часть в отдельный модуль, как показано далее в листинге.

Листинг 11.9 ❖ Спецификация HTTP-сервера (todo_web/lib/todo/web.ex)

```
defmodule Todo.Web do
  ...

  def child_spec(_arg) do
    Plug.Adapters.Cowboy.child_spec(
      scheme: :http,
      options: [port: 5454],
      plug: __MODULE__
    )
  end

  ...
end
```

Аргумент, передаваемый функции `Plug.Adapters.Cowboy.child_spec/1`, – это параметры сервера. В данном случае указываем, что весь HTTP-трафик должен обслуживаться портом 5454. Опция `:plug` означает, что для обработки запроса будет вызвана какая-либо функция из указанного модуля.

После добавления `Todo.Web` в список потомков супервизора в системе появится несколько новых процессов. Во-первых, будет как минимум один процесс, слушающий данный порт и принимающий запросы. Во-вторых, каждое TCP-подключение будет обрабатываться в отдельном процессе, и в них будут вызываться функции обратного вызова (которые вам еще предстоит реализовать). Однако вам не придется беспокоиться об этом: благодаря функции `child_spec/1` библиотека Plug маскирует все эти подробности реализации.

Имея готовую функцию `child_spec/1`, можно добавить HTTP-сервер в дерево супервизоров. Правильнее всего будет сделать его потомком главного супервизора `Todo.System`.

Листинг 11.10 ❖ Запуск HTTP-сервера (todo_web/lib/todo/system.ex)

```
defmodule Todo.System do
  def start_link do
    Supervisor.start_link(
      [
        Todo.ProcessRegistry,
        Todo.Database,
        Todo.Cache,
        Todo.Web    <-------- Новый потомок
      ],
      strategy: :one_for_one
    )
  end
end
```

Благодаря вынесению частей системы в отдельные модули и грамотному именованию можно легко понять, что система состоит из реестра процессов, базы данных, кеша и веб-сервера.

ПРИМЕЧАНИЕ Помните, что приложения являются синглтонами – в текущем экземпляре BEAM запустить можно только один экземпляр конкретного приложения. Но это не значит, что в системе может быть только один веб-сервер. Приложения Plug и Cowboy можно считать создателями HTTP-серверов. Сразу после запуска этих приложений ничего не происходит, но, используя функцию `child_spec/1`, вы можете добавить HTTP-сервер в любое место дерева супервизоров, и система может иметь несколько таких серверов. Например, вы могли бы добавить отдельный сервер для администрирования.

11.3.3. Обработка запросов

Добавим в реализацию сервера возможность обработки нескольких типов запросов. Начнем с запроса на добавление записи `add_entry`. Это будет запрос типа POST, но для простоты все параметры будут передаваться по URL. Запрос будет выглядеть примерно так:

```
http://localhost:5454/add_entry?list=bob&date=2018-12-19&title=Dentist
```

Добавим маршрут для данного запроса, как показано ниже.

Листинг 11.11 ❖ Определение маршрута запроса add_entry (todo_web/lib/todo/web.ex)

```
defmodule Todo.Web do
  use Plug.Router          ←——— Добавление шаблонного кода

  plug :match
  plug :dispatch           | Установка плагов

  ...

  post "/add_entry" do     ←——— Определение обработчика
    ...
  end

  ...
end
```

В данном коде присутствуют неизвестные вам конструкции, присущие модулю Plug. Я объясню их в общих чертах, но тратить время на их изучение не стоит, ведь цель данного упражнения заключается не в этом.

Вызов `use Plug.Router` автоматически добавляет в текущий модуль несколько определений функций точно так же, как это было при вызове `use GenServer` ранее. По большому счету, эти функции используются Plug на внутреннем уровне.

Выражения `plug :match` и `plug :dispatch` заслуживают особого внимания. На этапе компиляции они проделают некоторые действия, которые позволят соответствовать различным HTTP-запросам. Данные выражения являются примерами вызовов макросов Elixir, выполняющихся во время компиляции. В результате в текущий модуль будет добавлено несколько дополнительных функций (которые опять же необходимы для внутренней работы Plug).

Для определения обработчика запросов используется макрос `post`. Он работает аналогично макросу `test`, который был рассмотрен в главе 7. Под капотом этот макрос генерирует функцию, используемую всеми теми функциями, добавленными после вызовов макросов `plug` и `use Plug.Router`. Она будет выглядеть следующим образом:

```
defp do_match("POST", ["add_entry"]) do
  fn(conn) ->
    ...   <-- Код, передаваемый макросу post
  end
end
```

Сгенерированная функция `do_match` вызывается другим кодом, добавленным в ваш модуль после выполнения выражений `use Plug.Router` и `plug :match`.

Если вызвать макрос `post` несколько раз, то вы получите несколько экземпляров функции `do_match`, каждый из которых будет соответствовать определенному маршруту. У вас получится примерно следующее:

```
defp do_match("POST", ["add_entry"]), do: ...
defp do_match("POST", ["delete_entry"]), do: ...
...
defp do_match(_, _), do: ...   <-- Обработка всех остальных запросов
```

Таким образом, код, следующий после выражения `post "/add_entry"`, выполняется тогда, когда на сервер приходит HTTP-запрос типа POST, имеющий путь `/add_entry`.

А теперь перейдем к обработчику запросов.

Листинг 11.12 ❖ Реализация запроса add_entry (todo_web/lib/todo/web.ex)

```
defmodule Todo.Web do
  ...

  post "/add_entry" do
    conn = Plug.Conn.fetch_query_params(conn)
    list_name = Map.fetch!(conn.params, "list")
    title = Map.fetch!(conn.params, "title")
    date = Date.from_iso8601!((Map.fetch!(conn.params, "date"))   <-- Расшифровка входных параметров

    list_name
    |> Todo.Cache.server_process()
    |> Todo.Server.add_entry(%{title: title, date: date})   <-- Выполнение операции
    conn
    |> Plug.Conn.put_resp_content_type("text/plain")
    |> Plug.Conn.send_resp(200, "OK")   <-- Отправка ответа
  end

  ...
end
```

Обратите внимание, что в обработчике запросов используется переменная `conn`, которую вы нигде не объявляли. Макрос `post` генерирует эту переменную и привязывает к ней необходимое значение.

Как вы могли догадаться, переменная `conn` отвечает за хранение данных о соединении. Она представляет собой экземпляр структуры `Plug.Conn`, содержащей TCP-сокет и информацию о состоянии обрабатываемого запроса. Обработчик должен возвращать обновленное соединение, содержащее информацию об ответе (его тело и статус).

Реализация обработчика запроса состоит из трех частей: расшифровка входных параметров, выполнение кода, добавляющего новую запись, и отправка ответа клиенту.

Входные параметры преобразуются с помощью функции `Plug.Conn.fetch_params/1`, возвращающей новую версию структуры соединения, поле `params` которой содержит параметры запроса (в виде словаря). По сути, происходит кеширование данных: Plug сохраняет результат функции `fetch_params` в структуру соединения, что позволит избежать излишней работы парсера при многочисленных вызовах `fetch_params`.

Подготовив параметры, можно выполнить запрос и добавить запись, после чего определить тип содержимого ответа, его тело и статус.

Запросы типа `entries`, реализацию которых можно увидеть в следующем листинге, обрабатываются похожим образом.

Листинг 11.13 ❖ Реализация запроса entries (todo_web/lib/todo/web.ex)

```
defmodule Todo.Web do
  ...

  get "/entries" do
    conn = Plug.Conn.fetch_query_params(conn)
    list_name = Map.fetch!(conn.params, "list")
    date = Date.from_iso8601!((Map.fetch!(conn.params, "date"))

    entries =
      list_name
      |> Todo.Cache.server_process()
      |> Todo.Server.entries(date)

    formatted_entries =
      entries
      |> Enum.map(&"#{&1.date} #{&1.title}")
      |> Enum.join("\n")

    conn
    |> Plug.Conn.put_resp_content_type("text/plain")
    |> Plug.Conn.send_resp(200, formatted_entries)
  end

  ...
end
```

Как видите, в данном примере используется тот же подход, что и с `add_entry`. Единственная поправка состоит в том, что `entries` – это запрос типа GET, поэтому вместо макроса `post` здесь используется макрос `get`.

В данной реализации сначала производится расшифровка параметров и выборка запрашиваемых записей. Затем задается текстовый тип содержимого ответа и клиенту возвращается ответ.

> **ПРИМЕЧАНИЕ** В сообществе Elixir такой подход к обработке HTTP-запросов получил название «Вы не Phoenix разрабатываете» (Phoenix is not your Application). Эта фраза принадлежит Лансу Хальворсену (Lance Halvorsen) и означает, что HTTP-сервер следует расценивать лишь как интерфейс основной части системы. Обратите внимание, что в обоих запросах входные данные преобразуются в определенные типы и затем для выполнения операции вызывается не относящийся к HTTP код. Для основного кода обработчиков запросов не имеет значения то, что он запускается в контексте HTTP-запроса. Данный подход позволяет разделить функциональность, улучшить тестируемость и упрощает добавление дополнительных интерфейсов системы.

Теперь вы можете запустить систему командой `iex -S mix` и отправить несколько запросов. Вот что получается с использованием инструмента командной строки `curl`:

```
$ curl -d "" \
  "http://localhost:5454/add_entry?list=bob&date=2018-12-19&title=Dentist"
OK

$ curl "http://localhost:5454/entries?list=bob&date=2018-12-19"
2018-12-19 Dentist
```

Все работает, но давайте подробнее разберем взаимодействие отдельных частей системы.

11.3.4. Логика работы системы

Начнем с HTTP-сервера. Основная идея показана на рис. 11.2.

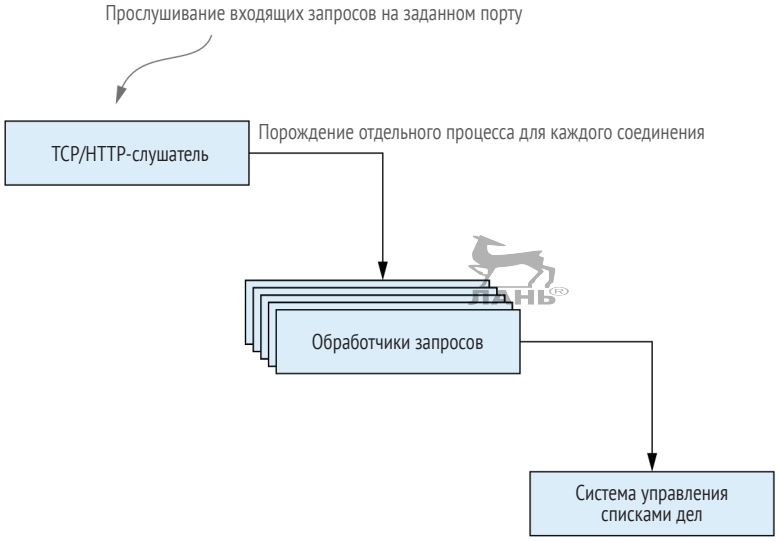


Рис. 11.2 ❖ Обработка запросов в отдельных процессах

Главное, что стоит отметить, – это то, что каждому подключению выделяется отдельный процесс. На практике это означает, что разные запросы обрабатываются в разных процессах. Именно так устроен веб-сервер Cowboy: он использует один процесс для прослушивания порта и порождает отдельные процессы для каждого из поступающих запросов.

Такая архитектура имеет множество преимуществ еще и благодаря тому, как виртуальная машина BEAM воспринимает процессы. Поскольку процессы BEAM конкурентны, ресурсы ЦП используются по максимуму, а система имеет способность к масштабированию. Процессы BEAM легковесны, а значит, можно без труда управлять большим количеством одновременных подключений. Более того, благодаря вытесняющей природе планировщика BEAM случайные процессы с дли-

тельным временем выполнения или запросы, поглощающие значительную часть ресурсов ЦП, не будут блокировать всю систему. И наконец, изолированность процессов не позволит одному неудачному запросу каким-либо образом повлиять на остальные части системы.

С помощью процессов также довольно легко понять логику работы системы. Например, вы можете быть уверены, что независимые запросы не будут блокировать друг друга, а многочисленные запросы к одному и тому же списку дел будут упорядочены, как проиллюстрировано на рис. 11.3.

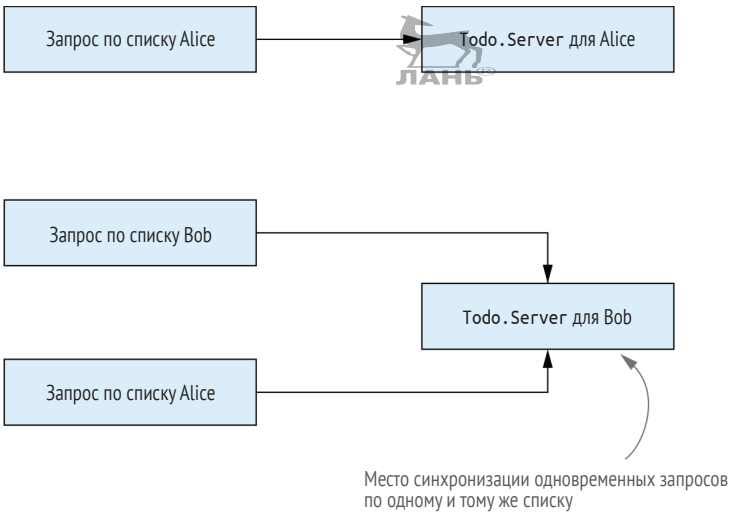


Рис. 11.3 ❖ Конкурентная обработка независимых одновременных запросов и последовательная обработка нескольких запросов к одному и тому же списку

Это происходит благодаря тому, как устроена работа кеша. Если клиент хочет выполнить какие-либо действия со списком Bob, сначала он запрашивает у кеша pid процесса списка. Все запросы, касающиеся списка Bob, приходят одному и тому же процессу, а значит, они выполняются в порядке очереди.

Производительность

Мы проделали достаточно большое количество изменений кода, и пришло время измерить производительность системы. Я провел быстрый тест с помощью инструмента `wrk` (**https://github.com/wg/wrk**). 30-секундный нагрузочный тест на моем компьютере показал, что текущая система обрабатывает 25 000 запросов в секунду со средней задержкой в 0,90 мс и 99-м процентилем в 5 мс. Пока тест выполнялся, все ядра ЦП были по максимуму нагружены, и это доказывает, что в системе нет узких мест. Текущая система является высококонкурентной и использует все предоставленные ей ресурсы.

Полученные результаты довольно неплохие, учитывая, что некоторые решения по реализации, откровенно говоря, далеки от идеала. Вот несколько таких примеров:

- при каждом действии со списком сохраняется весь список целиком;

- при поиске данных в абстракции списка дел итерации выполняются со всем списком;
- каждая операция поиска проходит через динамический супервизор.

Как видите, нам еще есть над чем поработать, но приятно осознавать, что система показывает хорошую производительность без каких-либо дополнительных усилий.

Запросы call и cast

Давайте выясним, какое влияние оказывают на систему запросы типа call и cast. Напоминаю, cast-запросы работают по принципу «запустили и забыли»: вызывающий процесс отправляет сообщение серверу и незамедлительно приступает к выполнению других задач. Call-запросы, напротив, являются блокирующими, и вызывающий процесс не может ничего делать, пока не получит ответ.

Как вы помните, было принято, так сказать, спонтанное решение использовать cast-запросы для всех тех операций, которым не требуется возвращать ответ. В реальности эти запросы имеют большой недостаток: результат запроса неизвестен. В свою очередь, это означает, что конечный пользователь может получить неактуальные данные, как показано на рис. 11.4.

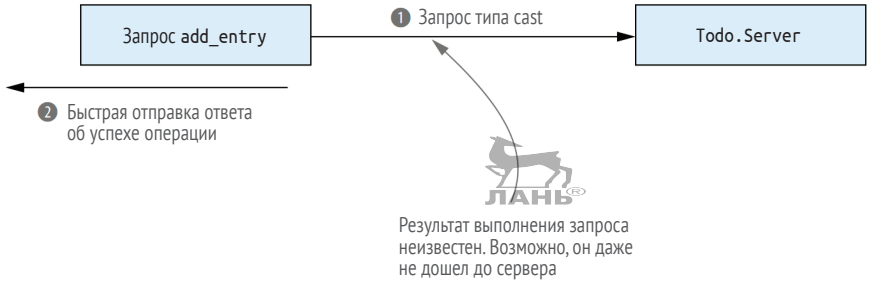


Рис. 11.4 ❖ Как использование cast-запросов влияет на достоверность ответа

Так как добавление записи в список – это запрос типа cast, результат его выполнения всегда неизвестен. Отправка пользователю сообщения об успехе – догадка, а никак не истина.

Как вы уже догадались, это можно исправить, заменив cast-запросы на синхронные call-запросы, как показано на рис. 11.5.

Синхронные запросы позволяют обеспечить целостность данных, потому что ответ об успешном выполнении операции возвращается, только если запись была сохранена в базе данных. Однако и у этого метода есть свой недостаток: теперь производительность всей системы зависит от времени отработки рабочих процессов базы данных. Рабочих процессов в текущей реализации всего три, да и существующая база данных далеко не самая эффективная.

Проблему можно решить с помощью добавления промежуточного процесса. Основная идея состоит в том, чтобы предоставлять клиенту быстрый ответ, сообщая ему, что запрос был поставлен в очередь, а затем обрабатывать этот запрос и уведомлять клиента о его статусе (см. рис. 11.6).

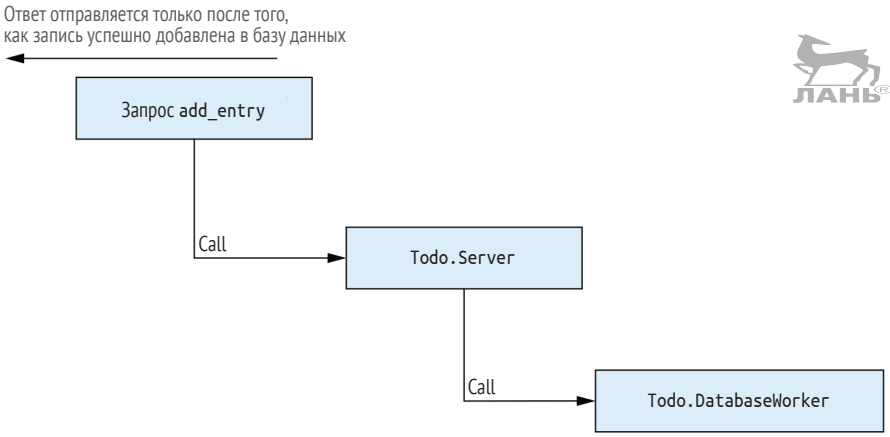


Рис 11.5 ❖ Использование синхронных запросов обеспечивает целостность данных, но снижает отзывчивость системы

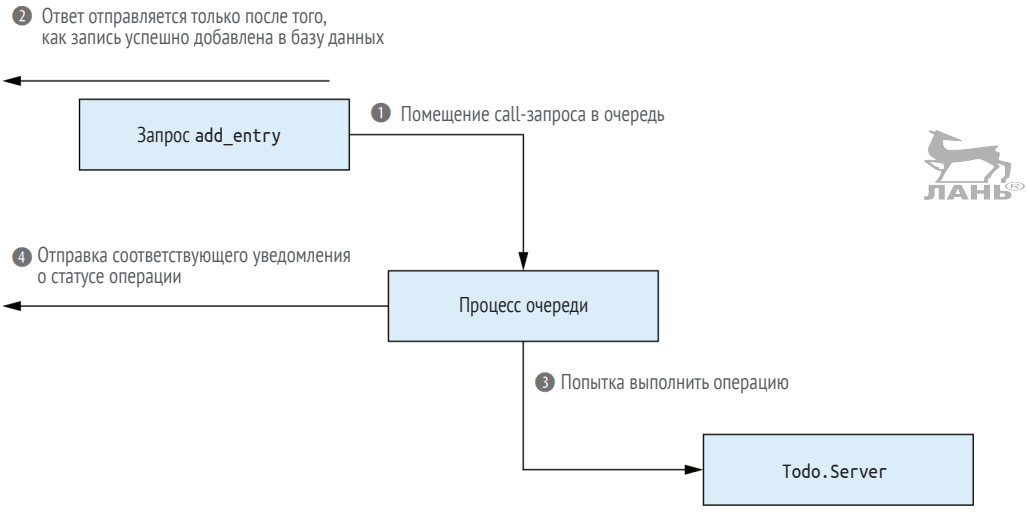


Рис. 11.6 ❖ Постановка запроса в очередь и отправка уведомления о его статусе

Эта схема уже гораздо более продуманная и сложная. В элементарных случаях будет достаточно использовать простые синхронные запросы, тогда как в случаях чрезмерно высокой нагрузки, когда выполнение операций занимает большое количество времени, лучше ввести в систему промежуточный процесс. Это позволит повысить отзывчивость системы и сохранить целостность данных. Более того, этот процесс поможет справиться с заторами и повышенной нагрузкой. Если система перегружена, а очередь запросов переполнена, то можно на время приостановить прием новых запросов от клиентов, пока ситуация не изменится.

Как это обычно бывает, нет универсального решения для всех случаев. Попробуйте начать с синхронных запросов для достижения целостности данных, а по-

том вы с легкостью сможете переключиться на cast-запросы или добавить промежуточный процесс, если того потребует конкретный случай.

Теперь можно сказать, что HTTP-сервер готов. Далее мы поговорим о настройке приложений.

11.4. Настройка приложений

Настройка OTP-приложения выполняется с помощью особой функции, называемой *окружением приложения*, которое представляет собой хранилище пар ключ/значение в оперативной памяти. Как ключи, так и значения этого хранилища должны быть выражены термами Elixir. Значения окружения можно задать в файлах конфигурации – скриптах Elixir, находящихся в папке config. Эти файлы считываются еще до компиляции и запуска проекта, и их единственной целью является хранение этих самых значений для окружений приложения. За загрузку и применение указанных в этих файлах настроек до запуска приложения отвечает инструмент `mix`. Получить какие-либо значения окружения в коде приложения можно с помощью функций модуля `Application`.

11.4.1. Окружение приложения

Рассмотрим простой пример. Текущий HTTP-сервер прослушивает порт 5454, заданный явным образом в коде. Попробуем установить HTTP-порт с помощью окружения приложения.

Наиболее часто используемый способ установки окружения приложения – это файл config/config.exs. В этом файле указываются различные настройки окружения, которые затем загружаются перед запуском OTP-приложения.

Настройка HTTP-порта приводится в следующем листинге.

Листинг 11.14 ❖ Настройка HTTP-порта (todo_env/config/config.exs)

```
use Mix.Config
```
← **Добавление вспомогательных функций**

```
config :todo, http_port: 5454
```
← **Добавление значения окружения приложения**

Файл `config.exs` содержит скрипт, выполняемый инструментом `mix` во время компиляции проекта и запуска приложения. Параметру `:http_port` приложения `:todo` присваивается значение 5454 с помощью макроса `config`. Как только вы запустите приложение, этот параметр станет доступен в окружении приложения, и его можно будет получить, например, с помощью функции `Application.get_env/2`:

```
$ iex -S mix
iex(1)> Application.get_env(:todo, :http_port)
5454
```

Теперь немного доработаем код `Todo.Web`, как показано в листинге ниже.

Листинг 11.15 ❖ Получение параметра http_port (todo_env/lib/todo/web.ex)

```
defmodule Todo.Web do
  ...
```

```
  def child_spec(_arg) do
    Plug.Adapters.Cowboy.child_spec(
      scheme: :http,
      options: [port: Application.fetch_env!(:todo, :http_port)],
        plug: __MODULE__
    )
  end

  ...
end
```

Получение HTTP-порта из окружения приложения

В данном случае используется функция `Application.fetch_env!`, возбуждающая ошибку, если HTTP-порт не сконфигурирован.

11.4.2. Изменяемость настроек

В некоторых случаях вам может понадобиться установить различные настройки для получения различных результатов после компиляции. Например, в текущей системе используется один и тот же HTTP-порт как для разработки, так и для тестирования. Как следствие, если система запущена в окружении разработки, тестирование провести не удастся.

Как вы помните, перед тем как выполнить тест, mix запускает все приложение. Веб-сервер этого приложения (`Todo.Web`) должен прослушивать порт 5454, но при запущенной сессии `iex -S mix` сервер `Todo.Web` не сможет связаться с тем же портом, и у вас не получится выполнить тесты.

Решить эту проблему можно, используя другой HTTP-порт в тестовом окружении `mix`. Таким образом, конфликт будет разрешен, и вы сможете запускать тесты, даже если система запущена в окружении разработки.

> **ПРИМЕЧАНИЕ** Термин «окружение» имеет два применения. Окружение приложения – это хранилище пар ключ/значение, содержащее различные настройки вашего OTP-приложения. Окружение `mix` определяет цель компиляции приложения – дальнейшая разработка, тестирование или промышленная эксплуатация.

Чтобы задать разные параметры для разных окружений `mix`, необходимо немного изменить файл config/config.exs, как показано далее.

Листинг 11.16 ❖ Настройки окружения `mix` (todo_env/config/config.exs)

```
use Mix.Config

config :todo, http_port: 5454

...

import_config "#{Mix.env()}.exs"
```

Переопределение окружения mix

Последняя строка данного листинга является самой важной. Выражение `import_config "#{Mix.env()}.exs"` осуществляет импорт параметров, установленных для текущего окружения `mix`. Это делается с помощью функции `Mix.env/0`, возвращающей окружение mix в виде атома (`:dev`, `:test` или `:prod`).

Функция `Mix.env/0` недоступна во время выполнения приложения

Функцию `Mix.env/0` не следует включать в основной код приложения, содержащий стандартные функции, компилируемые и вызываемые во время выполнения. Используйте эту функцию исключительно в скриптах конфигурации, в специальных задачах `mix` и в коде, выполняющемся на этапе компиляции (макросы Elixir).

В зависимости от окружения mix это выражение выполнит импорт конфигураций из файлов `config/dev.exs`, `config/test.exs` или `config/prod.exs`. Вам необходимо создать эти файлы, и они как минимум должны содержать выражение `use Mix.Config`. В каждом из этих файлов вы можете указать дополнительные параметры или переопределить какие-либо из существующих в `config.exs` настроек.

В файле config.exs должны находиться общие для всех окружений `mix` настройки, а в отдельных файлах вроде test.exs – настройки конкретных окружений.

Чтобы использовать для тестового окружения другой HTTP-порт, вам необходимо указать его в файле test.exs (см. листинг 11.17).

Листинг 11.17 ❖ Переопределение настроек (todo_env/config/test.exs)

```
use Mix.Config

config :todo, http_port: 5455
```

Давайте проверим, как это работает. Запустите приложение в стандартной среде dev:

```
$ iex -S mix

iex(1)> Application.get_env(:todo, :http_port)
5454
```

Попробуйте выполнить то же самое, но уже в среде test:

```
$ MIX_ENV=test iex -S mix

iex(1)> Application.get_env(:todo, :http_port)
5455
```

Теперь вы можете выполнять тесты, даже если система запущена в другом процессе ОС. Стоит отметить, что код, находящийся в папке todo_env, также создает новую папку базы данных с настройками окружения. Благодаря этому вы можете использовать другую папку базы данных в процессе тестирования, чтобы не перегружать основную базу данных.

11.4.3. Особенности скриптов конфигурации

Всегда помните о том, что скрипты с общими настройками (config.exs) и скрипты с настройками определенного окружения (dev.exs) выполняются перед компиляцией и запуском проекта. Это означает, что количество действий с этими скриптами сильно ограничено. Даже несмотря на то, что эти скрипты представляют собой обычный код на Elixir, в нем можно вызывать только функции стандартной

библиотеки Elixir и вспомогательные функции инструмента mix (например, `Mix.env/0`). Нельзя использовать функции ваших модулей или код зависимостей, потому что эти модули не доступны во время выполнения скриптов.

К тому же, если вы используете релизы OTP (о них будет рассказано в главе 13), в скриптах конфигурации вы не можете принимать решения во время выполнения программы. В этом случае скрипты конфигураций выполняются во время компиляции, и значения параметров вшиваются в OTP-релиз, и это, как правило, происходит не на том компьютере, на котором запущена система, а на том, где осуществляется сборка. Следовательно, скрипты конфигурации не получится использовать для предоставления параметров внешних источников данных (среды ОС, файлов ini, внешних баз данных, систем etcd или Vault), необходимых во время выполнения программы. Для получения этих параметров следует прописать соответствующие функции в обычном коде приложения, вызывающемся во время выполнения.

Создателям библиотек необходимо всегда помнить об этих ограничениях. Когда вы разрабатываете библиотеку, старайтесь не передавать эти параметры через окружение приложения, потому что в этом случае ее пользователям придется указывать их в скриптах конфигурации. Получить эти параметры во время выполнения будет довольно сложно. Гораздо более гибким подходом будет сделать так, чтобы API вашей библиотеки принимал все эти параметры в качестве аргументов функции. Это позволит пользователям выбирать, каким образом им удобнее задавать параметры – прописать их в коде, добавить с помощью окружения или получить из внешних источников во время выполнения. За более подробной информацией обратитесь к разделу «Library Guidelines» официальной документации Elixir (**https://hexdocs.pm/elixir/master/library-guidelines.html**).

Выводы

- Приложение OTP – это компонент, который может быть использован повторно. В его основе может лежать полноценное дерево супервизоров или вспомогательные модули (как у библиотечных приложений).
- В одном экземпляре BEAM может быть запущен только один экземпляр OTP-приложения.
- Небиблиотечное приложение представляет собой модуль обратного вызова, запускающий дерево супервизоров.
- Приложения позволяют указывать зависимости времени выполнения (другие приложения). Есть также и зависимости времени компиляции, с помощью которых можно получить доступ к внешнему коду и скомпилировать его из вашего проекта.
- Окружения позволяют установить настройки приложения. Используя различные окружения, можно задать разные настройки для разных сборок.

Глава 12

Создание распределенной системы

В главе рассматривается:

- работа с примитивами распределения;
- создание отказоустойчивого кластера;
- особенности сетевого соединения.

Вы разработали HTTP-сервер, и теперь самое время сделать его более надежным. Чтобы система была поистине надежной, необходимо изначально запускать ее на нескольких компьютерах. Если система запущена только на одном компьютере, то он будет считаться точкой отказа, потому что при выходе его из строя остановится вся система. Группа компьютеров, в свою очередь, позволяет системе оставаться в рабочем состоянии даже при выходе из строя отдельных машин.

Более того, использование нескольких компьютеров предоставляет возможность горизонтального масштабирования. При росте нагрузки на систему вы можете добавить дополнительные компьютеры, чтобы система могла справиться с нагрузкой. Это показано на рис.12.1.

Как видно по рисунку, в системе имеется несколько узлов, и при отказе одного из них оставшиеся за ним задачи распределяются по другим узлам, и система продолжает функционировать. При высоких нагрузках вы можете добавить дополнительные узлы. Клиенты взаимодействуют с определенным интерфейсом, и им неизвестны внутренние подробности устройства системы.

Распределенные системы обладают значительными преимуществами, а Elixir и Erlang предоставляют несколько простых, но эффективных примитивов для работы с ними. Основными инструментами для создания распределенных систем на основе Erlang являются процессы и сообщения. Вы можете отправить сообщение другому процессу независимо от того, где он запущен: в том же экземпляре BEAM, в другом экземпляре или на удаленном компьютере.

Не стоит путать это с классическим подходом к удаленному обмену данными, при котором удаленный вызов оборачивается в функцию и выглядит как локальный вызов. Erlang и, соответственно, Elixir идут другим путем, и распределенные вычисления вступают в игру намного раньше. На самом деле любую конкурентную систему, состоящую из большого количества процессов, уже можно считать распределенной.

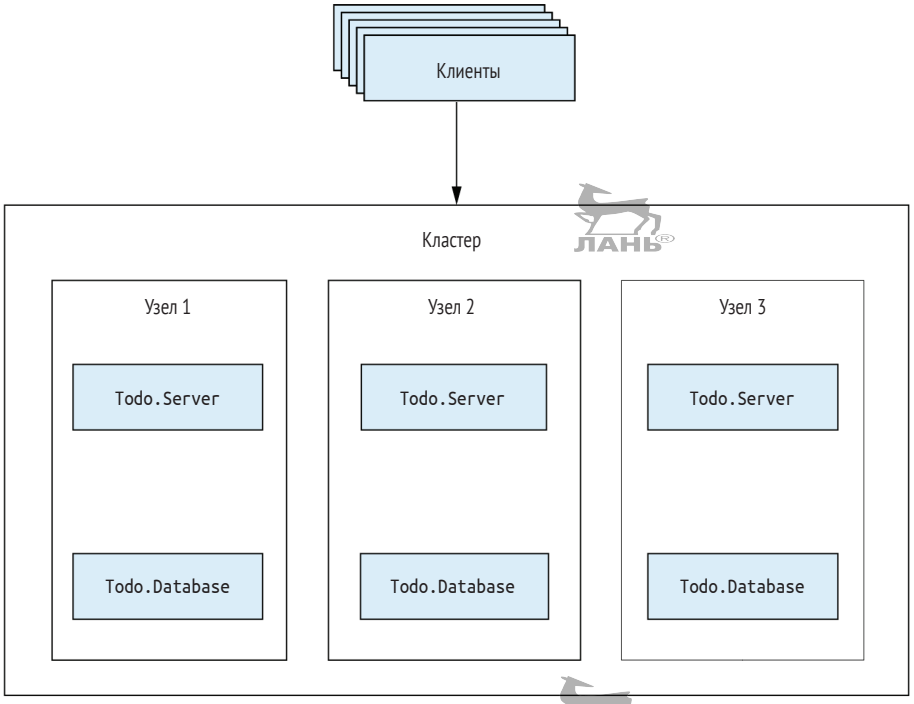


Рис. 12.1 ❖ Использование группы компьютеров

Подобно удаленным компонентам, процессы живут своей жизнью и работают полностью изолированно друг от друга. Отправку запроса от одного локального процесса другому можно считать удаленным вызовом, да и обмен сообщениями имеет много общего с сетевым соединением. По умолчанию при отправке сообщения вы не знаете ничего о его результате, и даже иногда не знаете, дошло ли оно до получателя. Если вам нужен гарантированный результат, вы можете реализовать протокол для отправки получателем ответа (например, с помощью синхронного вызова). Кроме этого, необходимо учитывать, что передача сообщений тратит ресурсы памяти (содержимое сообщений копируется), и в отдельных случаях это влияет на реализацию протокола обмена данными между процессами.

Все эти особенности характерны для конкурентной модели и распределенных систем Erlang, и их необходимо учитывать. Самое интересное, что грамотно спроектированную конкурентную систему довольно легко превратить в распределенную и запускать на нескольких компьютерах. Конечно, это преобразование далеко не бесплатно и приводит к возникновению некоторых нестандартных проблем. Но благодаря специальным средствам для создания распределенных систем, многие из которых вам уже известны, беспокоиться о них не придется, и вы сможете сконцентрировать свое внимание на решении основных задач.

В этой главе вы увидите, как легко можно превратить обычную систему в отказоустойчивый кластер. Для начала вам необходимо познакомиться с основными примитивами распределенных систем.

12.1. Примитивы распределенных вычислений

Распределенные системы на основе BEAM представляют собой объединение нескольких узлов в один кластер. Каждый узел – это экземпляр BEAM, с которым связано его имя.

Узлы можно запускать как на одном хосте, так и на нескольких. После установления связи между узлами процессы одного узла могут взаимодействовать с процессами других узлов с помощью стандартного механизма обмена сообщениями.

12.1.1. Запуск кластера

Первым делом необходимо запустить несколько узлов. Для этого укажите параметр `--sname` при запуске оболочки:

```
$ iex --sname node1@localhost  <-- Задание имени узла
iex(node1@localhost)1>         <-- Импорт оболочкой имени узла
```

Если указать параметр `--sname`, текущий экземпляр BEAM превратится в узел с именем `node1@localhost`. Выражение до символа `@` – префикс, уникально идентифицирующий данный узел на данном компьютере. Вторая часть имени (`localhost`) идентифицирует хост-компьютер. Если опустить эту часть, то имя текущего компьютера вставится автоматически.

Параметр `--sname` задает *короткое имя*, в котором указывается только имя хост-компьютера. Возможно также задать длинное имя, где будет присутствовать полное символьное имя или IP-адрес. Это будет рассмотрено более подробно в последнем разделе данной главы.

Запустив узел, вы можете получить его имя с помощью функции *`Kernel.node/0`*:

```
iex(node1@localhost)1> node()
:node1@localhost  <-- Имя узла
```

Как видно из возвращаемого результата, внутри имя представлено атомом.

Обычно узлы создаются для взаимодействия с другими узлами. Попробуем создать еще один узел `node2` в новой сессии оболочки ОС и установим соединение с первым узлом `node1`:

```
$ iex --sname node2@localhost
iex(node2@localhost)1> Node.connect(:node1@localhost)  <-- Установка соединения с другим узлом
true
```

Аргумент, передаваемый функции *`Node.connect/1`*, – это атом, содержащий имя узла, с которым требуется установить соединение. После вызова `Node.connect/1` виртуальная машина пытается установить TCP-соединение с указанным экземпляром BEAM. Как только соединение будет установлено, данные узлы считаются связанными, и взаимодействие между ними осуществляется через это соединение.

Проверить, установлено ли соединение между узлами, можно с помощью функции *`Node.list/0`*, возвращающей список всех узлов, связанных с текущим узлом (текущий узел в списке не указывается). Проверка соединения между `node1` и `node2` дает ожидаемые результаты:

```
iex(node1@localhost)2> Node.list()
[:node2@localhost]  <-- Узлы, связанные с node1

iex(node2@localhost)2> Node.list()
[:node1@localhost]  <-- Узлы, связанные с node2
```

Можно устанавливать соединение между несколькими узлами. На практике BEAM сразу же помогает создать кластер со взаимосвязанными между собой компьютерами. Если добавить третий узел node3 и установить соединение с node2, то автоматически установится соединение и со всеми остальными узлами, с которыми связан узел node2:

```
$ iex --sname node3@localhost
iex(node3@localhost)1> Node.connect(:node2@localhost)

iex(node3@localhost)2> Node.list()
[:node2@localhost, :node1@localhost]  <-- Узел node3 соединен со всеми узлами
```

Это очень удобно в случаях, когда вам необходимо создать кластер, состоящий из нескольких соединенных между собой узлов. При добавлении нового узла в такой кластер этот узел автоматически связывается со всеми остальными узлами кластера.

Чтобы получить список всех узлов кластера, включая текущий, можно использовать функцию Node.list/1:

```
iex(node1@localhost)3> Node.list([:this, :visible])
[:node1@localhost, :node2@localhost, :node3@localhost]
```

Чтобы список включал текущий узел, необходимо указать параметр :this. Параметр :visible позволяет получить список всех *видимых* узлов. Узлы также можно сделать скрытыми, но об этом мы поговорим в последнем разделе данной главы.

Обнаружение отсоединенных узлов

Обнаружение разрыва соединения между узлами заслуживает особого внимания. После установления соединения каждый узел периодически рассылает сообщения всем подключенным к нему узлам, чтобы проверить их активность. Узлы, не ответившие на четыре сообщения подряд, считаются отсоединенными и удаляются из списка подключенных узлов.

Такие узлы не будут подключены снова автоматически, но с помощью функции *Node.monitor/1* (**https://hexdocs.pm/elixir/Node.html#monitor/2**) можно реализовать регистрацию и отправку уведомлений при отключении того или иного узла. Также вы можете отслеживать все подключения и разрывы соединений между узлами с помощью функции *:net_kernel.monitor_nodes* (**http://erlang.org/doc/man/net_kernel.html#monitor_nodes-1**). Мы рассмотрим это на практике чуть позже, когда будем говорить о нарушениях связности сети.

12.1.2. Взаимодействие узлов

Запустив несколько узлов и установив между ними соединение, вы можете наладить между ними взаимодействие. Проще всего сделать это с помощью функции

Node.spawn/2, принимающей имя узла (атом) и анонимную функцию. Эта функция порождает новый процесс на заданном узле и запускает в нем анонимную функцию.

Запустим процесс с узла node1 на узле node2.

Листинг 12.1 ❖ Порождение процесса с одного узла на другом

```
iex(node1@localhost)4> Node.spawn(
                       :node2@localhost,    <-- Заданный узел
                       fn -> IO.puts("Hello from #{node}") end    <-- Запуск на заданном узле
                       )
Hello from node2@localhost
```

Вывод показывает, что анонимная функция была выполнена на другом узле.

Главный процесс группы

В листинге 12.1 происходит кое-что непредвиденное. Несмотря на то что анонимная функция выполнилась на втором узле, результат выводится в оболочке первого узла. Как это возможно? Причина кроется в способе проведения Erlang стандартных операций ввода-вывода.

Все стандартные вызовы ввода-вывода (вроде IO.puts/1) перенаправляются к *главному процессу группы* – процессу, ответственному за выполнение этих операций. Для нового процесса главным является породивший его процесс, даже если он запускается на другом узле. Процесс в предыдущем примере запущен на узле node2, но главный процесс его группы запущен на узле node1. Поэтому данные для вывода – на втором узле (об этом свидетельствует содержимое строки), а результат выводится на экран на узле node1.

Еще один примитив – отправка сообщений процессам вне зависимости от их местонахождения. Он также называется *прозрачностью местоположения*. Операция send работает одинаково, на каком бы узле ни был запущен процесс-получатель.

Рассмотрим простой пример. Запустим вычисления с первого узла на втором и отправим ответ обратно первому узлу:

```
iex(node1@localhost)5> caller = self()

iex(node1@localhost)6> Node.spawn(
                       :node2@localhost,
                       fn -> send(caller, {:response, 1+2}) end    <-- Отправка ответа вызывающему процессу
                       )

iex(node1@localhost)7> flush()
{:response, 3}    <-- Ответ получен
```

Эта схема напоминает классический пример использования процессов. На удаленном узле порождается процесс, и затем от него отправляется сообщение вызывающему процессу. Обратите внимание на то, как используется переменная caller. Механизм замыканий работает, несмотря на то что анонимная функция выполняется на другом узле.

Наконец, используется вспомогательная функция оболочки `IEx – flush`, извлекающая все сообщения из почтового ящика текущего процесса и выводящая их в консоль. Это доказывает, что сообщения были приняты процессом на вызывающем узле.

Ограничений на содержимое сообщений нет. Сообщение одного экземпляра BEAM сработает и в других экземплярах (с одной оговоркой, описанной в примечании ниже). Когда процесс-получатель находится на другом узле, на узле-отправителе сообщение кодируется функцией `:erlang.term_to_binary/1` и расшифровывается на узле-получателе с помощью `:erlang.binary_to_term/1`.

Не порождайте анонимные функции и не отправляйте их на другие узлы

Вы можете порождать анонимные функции из оболочки, что не совсем обычный случай, так как объявленные в оболочке функции внедряют свой код и интерпретируются динамически при каждом вызове. Функции, определенные в модулях, напротив, можно порождать удаленно (или передавать удаленному узлу в сообщении), только в основе обоих узлов лежит один и тот же скомпилированный код. Соблюсти эти требования довольно сложно, если ваша система запущена на группе компьютеров с несколькими узлами и вам нужно обновить код. Внести изменения в код всех узлов одновременно не получится, и на какой-то момент времени их код не будет согласован.

Поэтому рекомендуется избегать передачи анонимных функций удаленным узлам. Используйте функцию `Node.spawn/4`, принимающую MFA (модуль, функцию, список аргументов), описывающий функцию, которую необходимо выполнить на удаленном узле. Пока модуль существует на заданном узле и экспортирует соответствующую функцию, это вполне безопасно.

В системе с несколькими узлами понятие локальной регистрации наконец начинает приобретать смысл. Когда вы регистрируете процесс локально, он становится видимым только для текущего узла. Это означает, что вы можете использовать одно и то же имя для регистрации процессов на разных узлах (но только один раз на одном узле). Например, зарегистрируем процессы оболочки для `node1` и `node2`:

```
iex(node1@localhost)8> Process.register(self(), :shell)
true

iex(node2@localhost)3> Process.register(self(), :shell)
true
```

Вызов `send(:shell, some_message)` отправит сообщение либо первому, либо второму узлу, в зависимости от того, что вы укажете в вызове.

Обратиться к локально зарегистрированному процессу на другом узле можно с помощью выражения `{some_alias, some_node}`. Например, чтобы отправить сообщение от `node1` к `node2`, можно сделать следующее:

```
iex(node1@localhost)9> send(
                         {:shell, :node2@localhost},
                         "Hello from node1!"
                       )
```

Идентификация процесса, зарегистрированного на другом узле

Теперь проверим на втором узле, что сообщение получено:

```
iex(node2@localhost)4> flush()
"Hello from node1!"
```

Вы также можете использовать формат `{some_alias, some_node}` при отправке GenServer-запросов (типа call и cast). Существуют две специальные функции – `GenServer.abcast/3` и `GenServer.multi_call/4`, позволяющие отправлять запросы всем локально зарегистрированным на заданных узлах процессам.

12.1.3. Обнаружение процессов

Обнаружение процессов – очень важная для кластера операция. Не важно, распределенная ваша система или нет, порядок взаимодействия процессов всегда один и тот же:

1) получение клиентским процессом pid сервера;
2) отправка сообщения от клиента к серверу.

Первый шаг – это обнаружение процесса. В главе 9 вы использовали метод обнаружения процессов с помощью модуля `Registry`.

Даже если в вашей системе всего один узел, обнаружение pid процессов все равно требуется. В распределенной системе эта необходимость остается, разве что потребуется использовать другой способ обнаружения, потому что модуль `Registry` работает только в рамках одного локального узла.

Глобальная регистрация

Проще всего реализовать обнаружение процессов по всему кластеру с помощью модуля *`:global`* (**http://erlang.org/doc/man/global.html**), предоставляющего возможность глобальной регистрации. Пусть у вас есть система на основе кластера с несколькими узлами и вы хотите запустить по одному процессу для каждого списка дел. Глобальная регистрация поможет вам это реализовать.

Например, процесс оболочки узла `node1` можно сделать ответственным за управление списком Bob:

```
iex(node1@localhost)10> :global.register_name({:todo_list, "bob"}, self())
:yes
```

Глобальный псевдоним текущего процесса – `{:todo_list, "bob"}`. Результат (`:yes`) означает, что глобальная регистрация прошла успешно.

Теперь все процессы на всех узлах смогут обнаружить процесс, зарегистрированный под этим псевдонимом. Зарегистрировать глобально процесс оболочки узла `node2` не получится:

```
iex(node2@localhost)7> :global.register_name({:todo_list, "bob"}, self())
:no
```

> **Принцип работы глобальной регистрации**
>
> Модуль глобальной регистрации реализован на Erlang, и его можно реализовать самостоятельно на Elixir. Это всего лишь более продуманная версия реестра процессов для систем с несколькими узлами. Во время регистрации глобального

псевдонима устанавливается своеобразная блокировка на уровне всего кластера, предотвращающая одновременные регистрации процессов на других узлах. Затем проверяется, не зарегистрирован ли уже такой псевдоним. Если нет, то все узлы уведомляются о регистрации нового процесса, после чего блок снимается. Как вы понимаете, все это сопровождается передачей многочисленных коротких сообщений между узлами.

Для обнаружения процесса можно использовать функцию *:global.whereis_name/1*:

```
iex(node2@localhost)8> :global.whereis_name({:todo_list, "bob"})
#PID<7954.90.0>
```

Данные операции поиска локальны. Когда выполняется регистрация процесса, все узлы получают уведомление об этом и сохраняют информацию о регистрации в свои локальные таблицы ETS. Каждый последующий поиск по любому узлу выполняется на этом узле без каких-либо дополнительных взаимодействий с другими процессами. Операция поиска выполняется быстро, в отличие от регистрации, при которой уходит некоторое количество времени на рассылку уведомлений всем процессам.

Обратите внимание на формат pid `#PID<7954.90.0>`. Первый номер в его строковом представлении не 0, и это означает, что данный процесс запущен не на текущем узле.

Распознавание процессов, запущенных на удаленных узлах

Очевидно, что pid есть не только у локальных процессов, но и у процессов, запущенных на удаленных компьютерах. Практически во всех случаях физическое расположение процесса значения не имеет. Но вам необходимо быть в курсе некоторых особенностей pid процессов в распределенных системах.

Все идентификаторы, которые вы могли видеть до этого момента, имели формат `<0.X.0>`, где `X` – положительное целое число. У каждого процесса есть уникальный идентификатор на уровне узла, который заключается в двух последних цифрах строкового представления pid. Если создать достаточное количество процессов на одном узле, третье число pid также может быть отличным от нуля.

Первое число – это номер, внутренний идентификатор узла, на котором запущен процесс. Число 0 на этом месте означает, что процесс запущен на локальном узле. Соответственно, если вы получили pid процесса в формате `<X.Y.Z>` и `X` не равен 0, то это pid удаленного процесса. Определить узел, на котором запускается процесс, программно можно с помощью функции `Kernel.node/1` (**https://hexdocs.pm/elixir/Kernel.html#node/1**).

Глобальная регистрация позволяет направить все запросы по одному и тому же ресурсу (в данном случае списку дел) в единую точку синхронизации (процесс). Такой же подход вы использовали ранее с системой, состоящей из одного узла. Вы увидите, как это работает, чуть позже, когда начнете превращать текущую систему в распределенную.

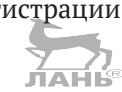
Глобальную регистрацию можно использовать и с GenServer, как показано в следующем примере:

```
GenServer.start_link(
  __MODULE__,
  arg,
  name: {:global, some_global_alias}
)

GenServer.call({:global, some_global_alias}, ...)
```

Регистрация процесса под глобальным псевдонимом

Использование глобального псевдонима для отправки запроса

Наконец, если зарегистрированный процесс откажет или узел, на котором он запущен, отключится, псевдоним автоматически снимается с регистрации на всех остальных компьютерах.



Группы процессов

Существует еще один часто используемый подход для того случая, когда вам необходимо зарегистрировать несколько процессов под одним псевдонимом. Это кажется странным, но может пригодиться, если вы захотите категоризировать процессы кластера и массово рассылать сообщения всем процессам одной категории.

Например, в резервных кластерах необходимо сохранять несколько копий одних и тех же данных. Если один узел откажет, его копия должна храниться на каком-либо другом узле кластера. Для этого вы можете использовать модуль *:pg2* (**http://erlang.org/doc/man/pg2.html**). Он позволяет создавать группы на уровне кластера и добавлять в них большое количество процессов. При добавлении нового узла в группу все остальные узлы группы оповещаются, и вы можете позже обратиться в группе и запросить список всех входящих в нее процессов.

Попробуем сделать так, чтобы оба процесса оболочки node1 и node2 обрабатывали список Bob. Для этого нужно:

1) создать группу процессов для списка Bob;
2) добавить оба процесса в эту группу.

Чтобы создать группу процессов, необходимо лишь вызвать функцию *:pg2.create/1* на любом узле и указать произвольный терм в качестве идентификатора группы. Сделаем это на узле node1:

```
iex(node1@localhost)11> :pg2.start()
iex(node1@localhost)12> :pg2.create({:todo_list, "bob"})
:ok
```

Эта группа сразу же становится видимой для узла node2:

```
iex(node2@localhost)9> :pg2.start()
iex(node2@localhost)10> :pg2.which_groups()
[todo_list: "bob"]
```

Оставшись на узле node2, добавим его процесс оболочки в эту группу:

```
iex(node2@localhost)11> :pg2.join({:todo_list, "bob"}, self())
:ok
```

Это изменение должно быть замечено узлом node1:

```
iex(node1@localhost)13> :pg2.get_members({:todo_list, "bob"})
[#PID<8531.90.0>]
```

И наконец, добавим процесс оболочки узла node1 в ту же группу:

```
iex(node1@localhost)14> :pg2.join({:todo_list, "bob"}, self())
:ok
```

Теперь оба процесса объединены в одну группу, и оба узла знают об этом:

```
iex(node1@localhost)15> :pg2.get_members({:todo_list, "bob"})
[#PID<8531.90.0>, #PID<0.90.0>]

iex(node2@localhost)12> :pg2.get_members({:todo_list, "bob"})
[#PID<0.90.0>, #PID<7954.90.0>]
```

Где это применимо? При обновлении списка Bob вы можете отправить запрос соответствующей группе процессов и получить список всех процессов, отвечающих за список Bob. Затем можно отправить запрос всем процессам этой группы, например с помощью функции GenServer.multi_call/4. Так вы наверняка обеспечите обновление всех копий данных списка в кластере.

Но если вам необходимо, например, получить записи списка дел, это можно делать в одном процессе группы (отправлять запросы ко всем процессам имеет смысл разве что только для перепроверки). Значит, вам понадобится узнать pid только одного процесса группы, что делается с помощью функции *:pg2.get_closest_pid/1*, возвращающей pid локального процесса, а если его не существует – pid случайного процесса группы.

Как и модуль :global, модуль :pg написан полностью на Erlang и является улучшенной версией реестра процессов. Информация о создании групп и добавлении в них процессов распространяется по всему кластеру, но поиск выполняется по локально сохраненной копии таблицы ETS. Аварийное завершение процессов и отключение узлов автоматически обнаруживается, и несуществующие процессы удаляются из группы.

12.1.4. Ссылки и мониторы

Ссылки и мониторы работают, даже если процессы находятся на разных узлах. Процесс получает сигнал выхода или сообщение типа :DOWN (в случае использования монитора), если:

- отказал связанный через ссылку или монитор процесс;
- отказал экземпляр BEAM или целый компьютер, на котором запущены процессы, связанные с текущим через ссылку или монитор;
- потеряно сетевое соединение.

Докажем это. Запустите два узла, соедините их и установите монитор между процессом оболочки node1 и процессом оболочки node2:

```
$ iex --sname node1@localhost
$ iex --sname node2@localhost

iex(node2@localhost)1> Node.connect(:node1@localhost)
iex(node2@localhost)2> :global.register_name({:todo_list, "bob"}, self())

iex(node1@localhost)1> Process.monitor(
  :global.whereis_name({:todo_list, "bob"})
)
```

Установка монитора с процессом, находящимся на другом узле

Теперь остановим узел node2 и сбросим сообщения на node1:

```
iex(node1@localhost)2> flush()
{:DOWN, #Reference<0.0.0.99>, :process, #PID<7954.90.0>, :noconnection}
```

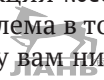
Как видите, вы получили уведомление о том, что связанного посредством монитора процесса больше не существует. Таким образом вы можете обнаружить ошибки в распределенных системах и восстановить систему после их возникновения. На самом деле механизм обнаружения ошибок работает таким же образом, как и в конкурентных системах. Ничего удивительного, ведь конкурентность – это тоже примитив распределенных вычислений.

12.1.5. Прочие сервисы распределения

Есть еще несколько полезных сервисов, являющихся частью стандартной библиотеки Erlang. Далее мы кратко пробежимся по ним, но когда вам предстоит разработать распределенную систему, стоит уделить больше времени на их изучение.

Как отмечалось ранее, большинство базовых примитивов находятся в модуле *Node* (**https://hexdocs.pm/elixir/Node.html**). Помимо этого, имеется несколько сервисов в модулях *:net_kernel* (**http://erlang.org/doc/man/net_kernel.html**) и *:net_adm* (**http://erlang.org/doc/man/net_adm.html**).

Иногда существует необходимость вызова функций на других узлах. Как вы уже видели, это можно осуществить с помощью функции Node.spawn, но этот низкоуровневый способ уместен далеко не всегда. Проблема в том, что Node.spawn работает по принципу «запустили и забыли», а потому вам ничего не известно о результате данной операции.



Чаще всего необходимо получить результат выполнения удаленной функции или вызвать какую-либо функцию на нескольких узлах и собрать все результаты. В таких случаях удобно пользоваться возможностями Erlang-модуля *:rpc* (**http://erlang.org/doc/man/rpc.html**).

Например, чтобы вызвать функцию на другом узле и получить ее результат, можно использовать функцию *:rpc.call/4*, принимающую в качестве аргументов узел и MFA функции, которую необходимо вызвать удаленно. Вот так выглядит вызов функции Kernel.abs(-1) на узле node2:

```
iex(node1@localhost)1> :rpc.call(:node2@localhost, Kernel, :abs, [-1])
1
```

Модуль :rpc содержит и другие вспомогательные функции, позволяющие вызвать функцию удаленно на нескольких узлах кластера. Они будут рассмотрены на примере чуть позже, когда мы добавим к базе данных возможность репликации.

Обмен сообщениями – основной примитив распределенных вычислений

Многие сервисы вроде :rpc полностью реализованы на Erlang. Модуль :rpc, так же как и модули :global и :pg2, берет за основу прозрачный обмен сообщениями и возможность посылать сообщения локально зарегистрированным процессам на удаленных узлах. Например, :rpc использует локально зарегистрированный процесс :rex (который запускается при запуске Erlang-приложения :kernel). При от-

правке rpc-запроса на другие узлы происходит передача сообщения, содержащего MFA, :rex-процессам заданных узлов, вызову функции `apply/3` на этих узлах и отправке ответа.
Если вы хотите углубиться в изучение программирования распределенных систем на Erlang, обратите особое внимание на код модулей `rpc.erl`, `pg2.erl` и `global.erl`, потому что они содержат большое количество идиом и шаблонов проектирования.

Поговорим немного о блокировках на уровне кластера, реализованных в модуле `:global`. Вы можете установить блокировку с произвольным именем. Как только вы ее установите, ни один процесс кластера не сможет ее получить, пока вы не снимете ее.

Посмотрим на это в действии. Запустите узлы `node1` и `node2`. Установите блокировку на `node1` с помощью функции *`:global.set_lock/1`*:

```
iex(node1@localhost)1> :global.set_lock({:some_resource, self()})
true
```

Кортеж, который вы указываете, состоит из ID ресурса и ID инициатора запроса. ID ресурса – это произвольный терм, а ID инициатора запроса – уникальный идентификатор. Два различных инициатора кластера не могут получить одну и ту же блокировку. Как правило, на месте ID инициатора стоит ID процесса, и это значит, что в определенный момент времени как минимум один процесс может установить конкретную блокировку.

Установка блокировки предполагает взаимодействие с другими узлами кластера. Как только функция `:set_lock` возвратит результат, можно быть уверенным, что этот процесс получил блокировку, и никакой другой процесс в кластере не сможет получить эту же блокировку. Попробуйте установить блокировку на `node2`:

```
iex(node2@localhost)1> :global.set_lock({:some_resource, self()})
```

Ожидает до тех пор, пока блокировка не будет снята

Процесс оболочки будет находиться в ожидании неопределенное количество времени (это время можно ограничить с помощью дополнительного параметра). Как только вы снимете блокировку с узла `node1`, она установится на узле `node2`:

```
iex(node1@localhost)2> :global.del_lock({:some_resource, self()})
iex(node2@localhost)2>
```

Блокировка установлена на процесс оболочки node2

Для реализации установки и снятия блокировки существует также вспомогательная функция *`:global.trans/2`* (**http://erlang.org/doc/man/global.html#trans-2**), которая принимает блокировку, выполняет указанную анонимную функцию и снимает блокировку.

Стоит отметить, что использования блокировок все же следует избегать, так как они вызывают все те же проблемы, что и классические подходы к обеспечению синхронизации: возможность взаимоблокировок, активных взаимоблокировок и зависаний. Синхронизацию лучше всего выполнять с помощью процессов: так проще планировать работу системы.

Однако в умеренном количестве блокировки могут в отдельных случаях повысить производительность системы. Как вы помните, при передаче сообщений

расходуется определенное количество памяти, и это особенно актуально в распределенных системах, в которых сообщения должны быть упорядочены и разосланы по всей сети. Если сообщение очень большое, то его передача приведет к значительной задержке, что, в свою очередь, повлияет и на общую производительность системы.

Именно в таких случаях могут выручить блокировки: они позволят синхронизировать несколько процессов на разных узлах без отправки больших сообщений. Идея довольно проста. Скажем, вам необходимо убедиться в том, что операции по обработке большого объема данных во всем кластере происходят по порядку (в любой момент времени может работать только один процесс во всем кластере). Обычно это осуществляется путем передачи данных процессу, являющемуся точкой синхронизации системы, но передача большого объема данных может привести к снижению производительности, ведь эти данные необходимо скопировать и разослать по всей сети. Чтобы этого избежать, можно произвести синхронизацию процессов с помощью блокировки, а затем обработать данные в контексте вызывающей стороны:

```
def process(large_data) do
  :global.trans(                                  ← Блокировка на уровне кластера
    {:some_resource, self},
    fn ->
      do_something_with(large_data)               ← Выполнение в вызывающем процессе
    end
  )
end
```

Вызов функции `:global.trans/2` обеспечивает изоляцию на уровне кластера. Только один процесс во всем кластере может выполнять операцию `do_something_with/1` на `:some_resource` в единицу времени. Поскольку операция `do_something_with/1` выполняется в вызывающем процессе, объемное сообщение другому процессу не посылается. Вызов `:global_trans/2` приводит к дополнительному взаимодействию между процессами, но сообщения, используемые для установки блокировки, намного короче, чем `large_data`, благодаря чему пропускная способность возрастает.

На этом мы закончим изучение примитивов распределенных вычислений. Некоторые немаловажные моменты, имеющие место при использовании сети в качестве канала передачи данных, остались без внимания. Мы вернемся к этой теме в последнем разделе данной главы, а пока сосредоточимся на превращении имеющейся системы в распределенную.

12.2. Создание отказоустойчивого кластера

Имея в своем арсенале парочку примитивов распределенных вычислений, вы можете начать работать над кластером веб-серверов. Текущая цель – сделать систему более гибкой по отношению к различным сбоям, включая отказы целых узлов. Приведенное далее решение будет элементарным. Создание полноценной распределенной системы требует гораздо большего внимания к деталям, и информации по этой теме наберется на отдельную книгу.

Положительный момент заключается в том, что создать простейшую распределенную систему совсем не сложно. В данном разделе вы увидите, как примитивы распределенных вычислений отлично вписываются в существующую конкурентную модель BEAM.

Большинство преобразований будут основаны на абстракции `GenServer`. Ничего удивительного, ведь обмен сообщениями – основной инструмент распределенных вычислений BEAM. Перед тем как приступить к следующему разделу, убедитесь, что вы помните принцип работы `GenServer`, и при необходимости освежите свои знания, обратившись к главе 6.

12.2.1. Устройство кластера

Цели будущего кластера обманчиво просты:

- кластер будет состоять из нескольких узлов, в основе которых будет лежать один и тот же код и которые будут предоставлять одни и те же функции (веб-интерфейс для управления несколькими списками дел);
- об изменениях данных должны оповещаться все узлы кластера: операция по одному списку дел на одном узле должна быть видима другому узлу, а для клиентов не должно иметь значения, к какому узлу обращаться;
- отказ одного узла не должен оказывать влияния на кластер, система должна продолжать обслуживать клиентов, а данные отказавшего узла не должны быть потеряны.

Эти цели характеризуют отказоустойчивую систему, которая всегда готова обслуживать клиентов, а отказы каких-либо ее частей не вызывают проблем. Благодаря этому система становится более гибкой и высокодоступной.

Нарушение связности сети

Обращаю ваше внимание, что в рамках данной книги не будет показано решение самой сложной задачи распределенных систем – нарушения связности сети. Это такие ситуации, при которых в канале связи между двумя узлами происходит сбой, и соединение между ними пропадает. В этом случае вы можете столкнуться с проблемой раскола кластера (split brain), когда кластер разделяется на два или более мелких кластеров, каждый из которых продолжает по-прежнему обслуживать клиентов. В таком случае вы получите несколько изолированных систем, принимающих входные данные от пользователей. В конце концов, данные этих узлов начнут конфликтовать между собой. В этом разделе данная проблема игнорируется, но некоторые последствия нарушения связности сети будут обговорены.

Начнем работать над преобразованием текущей системы в распределенную. Первым в игру вступает кеш-процесс.

12.2.2. Распределенный кеш

В каком-то смысле кеш является центральным элементом системы, обеспечивающим целостность данных. Давайте вспомним, как он работает (см. рис. 12.2).

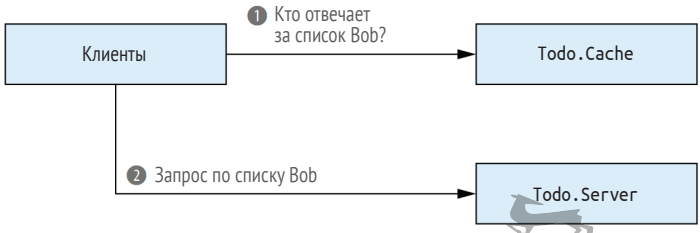


Рис. 12.2 ❖ Использование кеша

Для выполнения каких-либо операций со списком дел кеш предоставляет соответствующий серверный процесс, который затем превращается в место синхронизации этого списка дел. Все запросы по списку Bob проходят через этот процесс, что обеспечивает целостность данных и предотвращает возникновение состояния гонки.

Превращая систему в кластер, вы должны сохранить эту особенность. Разница лишь в том, что кеш необходимо каким-то образом приспособить к работе на всех узлах кластера. Где бы в кластере ни возник вопрос о том, какой процесс отвечает за указанный список, ответ всегда должен приводить к одному и тому же процессу (если он не отказал).

Это единственное, что вам требуется реализовать, чтобы приспособить кеш к работе в распределенной системе. Как вы увидите далее, эта работа не займет у вас много времени.

Обнаружение серверов

Существуют различные способы обнаружения процессов внутри кластера. Пожалуй, самый простой из них (хотя и не самый эффективный) – использование сервисов модуля `:global`, позволяющих зарегистрировать процесс под глобальным псевдонимом (произвольным термом, идентифицирующим данный процесс в кластере). Для этого необходимо:

- добавить в модуль `Todo.Server` поддержку глобальной регистрации;
- научить `Todo.Cache` работать с новым видом регистрации.

Попробуем это реализовать. До этого вами использовался модуль `Registry`, подходящий только для регистрации процессов на одном узле. Для регистрации и обнаружения процессов в распределенной системе вы можете полагаться на возможности модуля `:global`.

Регистрация процессов

Должно быть, вам непросто разобраться во всех разновидностях регистраций, поэтому давайте вспомним их основные отличия:

- основной разновидностью регистрации является локальная регистрация, при которой в качестве псевдонима процесса на узле используется простой атом;
- модуль `Registry` позволяет использовать более сложные псевдонимы (любой терм Elixir);
- модуль `:global` предоставляет возможность регистрировать псевдонимы на уровне кластера;

- модуль :pg2 используется для регистрации нескольких процессов под псевдонимом внутри кластера (группы процессов), что чаще всего необходимо в распределенных системах для сценариев «издатель–подписчик».

Заменить модуль Registry на :global в модуле Todo.Server можно одним действием. Текущая версия соответствующего кода, разработанного в главе 9, выглядит следующим образом:

```
defmodule Todo.Server do
  def start_link(name) do
    GenServer.start_link(Todo.Server, name, name: via_tuple(name))
  end

  defp via_tuple(name) do
    Todo.ProcessRegistry.via_tuple({__MODULE__, name})
  end

  ...
end
```

Чтобы при регистрации процессов использовался модуль :global, функции via_tuple/1 необходимо возвращать {:global, registered_name}. Работая над этим, вы также можете переименовать эту функцию, как показано в листинге 12.2.

Листинг 12.2 ❖ Глобальная регистрация серверных процессов (todo_distributed/lib/todo/server.ex)

```
defmodule Todo.Server do
  ...

  def start_link(name) do
    GenServer.start_link(Todo.Server, name, name: global_name(name))
  end

  defp global_name(name) do
    {:global, {__MODULE__, name}}   <-- Глобальная регистрация
  end

  ...
end
```

Всего одна правка, и в вашей системе теперь поддерживается распределенная регистрация и обнаружение процессов. Больше никаких изменений не требуется, система будет работать должным образом.

Однако в данной реализации есть возможная проблема, связанная с производительностью. Когда процесс регистрируется под глобальным псевдонимом, модуль :global синхронно рассылает сообщения всем процессам кластера. Это означает, что глобальная регистрация требует намного больше ресурсов, чем локальная, и это крайне нежелательно, особенно с текущей реализацией кеша, представленной в главе 9. Освежим в памяти необходимые нам части кода модуля кеша:

```
defmodule Todo.Cache do
  ...

  def server_process(todo_list_name) do
```

```
    case start_child(todo_list_name) do ◄──── Запуск нового процесса
      {:ok, pid} -> pid
      {:error, {:already_started, pid}} -> pid
    end
  end

  defp start_child(todo_list_name) do
    DynamicSupervisor.start_child(
      __MODULE__,
      {Todo.Server, todo_list_name}
    )
  end

  ...
end
```


Как вы помните, в разделе 9.2.3 был выбран следующий упрощенный подход: каждый раз при поиске потомка создается новый процесс и происходит его регистрация. Если она не удается, функция `DynamicSupervisor.start_child/2` возвращает `{:error, {:already_started, pid}}`. Теперь, имея распределенную систему, вы больше не можете положиться на такое простое решение, потому что беспричинная регистрация может создать в системе узкое место. Каждый раз, когда необходимо выполнить операции со списком, даже если серверный процесс уже запущен, производите глобальную регистрацию, которая установит блокировку и оповестит о регистрации нового процесса все узлы системы.

Для начала реализуем явный поиск. Будем проверять, не зарегистрирован ли конкретный серверный процесс, и запускать его только в случае неудачи поиска. Для этого необходимо сначала добавить в модуль `Todo.Server` функцию `whereis/1`, принимающую имя и возвращающую pid зарегистрированного процесса или nil, в случае если процесс под заданным именем не найден. Соответствующий код приведен в следующем листинге.

Листинг 12.3 ❖ Обнаружение серверных процессов (todo_distributed/lib/todo/server.ex)

```
defmodule Todo.Server do
  ...

  def whereis(name) do
    case :global.whereis_name({__MODULE__, name}) do
      :undefined -> nil
      pid -> pid
    end
  end

  ...
end
```

Стоит еще раз заметить, что функция `:global.whereis_name/1` не приводит к взаимодействию каких-либо узлов. Она лишь осуществляет поиск по локальной таблице ETS. Поэтому от функции `Todo.Server.whereis/1` можно ожидать достаточно высокой и стабильной производительности.

Следующим шагом перепишите код модуля `Todo.Cache`, как показано в листинге ниже.

Литсинг 12.4 ❖ Оптимизированное обнаружение процессов (todo_distributed/lib/todo/cache.ex)

```
defmodule Todo.Cache do
  ...

  def server_process(todo_list_name) do
    existing_process(todo_list_name) || new_process(todo_list_name)
  end

  defp existing_process(todo_list_name) do
    Todo.Server.whereis(todo_list_name)
  end

  defp new_process(todo_list_name) do
    case DynamicSupervisor.start_child(
      __MODULE__,      {Todo.Server, todo_list_name}    ) do
      {:ok, pid} -> pid
      {:error, {:already_started, pid}} -> pid
    end
  end
end
```

С помощью обертки и оператора || функция `server_process/1` делает подход к обнаружению серверных процессов понятным: она либо возвращает pid существующего процесса, либо пытается запустить новый. Как уже говорилось в разделе 9.2.3, функция `new_process/1` прекрасно справляется с ситуациями, когда два разных клиентских процесса одновременно пытаются запустить сервер для одного и того же списка дел. Она будет работать точно так же и в распределенной системе, а также разрешит состояние гонки между двумя клиентами на двух разных узлах.

Проделав эти изменения, вы избавились от необходимости в модуле `Todo.ProcessRegistry`. Вы можете удалить его из проекта и из спецификации потомков супервизора `Todo.System`.

Альтернативный метод обнаружения

Не стоит забывать, что глобальная регистрация приводит ко множественному взаимодействию процессов и выполняется последовательно (только один процесс может производить глобальную регистрацию в единицу времени). Это означает, что предыдущий подход не отличается гибкостью по отношению к числу возможных списков дел или узлов кластера. Кроме того, в медленных сетях его производительность будет низкой.

Существуют и другие методы обнаружения процессов. Главной задачей является поиск процесса, ответственного за какой-либо список дел, и уменьшение количества сетевых взаимодействий. Это можно сделать путем введения правила, связывающего конкретный список дел с определенным узлом сети. Вот как это выглядит:

```
def node_for_list(todo_list_name) do
  all_sorted_nodes = Enum.sort(Node.list([:this, :visible]))

  node_index = :erlang.phash2(
    todo_list_name,
    length(all_sorted_nodes)
  )
```

```
  Enum.at(all_sorted_nodes, node_index)
end
```

В данном коде происходит получение списка всех узлов и его сортировка в определенном порядке. Затем исходное имя хешируется так, чтобы результат находился в диапазоне [`0..length(all_sorted_nodes)`]. И наконец, узел возвращается на исходную позицию в списке. Такой механизм при стабильности кластера (неизменности списка узлов) обеспечивает соответствие конкретного списка дел определенному узлу.

Теперь обнаружение процесса можно выполнить за одно короткое действие. С использованием предыдущей версии `Todo.Cache` (не той, что вы только что реализовали) код будет выглядеть так:

```
:rpc.call(
  node_for_list(todo_list_name),
  Todo.Cache,
  :server_process,
  [todo_list_name]
)
```

Вы просто переходите на указанный узел и получаете искомый процесс. Глобальная регистрация в данном случае не нужна, и реализация модуля `Todo.Cache` может остаться прежней. Результатом данного кода является pid процесса, который затем можно использовать для отправки запроса. Преимуществом данного метода является уменьшенное количество сетевых взаимодействий.

Главный его недостаток – неприспособленность к изменениям конфигураций кластера. Если вы добавите новые узлы или какие-то узлы отсоединятся, необходимо будет изменить правила соответствия. С этой проблемой справиться крайне сложно: нужно обнаружить изменение в кластере (что, в общем-то, возможно и будет показано далее) и перенести все данные на другие узлы, согласно новому правилу соответствия. Пока эти действия выполняются, система должна продолжать обслуживать клиентов, что еще больше усложняет задачу. Количество необходимых к переносу данных можно значительно уменьшить, используя определенную форму согласованного хеширования (**http://en.wikipedia.org/wiki/Consistent_hashing**) – более продуманный механизм связывания ключей и узлов, более устойчивый к изменениям в кластере.

Очевидно, что данную реализацию очень легко перегрузить, и именно поэтому мы начали с простого решения и выбрали глобальную регистрацию. Да, это решение не поддается масштабированию, но оно замечательно работает при своей элементарности. Если же вам хочется достичь большей производительности и масштабируемости, придется выбрать более сложный подход. Вместо того чтобы полагаться на удачу, попробуйте рассмотреть в качестве вариантов готовые сторонние библиотеки вроде Syn (**https://github.com/ostinelli/syn**) или Swarm (**https://github.com/bitwalker/swarm**).

12.2.3. Создание репликационной базы данных

Выполнив все описанные выше преобразования, вы получите следующее поведение:

1) при получении первого запроса по списку Bob на узле, отвечающем за обработку этого запроса, создается процесс списка дел;
2) все последующие запросы по списку Bob перенаправляются процессу, созданному на предыдущем шаге;
3) если узел (или процесс), созданный на шаге 1, отказывает, новый запрос по списку Bob приведет к регистрации нового серверного процесса.

На первый взгляд, у вас имеется полноценная распределенная система. Однако тестировать ее пока нет смысла, так как одна немаловажная проблема осталась нерешенной: база данных не устойчива к отказам. Пусть в списке Bob, находящемся на узле A, произведено несколько изменений. Если этот узел откажет, управление списком Bob будет передано другому узлу, например узлу Б, а данные, хранившиеся на отказавшем узле, не перенесутся на новый узел, и изменения в списке не сохранятся.

Как вы понимаете, необходимо хранить копию базы данных, чтобы ее можно было восстановить после отказа. Самый простой (и не самый эффективный) метод – сохранение этих данных во всех частях кластера. Он показан на рис. 12.3.

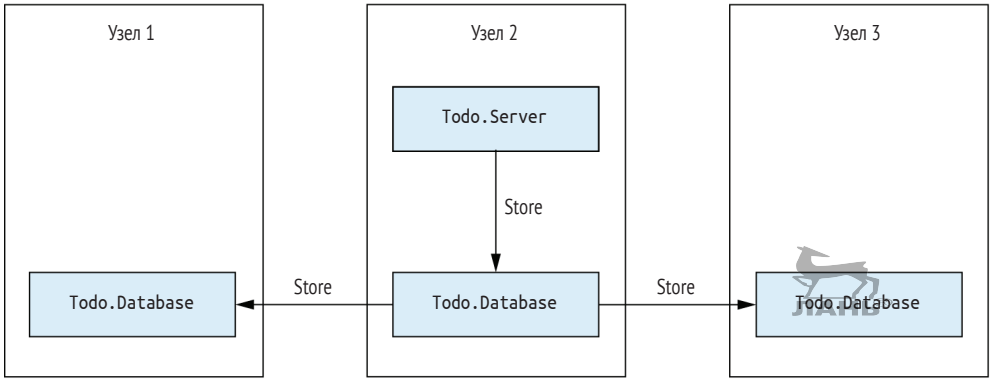


Рис. 12.3 ❖ Репликация базы данных

Идея довольно проста. Когда данные сохраняются в базу данных, изменения распространяются на все узлы кластера. Реализацию этого можно упростить с помощью сервисов модуля `:rpc`. Как уже отмечалось ранее, модуль `:rpc` позволяет вызывать функции на всех узлах кластера. Попробуем использовать эту возможность в текущей системе. Для этого немного изменим реализацию модуля `Database`.

1. Переименуйте функцию `Database.store` в `Database.store_local`. Ее код оставьте прежним.
2. Добавьте новую реализацию функции `Database.store` и сделайте так, чтобы она вызывала функцию `Database.store_local` на всех узлах кластера.

Необходимо также будет превратить вызов функции `Todo.DatabaseWorker.store/2` в синхронный запрос. Это должно было быть сделано с самого начала. В главе 7 в дидактических целях было принято решение использовать асинхронный запрос. На практике при отправке другому процессу запроса на сохранение данных следует запрашивать у него и подтверждающее сообщение, чтобы знать наверня-

ка, выполнена операция или нет. Это особенно важно при переходе к менее надежным средствам коммуникации (сетям), в которых может отказать что угодно. Если запрос создается для использования между узлами, он должен быть асинхронным.

Это все, что требуется изменить. Реализация функции `Todo.Database.get/1` остается прежней. Если вы хотите считать данные, можно выполнить эту операцию с локального узла, предполагая, что все остальные узлы имеют копию этих данных. Все требуемые изменения показаны в листинге ниже.

Листинг 12.5 ❖ Сохранение данных на всех узлах (todo_distributed/lib/todo/database.ex)

```
defmodule Todo.Database do
  ...

  def store(key, data) do
    {_results, bad_nodes} =
      :rpc.multicall(
        __MODULE__,
        :store_local,
        [key, data],
        :timer.seconds(5)
      )

    Enum.each(bad_nodes, &IO.puts("Store failed on node #{&1}"))
      :ok
  end

  ...
end
```

Вызов функции store_local на всех узлах

Запись в журнал результатов неудачных операций

В данном случае для вызова функции `store_local` на всех узлах кластера используется функция `:rpc.multicall/4`, принимающая MFA и значение тайм-аута. В результате `store_local` вызывается на всех узлах кластера, а все возвращаемые ею значения собираются в один кортеж вида `{results, bad_nodes}`, содержащий также список узлов, не ответивших за заданный промежуток времени.

Всегда задавайте тайм-ауты

Указанные в функции `multicall` тайм-ауты имеют большое значение. Без них сама функция и, как следствие, операция `store` зависнут на неопределенное время. Выполняя распределенные синхронные вызовы, следует всегда указывать для них тайм-аут. Этот случай ничем не отличается от синхронных вызовов между двумя процессами – для них тоже есть тайм-аут в 5 секунд, предоставляемый GenServer по умолчанию. Еще раз напоминаю, что операции между узлами практически не отличаются от операций между процессами, и во многих случаях вам предстоит решить примерно одинаковый набор задач.

Наконец, функция выводит список всех узлов, которые не смогли ответить на запрос в течение заданного промежутка времени. На самом деле этого недостаточно, и необходимо также будет убедиться, что каждый полученный запрос

возвратил :ok. Кроме того, следует выполнять какие-либо действия в случае частичного успеха, иначе целостность данных в кластере будет нарушена: каждый узел будет содержать совершенно разные данные. Чтобы правильно решить эту проблему, необходимо реализовать протокол двухфазного подтверждения и базу данных с возможностью журналирования. Благодаря этому вы сможете откатить изменения в случае частичной неудачи запроса. Для краткости и простоты в данной книге реализация вышеописанного не приводится, но стоит иметь в виду, что при разработке реальных проектов эта проблема требует особого внимания.

Еще одно изменение, не приведенное здесь, касается рабочих процессов базы данных. До этого момента данные сохранялись в папку persist, а теперь следует учитывать имя узла. Данные узла node1@localhost будут храниться в папке persist/node1. Это делается в основном для удобства тестирования, а также ради возможности запуска нескольких узлов из одного корневого каталога.

Благодаря этим простым изменениям данные могут быть скопированы на любой узел кластера. На этом простейшая кластерная система готова, и ее пора протестировать.

12.2.4. Тестирование системы

Итак, чтобы проверить систему в действии, необходимо запустить несколько узлов, установить между ними соединение и посмотреть, как работает кластер. В главе 11 мы назначили веб-серверу для прослушивания порт 5454, но два узла не могут прослушивать один и тот же порт. К счастью, в разделе 11.4 мы сделали возможной конфигурацию порта с помощью окружения приложения, а значит, порт можно изменить прямо из командной строки.

Запустите два экземпляра узла – node1 и node2, слушающих порты 5454 и 5555 соответственно:

```
$ iex --sname node1@localhost -S mix
$ iex --erl "-todo port 5555" --sname node2@localhost -S mix
```

Запуск узла node1, слушающего стандартный порт

Запуск узла node2 и задание альтернативного порта

Установим соединение между этими двумя узлами:

```
iex(node1@localhost)1> Node.connect(:node2@localhost)
```

Кластер готов, на нем уже можно использовать сервисы. Добавьте запись в список Bob на первом узле:

```
$ curl -d "" \
  "http://localhost:5454/add_entry?list=bob&date=2018-12-19&title=Dentist"
OK
```

Убедимся, что эта запись доступна на втором узле:

```
$ curl "http://localhost:5555/entries?list=bob&date=2018-12-19"
2018-12-19    Dentist
```

Этот тест показывает, что данные распространяются по всему кластеру. Кроме того, если посмотреть на оболочки обоих узлов, вы увидите сообщение о запуске

серверного процесса для списка Bob на node1, но не на node2. Это доказывает, что даже при попытке обращения к списку Bob на другом узле контекст перенаправляется к соответствующему процессу узла node1.

Операции со списком Bob можно выполнять на узле node2, не беспокоясь о безопасности данных:

```
$ curl -d "" \  "http://localhost:5555/add_entry?list=bob&date=2018-12-19&title=Movies"

$ curl "http://localhost:5454/entries?list=bob&date=2018-12-19"
2018-12-19    Dentist
2018-12-19    Movies
```

Теперь убедимся, что отказ одного из узлов не оказывает влияния на оставшуюся часть системы. Остановите узел node1, на котором запущен серверный процесс списка Bob, и попробуйте отправить запрос узлу node2:

```
$ curl "http://localhost:5555/entries?list=bob&date=2018-12-19"
2018-12-19    Dentist
2018-12-19    Movies
```

Очевидно, что кластер продолжает работать, а данные отказавшего узла сохранены. На узле node2 был создан новый серверный процесс списка Bob, получивший свое состояние из копии базы данных.

Теперь можно утверждать, что ваш простейший кластер готов к работе. Осталось несколько задач, которые в данной книге решены не будут:

- необходимо добавить распределитель нагрузки для управления количеством входящих запросов от клиентов;
- необходимо продумать механизм добавления новых узлов в работающий кластер: при добавлении нового узла база данных должна быть синхронизирована с одним из существующих узлов кластера, после чего новый узел сможет выполнять запросы;
- текущий механизм репликации базы данных слабоват, и необходимо реализовать что-то вроде протокола двухфазного подтверждения;
- необходимо добавить восстановление после потери связности сети.

С некоторыми из этих задач справиться не так просто, но они присущи любой распределенной системе, независимо от того, какие технологии вы выберете для ее разработки. Важно понимать, что Erlang не может по мановению волшебной палочки справиться со всеми проблемами, возникающими в распределенных системах. Вам предстоит самостоятельно принимать решения о том, как лучше всего справиться с задачей в той или иной ситуации путем комбинирования базовых примитивов распределенных вычислений, так как универсального способа просто не существует.

Чаще всего лучшим решением будет использование проверенных сторонних компонентов. Например, встроенная база данных Mnesia позволит обеспечить гарантированное выполнение операции записи и упростить процесс добавления новых узлов в кластер. Однако необходимо всегда как следует разобраться в том, как та или иная сторонняя библиотека работает при ее использовании в распределенной системе. Та же Mnesia не умеет справляться с проблемами потери связности сети или раскола кластера, предоставляя это право разработчику. Каждый сторонний компонент имеет свои достоинства и недостатки.

Примитивы распределенных вычислений Erlang очень эффективны. Всего несколько изменений позволили вам превратить обычную систему в распределенную без предварительной подготовки.

12.2.5. Обнаружение потери связности сети

Потеря связности сети – одна из наиболее коварных проблем при создании распределенной системы. Чтобы раскрыть этот вопрос в полной мере, понадобится целая отдельная книга, так что мы пробежимся по нему совсем кратко и рассмотрим только самые простые механизмы обнаружения потери связности.

Если вы приняли решение сделать свою систему распределенной, то вы так или иначе столкнетесь с данной проблемой, и игнорировать ее не стоит. Даже если для создания кластера и репликации вы используете какие-либо сторонние компоненты (например, внешнюю базу данных), необходимо обязательно проанализировать поведение этого компонента при возникновении потери связности сети. Вы должны заранее предвидеть трудности, с которыми предстоит столкнуться, чтобы принять обдуманное и обоснованное решение.

Потеря связности сети – это ситуация, при которой два узла больше не могут взаимодействовать друг с другом. На то есть несколько причин, распознать которые невозможно:

- потеря сетевого соединения;
- слишком медленное сетевое соединение;
- отказ удаленного узла;
- перегрузка удаленного узла и его неспособность отвечать за заданный период времени.

С точки зрения узла все эти ситуации выглядят одинаково: удаленный узел перестает отвечать по непонятной причине. Поэтому практически невозможно гарантировать, что потери связи не произойдет. Даже в сверхбыстрой и надежной сети всего один баг или большая нагрузка могут привести к перегрузке узла, и в результате он не будет способен оперативно отвечать, а другой узел расценит это как потерю связи. Именно поэтому необходимо заранее продумать стратегию разрешения данных ситуаций.

После потери связности сети система разделяется на несколько независимых кластеров, не соединенных друг с другом. Несмотря на то что эти кластеры не могут взаимодействовать, клиенты могут обратиться к любому узлу системы. Эта ситуация также называется *расколом кластера* (*split-brain*). Если кластеры продолжат обслуживать клиентов, система будет вести себя совсем не так, как вы задумывали. Запрос, выполняемый на одном кластере, не будет видим для другого кластера, что может привести к невозможности сохранения изменений или появлению ложных записей в базе данных. Восстановив соединение между этими узлами, вы получите противоречивые данные.

Чтобы решить данную проблему, необходимо сначала добавить обнаружение потерь соединений между узлами. Как отмечалось ранее, узлы периодически рассылают соединенным с ними узлам сообщения, и если какой-то из узлов на него не ответит, он считается отсоединенным. Каждый процесс может подписаться на уведомления об изменениях в списке подключенных узлов с помощью функции `:net_kernel.monitor_nodes/1` (**http://erlang.org/doc/man/net_kernel.html#monitor_nodes-1**).

Аргументом этой функции является логическое выражение, обозначающее добавление новой подписки (`true`) или установку единственного подписчика поверх всех остальных (`false`). В любом случае процесс, вызывающий `monitor_nodes`, будет получать уведомления о подключении и отключении удаленного узла.

Рассмотрим пример. Запустите узел `node1` и оформите подписку на уведомления:

```
$ iex --sname node1@localhost
iex(node1@localhost)1> :net_kernel.monitor_nodes(true)
```

Вызывающий процесс (процесс оболочки) теперь будет получать уведомления.

Запустите еще два узла и установите соединение с узлом `node1`:

```
$ iex --sname node2@localhost
iex(node2@localhost)1> Node.connect(:node1@localhost)
$ iex --sname node3@localhost
iex(node3@localhost)1> Node.connect(:node1@localhost)
```

В оболочке узла `node1` вы увидите соответствующие сообщения:

```
iex(node1@localhost)2> flush()
{:nodeup, :node2@localhost}
{:nodeup, :node3@localhost}
```

То же самое происходит и при отсоединении узлов `node2` и `node3`:

```
iex(node1@localhost)3> flush()
{:nodedown, :node3@localhost}
{:nodedown, :node2@localhost}
```

Как вариант вы можете использовать функцию `Node.monitor/2` (**https://hexdocs.pm/elixir/Node.html#monitor/2**) для отслеживания подключений и отключений конкретного узла.

Так или иначе обнаружение потери связи сводится к отслеживанию сообщений `:nodedown`. Например, чтобы убедиться, что отключившийся узел все еще связан с большинством других узлов системы, можно обработать сообщение `: nodedown`, запросив количество узлов, подключенных к данному. При достаточном количестве система должна продолжить работу, в противном случае необходимо прекратить обслуживание клиентов, например остановив приложение.

Наконец, как уже говорилось выше, вы можете установить монитор или связь с удаленным процессом. Они работают так же, как и между локальными процессами: если удаленный процесс завершится (или узел отсоединится), при использовании монитора вы получите сообщение, а при использовании связи – сигнал выхода.

12.2.6. Высокодоступные системы

Еще в самой первой главе были описаны некоторые свойства высокодоступных систем. Вы могли и не заметить, как достигли их в текущей системе:

- отзывчивость. Так как реализованная система имеет высокую степень конкурентности, аппаратные средства используются более эффективно, что позволяет ей выполнять большое количество запросов одновременно.

Благодаря процессам BEAM система работает без непредвиденных пауз, обычно возникающих из-за того же сбора мусора (процессы делают это самостоятельно и конкурентно). Случайные задачи с длительным временем выполнения не блокируют всю систему, потому что процессы часто вытесняют друг друга. Система работает предсказуемо с примерно одинаковым временем ожидания, которое постепенно увеличивается по мере роста входящей нагрузки;

- масштабируемость. Ваша система является конкурентной и распределенной, а значит, вы можете справиться с высокой нагрузкой с помощью более мощного компьютера или путем добавления дополнительных узлов. Система автоматически начнет использовать доступные ей ресурсы максимально эффективно;
- отказоустойчивость. Благодаря изолированности процессов влияние отдельных ошибок ограничено. С помощью ссылок между процессами вы можете распространять эти ошибки по системе и исправлять их. Супервизоры помогают системе устранить последствия ошибок и восстановиться. В то же время основной код выполняется в штатном режиме, и в нем отсутствуют конструкции для обнаружения ошибок. И наконец, распределенная система продолжает работать даже при отказе входящих в ее состав компьютеров.

Теперь вам должно быть совершенно очевидно, что основным инструментом достижения высокой доступности является конкурентная модель BEAM. Решение основывать работу системы на процессах предоставило нам множество полезных возможностей на пути к реализации полноценной высокодоступной системы.

Конечно, текущая система сильно упрощена. Реализации некоторых ее составляющих, таких как база данных, недоработаны, и она не умеет справляться с потерями связности сети. Несмотря на это, начав разрабатывать реальную высокодоступную систему, непрерывно обслуживающую пользователей, вы должны стремиться к достижению вышеописанных целей, а процессы должны стать вашим главным инструментом.

Итак, распределенная система готова. Перед тем как использовать ее, необходимо обсудить некоторые особенности сетевого соединения.

12.3. Особенности сетевого соединения

До этого момента вы запускали узлы локально, чего вполне достаточно для практики и тестирования на этапе разработки. Но, как правило, на этапе промышленной эксплуатации различные узлы запускаются на разных компьютерах. Для работы с такими кластерами требуется изучить дополнительную информацию, и начнем мы с имен узлов.

12.3.1. Имена узлов

Имена, которые вы использовали, были *короткими* и состояли из произвольного префикса узла (`node1` и `node2`) и имени хоста (`localhost`). Вы также можете задать *полное* имя, содержащее префикс и полное имя хоста. Это можно сделать с помощью опции командной строки `--name`:

```
$ iex --name node1@127.0.0.1
 iex(node1@127.0.0.1)1>  ◄──────── Полное имя узла
```

Можно использовать и символьные имена:

```
$ iex --name node1@some_host.some_domain
iex(node1@some_host.some_domain)1>
```

Имя узла играет важную роль при установлении соединения. Напоминаю, короткое имя имеет формат `arbitrary_prefix@host`, а длинное – `arbitrary_prefix@host.domain`. Это имя идентифицирует на компьютере экземпляр BEAM. Вторая часть имени (`host` или `host.domain`) должна указывать на IP-адрес компьютера, на котором запущен этот экземпляр. При попытке на `node1` установить соединение с `node2@some_host.some_domain` хост node1 должен получить IP-адрес хоста `some_host.some_domain`.

Также важно заметить, что устанавливать соединение можно только между узлами с одинаковым типом имени. Другими словами, если у одного узла короткое имя, а у другого – полное, то установить соединение между ними не получится.

12.3.2. Файлы cookie

Чтобы два узла подключились друг к другу, они должны иметь `cookie` – своеобразный пароль, проверяемый во время установки соединения. Когда вы запускаете первый экземпляр BEAM, генерируется случайный файл cookie (.erlang.cookie file), который сохраняется в корневую папку. По умолчанию у всех узлов, запущенных на этом компьютере, будет этот файл.

Получить cookie можно с помощью функции `Node.get_cookie/0`:

```
iex(node1@localhost)1> Node.get_cookie()
:JHSKSHDYEJHDKEDKDIEN
```

Обратите внимание, что внутренним представлением cookie является атом. Узел, запущенный на другом компьютере, имеет иной cookie, поэтому установить связь между двумя узлами на разных компьютерах просто так не получится – необходимо сделать так, чтобы они использовали одинаковый cookie. Это делается вызовом функции `Node.set_cookie/1` на всех узлах, которые вы хотели бы связать:

```
iex(node1@localhost)1> Node.set_cookie(:some_cookie)

iex(node1@localhost)2> Node.get_cookie()
:some_cookie
```

Также для этого можно указать опцию `--cookie` при старте системы:

```
$ iex --sname node1@localhost --cookie another_cookie
iex(node1@localhost)1> Node.get_cookie()
:another_cookie
```

Таким образом можно обеспечить базовый уровень защиты данных и создать кластер с частичными связями. Пусть вам необходимо связать узлы А и Б, а также Б и В, но при этом вы не хотите, чтобы узел А был связан с узлом В. Этого можно достичь путем установки различных cookie для каждого из узлов и последующего вызова на узлах А и В функции `Node.set_cookie/2`, позволяющей явным образом задать различные cookie для подключения к тем или иным узлам.

12.3.3. Скрытые узлы

К этому моменту уже очевидно, что практически все операции с узлами распространяются на весь кластер. Чаще всего абсолютно все подключенные узлы считаются частью кластера, но в отдельных случаях это не требуется. Существуют различные инструменты, которые позволяют подключиться к удаленному узлу и взаимодействовать с ним. Простые примеры – запуск локального узла в качестве удаленной оболочки для другого узла или использование узла в качестве инструмента сбора данных, когда он подключается к другому узлу, собирает все возможные метрики и выводит результат на экран.

Подобные узлы являются вспомогательными и не должны быть частью основного кластера. Обычно их делают невидимыми для других узлов, и для этих целей используется *скрытое* подключение. Когда вы запускаете экземпляр BEAM с аргументом `--hidden`, этот узел не показывается в списке подключений других узлов (и наоборот).

Имейте в виду, что скрытые узлы также находятся в этом списке, но под меткой `hidden`. Их можно получить с помощью явного вызова `Node.list([:hidden])`; вызов `Node.list([:connected])` возвратит все подключенные узлы, включая скрытые, а `Node.list([:visible])` – только видимые. Если вы хотите выполнить операцию, доступную для всех узлов кластера, используйте опцию `:visible`.

Сервисы, предоставляемые модулями `:global`, `:rpc` и `:pg2`, не распространяются на скрытые узлы. Регистрация глобального псевдонима на одном узле никак не повлияет на скрытые узлы, и наоборот.

12.3.4. Фаерволы

С учетом того, что все узлы взаимодействуют через TCP-подключение, необходимо предоставить открытые порты для других компьютеров. Когда один узел попытается подключиться к другому узлу на удаленном компьютере, ему предстоит взаимодействие с двумя компонентами, как показано на рис. 12.4.

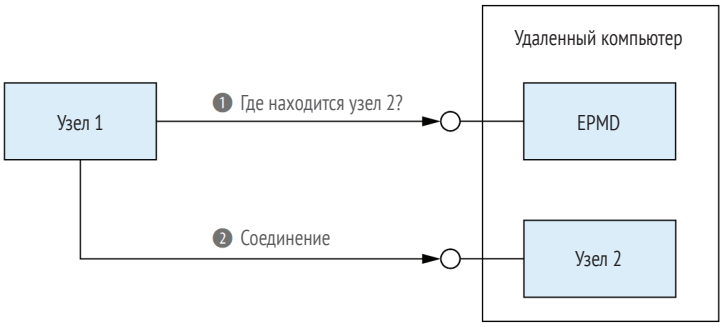


Рис. 12.4 ❖ Подключение к удаленному узлу

Первый компонент – Erlang Port Mapper Daemon (EPMD) – процесс ОС, запускаемый автоматически при запуске первого узла Erlang на компьютере-хосте. Данный компонент необходим для получения имени узлов – он знает имена всех узлов BEAM, запущенных на хосте. При установке соединения между двумя узлами на одном компьютере сначала отправляется запрос EPMD, для того чтобы узнать, какой порт прослушивает целевой узел, и уже после происходит подключение. EMPD прослушивает порт 4369, и он всегда должен быть доступен для удаленных компьютеров.

Кроме того, каждый узел прослушивает случайный порт, который также должен быть доступным, потому что именно он используется для установки соединения между двумя узлами. Однако из-за случайности порта невозможно определить правила фаервола.

Но решение есть: вы можете задать определенное количество портов, которые узлы будут прослушивать. Просто определите переменные окружения `inet_dist_listen_min` и `inet_dist_listen_max` приложения `kernel` в командной строке:

```
$ iex                                              \
  --erl '-kernel inet_dist_listen_min 10000'       \
  --erl '-kernel inet_dist_listen_max 10100'       \
  --sname node1@localhost
```

Задание диапазона портов

Узел `node1` будет прослушивать первый доступный порт из заданного диапазона. Вы можете определить одно и то же значение для обоих параметров, если уверены, что конфликтные ситуации не возникнут. Тогда один порт будет использоваться наиболее эффективно.

Чтобы проверить порты всех узлов на компьютере-хосте, вызовите функцию `:net_adm .names/0`:

```
iex(node1@localhost)1> :net_adm.names()
{:ok, [{'node1', 10000}]}
```

Альтернативным решением будет выполнить команду `epmd -names` в командной строке ОС.

Итак, чтобы обойти фаервол, необходимо открыть порт 4369 (EPMD) и диапазон портов, которые будут прослушивать ваши узлы.

Безопасность

Среди механизмов защиты в распределенных системах представлен только пароль cookie. Присоединяясь к удаленному узлу, на нем можно выполнять любые операции, включая системные команды. Если у удаленного узла есть права доступа root, то у подключенного узла есть полный доступ к удаленному хосту.

Нужно иметь в виду, что распределенная модель Erlang спроектирована для работы в проверенной среде. Это означает, что на этапе промышленной эксплуатации экземпляры BEAM должны быть запущены с минимальными правами. Кроме того, не следует делать их доступными в интернете. Если вам необходимо установить соединение с узлами других сетей, рассмотрите в качестве варианта переход на протокол соединения SSL. В официальной документации Erlang есть немного информации по этому поводу (**http://erlang.org/doc/apps/ssl/ssl_distribution.html**).

Выводы

- Распределенные системы могут повысить отказоустойчивость, устраняя риск отказа единой точки.
- Кластеризация открывает возможности масштабирования, позволяя распределить нагрузку на несколько компьютеров.
- Кластеры на основе BEAM состоят из узлов, называемых экземплярами BEAM, которые могут взаимодействовать друг с другом после установки соединения между ними.
- Два узла взаимодействуют посредством TCP-соединения, при разрыве которого разрывается связь и между узлами.
- Главным примитивом распределенных вычислений является процесс. Обмен сообщениями работает одинаково, независимо от расположения процесса. К зарегистрированному удаленно процессу можно обратиться с помощью `{alias, node_name}`.
- Многие сервисы высокого уровня, построенные на основе этих примитивов, доступны в модулях `:global`, `:rpc` и `GenServer`.
- При реализации взаимодействия между узлами используйте синхронные вызовы вместо асинхронных.
- Всегда продумывайте поведение системы в случае возникновения раскола кластера.

Глава 13

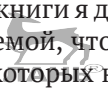
Запуск системы

В главе рассматривается:

- запуск системы с помощью инструментов Elixir;
- OTP-релизы;
- анализ поведения системы.

Вы потратили достаточное количество времени на разработку распределенной системы, и теперь пора подготовить ее к промышленной эксплуатации. Существует несколько способов запуска системы, но основная идея у них всех одна и та же – скомпилировать код и его зависимости, запустить экземпляр BEAM и убедиться, что все созданные после компиляции файлы находятся по заданным путям загрузки, после чего запустить приложение OTP и его зависимости из этого экземпляра BEAM. Система считается запущенной в момент запуска приложения OTP.

В данной главе будут рассмотрены два способа запуска системы – инструменты Elixir (по большей части `mix`) и OTP-релизы. В конце главы и книги я дам несколько советов относительно взаимодействия с запущенной системой, чтобы вы могли обнаружить и проанализировать ошибки, возникновение которых неизбежно во время выполнения.



13.1. Запуск системы с помощью инструментов Elixir

Независимо от способа, который вы выберете для запуска системы, порядок действий будет одинаковым. Запуск системы включает в себя следующие действия:

1) компиляция всех модулей. Соответствующие файлы с расширением .beam должны находиться на диске (как уже объяснялось в разделе 2.7). То же самое касается и ресурсных файлов (.app) всех приложений OTP, необходимых для работы системы;
2) запуск экземпляра BEAM и задание путей загрузки всех файлов из п. 1;
3) запуск необходимых приложений OTP.

Пожалуй, самый простой способ – запустить систему с помощью стандартных инструментов Elixir. Это довольно просто, и вы уже знакомы с некоторыми аспектами инструментов командной строки `mix`, `IEx` и `elixir`. Вы уже использовали IEx для запуска системы и взаимодействия с ней. При запуске системы с помощью команды `iex -S mix` выполняются все указанные выше действия.

На этапе промышленной эксплуатации простой запуск через оболочку IEx не всегда уместен, и иногда систему лучше запускать в качестве фонового процесса. В этом вам помогут команды mix и elixir.

13.1.1. Использование команд mix и elixir

До этого момента вы запускали систему с помощью команды iex -S mix. Есть еще одна команда – *mix run --no-halt*, запускающая экземпляр BEAM и приложения OTP со всеми зависимостями. Параметр --no-halt дает инструменту mix указание запустить экземпляр BEAM бессрочно:

```
$ mix run --no-halt    <-- Запуск системы без оболочки IEx

Starting to-do cache
```

Важное отличие команды mix run от iex -S mix состоит в том, что она не запускает интерактивную оболочку.

Следующий способ (команда elixir) немного сложнее:

```
$ elixir -S mix run --no-halt

Starting to-do cache
```

В нем чуть больше кода, и он позволяет запустить систему в фоновом режиме.

Флаг --detached необходимо указать, если вы хотите запустить систему в *обособленном режиме*. Процесс ОС будет отделен от терминала и не будет производить вывод в консоль (он будет отправляться в /dev/null). При запуске такой системы будет нелишним превратить экземпляр BEAM в узел, чтобы с ним можно было бы взаимодействовать в дальнейшем и при необходимости завершить. Следующее выражение запускает экземпляр BEAM в фоновом режиме:

```
$ elixir --detached --sname todo_system@localhost -S mix run --no-halt
```

Вы можете убедиться в этом, проверив, какие экземпляры BEAM запущены в вашей системе:

```
$ epmd -names

epmd: up and running on port 4369 with data:
name todo_system at port 51028    <-- Запущенный узел
```

Система работает, и вы уже можете ее использовать, например отправить HTTP-запрос на изменение списка дел.

Вы можете подключиться к запущенному экземпляру BEAM и взаимодействовать с ним. Существует возможность установить *удаленную оболочку* – нечто вроде сессии терминальной оболочки для экземпляра BEAM. В частности, с помощью опции --remsh можно запустить другой узел и использовать его в качестве оболочки узла todo_system:

```
$ iex --sname debugger@localhost --remsh todo_system@localhost --hidden

iex(todo_system@localhost)1>    <-- Оболочка, запущенная на узле todo_system
```

В данном примере запускается узел debugger, но оболочка работает в контексте узла todo_system. Это очень важный момент, так как теперь вы можете осуществ-

лять взаимодействие с вашей системой. Виртуальная машина BEAM предоставляет всевозможные сервисы, позволяющие отправлять запросы системе и отдельным ее процессам, как будет показано далее.

Обратите внимание, что узел `debugger` запускается как скрытый. Как говорилось в главе 12, это значит, что его не будет в списке, возвращаемом вызовом `Node.list` (или `Node .list([:this, :visible])`) на узле `todo_system`, и он не читается частью кластера.

Остановить работу системы вы можете с помощью функции *`System.stop`* (**https://hexdocs.pm/elixir/System.html#stop/1**), осуществляющей мягкий выход. Она закрывает все работающие приложения и завершает экземпляр BEAM:

```
iex(todo_system@localhost)1> System.stop()
```

При этом удаленная оболочка продолжит работать, и попытка выполнить любую другую команду приведет к возникновению ошибки:

```
iex(todo_system@localhost)2>
*** ERROR: Shell process terminated! (^G to start new job) ***
```

На этом этапе вы можете закрыть оболочку и проверить количество запущенных узлов BEAM:

```
$ epmd -names
epmd: up and running on port 4369 with data:
```

Чтобы остановить узел программно, используйте примитивы распределенных вычислений, описанные в главе 12. Простой пример:

```
if Node.connect(:todo_system@localhost) == true do
  :rpc.call(:todo_system@localhost, System, :stop, [])
  IO.puts "Node terminated."
else
  IO.puts "Can't connect to a remote node."
end
```

Вызов System.stop на удаленном узле

В данном случае устанавливается соединение с удаленным узлом и используется функция `:rpc.call/4` для вызова на нем функции `System.stop`.

Этот код можно сохранить в файл stop_node.exs (расширение .exs часто используется для скриптов Elixir) и выполнить скрипт в командной строке:

```
$ elixir --sname terminator@localhost stop_node.exs
```

При выполнении скрипта запускается отдельный экземпляр BEAM, в котором интерпретируется код. После выполнения кода экземпляр-хост завершается. Поскольку экземпляру скрипта необходимо подключиться к удаленному узлу (к тому, который вы хотите завершить), ему необходимо дать имя, чтобы превратить экземпляр BEAM в полноценный узел.

13.1.2. Выполнение скриптов

Уделим немного времени изучению скриптов и инструментов. Иногда существует необходимость создать инструмент командной строки, который выполняет определенные вычисления, возвращает результат и завершается. Проще всего сделать это с помощью скрипта.

Создайте обычный Elixir-файл с расширением .exs, реализуйте один или более модулей и вызовите функцию:

```
defmodule MyTool do
  ...

  def run do
    ...
  end
end

MyTool.run()  ◄──── Запуск инструмента
```

Выполнить данный скрипт можно с помощью команды `elixir my_script.exs`. Все определенные вами модули будут скомпилированы в оперативную память, а все выражения за пределами модулей будут интерпретированы, после чего скрипт завершится. Как вы понимаете, Elixir-скрипты сработают только в системе, в которой установлены действительные версии Erlang и Elixir.

Скрипт .exs подойдет для реализации простых инструментов, но для более сложного кода или при подключении сторонних библиотек в качестве зависимостей он не будет настолько эффективным. В таких случаях разумнее использовать проект `mix` и создавать полноценное OTP-приложение.

Но так как этой системе незачем функционировать продолжительное время, вам также нужно будет реализовать *запускающий* модуль, который будет производить вычисления и возвращать результат:

```
defmodule MyTool.Runner do
  def run do
    ...
  end
end
```

Теперь запустить инструмент можно с помощью команды `mix run -e MyTool.Runner.run`. Она запустит приложение OTP, вызовет функцию `MyTool.Runner.run/0` и завершит инструмент, как только эта функция выполнится.

И наконец, можно упаковать весь инструмент в *escript* – отдельный бинарный файл, собирающий все ваши файлы .beam, файлы .beam Elixir и запускающий код. Файл escript является полностью скомпилированным кроссплатформенным скриптом, для работы которого требуется только установка Erlang. Для получения более подробной информации обратитесь к документации команды `mix escript.build` (**https://hexdocs.pm/mix/Mix.Tasks.Escript.Build.html**).

13.1.3. Компиляция для промышленной эксплуатации

В главе 11 вы узнали, что существует конструкция под названием окружение mix – идентификатор времени выполнения, позволяющий включать в проект тот или иной код в зависимости от условий. По умолчанию используется окружение dev – окружение для разработки, а при запуске команды `mix test` код компилируется в окружении для тестирования.

Окружения mix используются для удобства работы с кодом во время разработки или тестирования. Например, с помощью функции *`Mix.env/0`* вы можете определить несколько разных вариантов одной и той же функции:

```
defmodule Todo.Database do
  case Mix.env() do
    :dev ->
      def store(key, data) do ... end

    :test ->
      def store(key, data) do ... end

    _ ->
      def store(key, data) do ... end
  end
end
```

Обратите внимание, как реализуется условность результата функции `Mix.env/0` на уровне модуля. Этот блок кода выполняется во время компиляции. Конечное определение функции `store/2` будет зависеть от окружения mix, которое вы зададите. В окружении dev вам может понадобиться дополнительное журналирование и сопоставительный анализ, тогда как в окружении test – альтернативная база данных и даже хранилище данных в памяти вроде публичной таблицы ETS.

Важно понимать, что функция `Mix.env/0` приобретает смысл только во время компиляции. Использовать ее во время выполнения программы нельзя.

В любом случае в коде могут присутствовать подобные условные определения функций, так что вы должны учитывать, что скомпилированный в окружении dev проект не полностью оптимизирован.

Чтобы запустить систему в промышленную эксплуатацию, задайте переменной окружения ОС MIX_ENV соответствующее значение:

```
$ MIX_ENV=prod elixir -S mix run --no-halt
```

После этого код и все зависимости скомпилируются повторно. Все файлы с расширением .beam сохранятся в папку _build/prod, и mix обеспечивает загрузку экземпляром BEAM файлов из этой папки.

> **СОВЕТ** Вы уже могли догадаться, что код, компилируемый по умолчанию (в окружении dev), не является оптимальным. Окружение dev делает разработку удобнее, но снижает производительность кода. Чтобы оценить поведение системы при высоких нагрузках, необходимо скомпилировать ее в окружении prod. Показатели, полученные в окружении dev, могут дать ложные представления об узких местах системы, и вы потратите время и силы на оптимизацию кода, не вызывающего проблем в промышленной эксплуатации.

Теперь вы знаете, как запускать систему с помощью инструментов `mix` и `elixir`. Эти довольно простые способы отлично вписываются в процесс разработки.

Однако у них есть некоторые серьезные недостатки. Во-первых, чтобы запустить проект с помощью `mix`, необходимо скомпилировать его, а это значит, что исходный код системы должен храниться на компьютере-хосте. Все зависимости также должны быть добавлены в проект и скомпилированы. Следовательно, на компьютер-хост придется установить все необходимые для компиляции инструменты, включая Erlang и Elixir, hex и, возможно, rebar, а также все остальные сторонние инструменты, интегрируемые в поток работы `mix`.

Например, если при разработке веб-сервера для управления пакетами JavaScript используется менеджер npm, то вам придется установить его на рабочий сер-

вер, тем самым занимая место на компьютере-хосте. Кроме того, если на одном компьютере запущено несколько систем, крайне сложно согласовать различные версии инструментов, необходимые для разных систем. К счастью, все не так безнадежно, ведь у вас есть OTP-релизы.

13.2. OTP-релизы

OTP-релиз – это независимая, скомпилированная, работоспособная система, состоящая из минимального набора OTP-приложений, необходимых основной системе. OTP-релиз может дополнительно включать минимальный набор бинарных Erlang-файлов времени выполнения, что делает его полностью самодостаточным. Релизы не содержат файлы с исходным кодом, файлы документации или тестов.

У этого подхода есть масса преимуществ. Во-первых, вы можете осуществлять сборку системы на том же компьютере, что и разработку, или собрать сервер, а на другие компьютеры перенести только итоговые бинарные файлы. Во-вторых, не требуется устанавливать какие-либо инструменты на хосте. Если в релиз будет внедрен минимум необходимых компонентов среды выполнения Erlang, то на рабочем сервере даже не придется устанавливать Elixir и Erlang. Все, что нужно для работы системы, будет поставляться в составе релиза. И наконец, релизы лежат в основе систематического обновления (и отката) системы в режиме онлайн, называемого в Erlang *управлением релизами*.

На уровне теории использовать релизы вроде бы совсем несложно. Необходимо скомпилировать основное приложение OTP со всеми его зависимостями и затем собрать все бинарные файлы в релиз вместе со средой выполнения Erlang. В стандартной поставке Erlang имеется все необходимое для этого, но при использовании этих инструментов приходится многое делать вручную. К счастью, существует такая библиотека, как *distillery* (**https://github.com/bitwalker/distillery**), которая значительно упрощает процесс сборки OTP-релизов.

13.2.1. Создание релиза с помощью distillery

В данном разделе вы увидите, что сборка полностью автономного релиза с помощью distillery – невероятно простая задача. Необходимо добавить библиотеку `distillery` в список зависимостей, создать файл конфигураций и затем создать релиз командой `mix release`.

Попробуем это сделать. Добавьте `distillery` в качестве зависимости, как показано в следующем листинге.

Листинг 13.1 ❖ Добавление `distillery` в список зависимостей

```
defmodule Todo.MixProject do
  ...

  defp deps do
    [
      ...
      {:distillery, "~> 2.0"}
```

```
    ]
  end
end
```

После этого можно вызвать `mix deps.get`, чтобы добавить зависимости в проект.

Теперь необходимо настроить сборку релиза. Создайте папку rel, а в ней – файл конфигураций релиза config.exs. Для получения этого файла выполните команду `mix release.init`.

В целях упрощения задачи будем использовать неполную версию файла rel/config.exs, содержащего настройки только для окружения prod. Его содержимое приведено в листинге ниже.

Листинг 13.2 ❖ Настройка релиза (todo_release/rel/config.exs)

```
use Mix.Releases.Config, default_environment: :prod

environment :prod do
  set(include_erts: true)
  set(include_src: false)
  set(cookie: :todo)
end

release :todo do
  set(version: current_version(:todo))
end
```

Этот файл – укороченная версия файла, получаемого после выполнения команды `mix release.init`. В нем указано, что среда выполнения Erlang должна быть включена в релиз, что делает его полностью самодостаточным.

Теперь можно осуществить сборку релиза с помощью команды `MIX_ENV=prod mix release`. Так как данный релиз создается для промышленной эксплуатации, необходимо установить это окружение по умолчанию для задачи `release` в файле mix.exs, как показано в следующем листинге.

Листинг 13.3 ❖ Установка окружения prod для задачи release (todo_release/mix.exs)

```
defmodule Todo.MixProject do
  ...

  def project do
    [
      ...
      preferred_cli_env: [release: :prod]
    ]
  end
```

Параметр `:preferred_cli_env` представляет собой ключевой список, в котором ключи – это имена задач (выраженные атомами), а значения – устанавливаемые для каждой задачи окружения mix.

Теперь все готово. Выполните команду `mix release`, и она скомпилирует ваш проект в окружении prod, после чего создаст релиз:

```
$ mix release

...

==> Assembling release..
```

```
==> Building release todo:0.1.0 using environment prod
==> Including ERTS 10.0.8
==> Packaging release..
==> Release successfully built!
    You can run it in one of the following ways:
      Interactive: _build/prod/rel/todo/bin/todo console
      Foreground: _build/prod/rel/todo/bin/todo foreground
      Daemon: _build/prod/rel/todo/bin/todo start
```

Релиз будет находиться в подпапке по адресу _build/prod/rel/todo/. Содержимое релиза мы разберем позже, а сейчас давайте посмотрим, как его можно использовать.

13.2.2. Использование релиза

Главный инструмент, используемый для взаимодействия с релизом, – это скрипт оболочки, который находится в папке _build/prod/rel/todo/bin/todo. С его помощью можно выполнять следующие задачи:

- запуск системы и оболочки IEx в приоритетном режиме;
- запуск системы в виде фонового процесса;
- останов работающей системы;
- подключение удаленной оболочки к запущенной системе.

Самый простой способ проверить работоспособность релиза – запустить систему в приоритетном режиме вместе с оболочкой IEx:

```
$ _build/prod/rel/todo/bin/todo console

Starting to-do cache
iex(todo@127.0.0.1)1>
```

Из строки приглашения оболочки видно, что текущий релиз автоматически запускается в виде узла системы todo. Инструмент distillery по умолчанию использует имя приложения в качестве имени узла.

Хотелось бы отметить, что данный релиз больше не зависит от наличия Erlang и Elixir в вашей системе. Он полностью автономный: вы можете скопировать содержимое подпапки _build/prod/rel/todo на любой другой компьютер, и даже если на нем не установлены Erlang и Elixir, релиз все равно будет работать. Конечно, так как релиз содержит бинарные файлы среды выполнения Erlang, на этом компьютере должна быть такая же ОС и архитектура.

Чтобы запустить систему в качестве фонового процесса, используйте аргумент start:

```
$ _build/prod/rel/todo/bin/todo start
```

Фоновый и обособленный процессы – это разные вещи. В обособленном режиме система запускается с помощью инструмента run_erl (**http://erlang.org/doc/man/run_erl.html**). Он перенаправляет стандартный вывод в файл журнала, находящийся в папке _build/prod/rel/todo/var/log. Этот файл впоследствии используется для анализа консольного вывода системы.

Когда система работает в фоновом режиме, вы можете запустить для нее удаленную оболочку:

```
$ _build/prod/rel/todo/bin/todo remote_console
iex(todo@127.0.0.1)1>
```

Как видите, запущенная сессия оболочки IEx связана с рабочим узлом. Нажав **Ctrl-C** дважды, вы остановите удаленную оболочку, но узел todo продолжит работать.

Также существует возможность подключиться к оболочке работающего процесса. На первый взгляд этот способ похож на работу удаленной оболочки, но у него есть одно важное преимущество – перехват стандартного вывода рабочего узла. Весь вывод рабочего узла, осуществленный, например, через IO.puts, становится видимым в подключенном процессе (чего не происходит при использовании удаленной оболочки).

Чтобы подключиться к оболочке, используйте аргумент attach:

```
$ _build/prod/rel/todo/bin/todo attach
iex(todo@127.0.0.1)1>
```

Будьте осторожны с этим способом. В отличие от удаленной оболочки, подключенная связывается с работающим узлом посредством канала ОС. Это означает, что при останове оболочки двойным нажатием **Ctrl-C** завершится и работающий узел. Чтобы отключить оболочку от узла, необходимо нажать **Ctrl-D**.

Если вы хотите остановить систему, работающую в фоновом режиме, используйте аргумент stop:

```
$ _build/prod/rel/todo/bin/todo stop
```

Скрипт todo может выполнять и другие задачи. Чтобы вывести справку, выполните команду _ build/prod/rel/todo/bin/todo без указания аргументов. В стандартном выводе появится текст справки. Для получения более подробной информации об этом обратитесь к официальной документации distillery (**https://hexdocs.pm/distillery**).

13.2.3. Структура релиза

Любой автономный релиз содержит:

- скомпилированные OTP-приложения, необходимые для работы системы;
- файл с аргументами, которые будут переданы виртуальной машине;
- скрипт загрузки, описывающий необходимые к запуску приложения OTP;
- файл конфигураций, содержащий переменные окружения для OTP-приложений;
- вспомогательный скрипт оболочки для запуска, останова системы и взаимодействия с ней;
- бинарные файлы среды выполнения Erlang.

Все эти файлы находятся в папке _build/prod/rel/todo. Рассмотрим подробнее некоторые наиболее важные составляющие релиза.

Скомпилированные бинарные файлы

Скомпилированные версии всех необходимых приложений находятся в папке _build/prod/rel/todo/lib:

```
$ ls _build/prod/rel/todo/lib
artificery-0.2.6
asn1-5.0.6
compiler-7.2.4
cowboy-1.1.2
cowlib-1.0.2
crypto-4.3.2
distillery-2.0.10
elixir-1.7.3
iex-1.7.3
kernel-6.0.1
logger-1.7.3
mime-1.2.0
mix-1.7.3
plug-1.4.3
poolboy-1.5.1
public_key-1.6.1
ranch-1.3.2
runtime_tools-1.13
sasl-3.2
ssl-9.0.1
stdlib-3.5.1
todo-0.1.0
```


Этот список включает все зависимости времени выполнения – как прямые (указанные в файле mix.exs), так и косвенные (зависимости зависимостей). Также в релиз автоматически добавляются OTP-приложения `kernel`, `stdlib` и `elixir` – основные приложения, необходимые для работы любой системы, разработанной на Elixir. Кроме этого, в релиз включается и приложение `iex`, предоставляющее возможность запуска удаленной оболочки.

Каждая из этих папок содержит подпапку ebin, в которой находятся скомпилированные бинарные файлы и файл с расширением .app. Папка каждого OTP-приложения также включает папку priv с дополнительными файлами приложения.

> **СОВЕТ** Если вам необходимо добавить в релиз дополнительные файлы, создайте папку priv в корневом каталоге проекта. Эта папка автоматически появляется в релизе под папкой приложения. Чтобы обратиться к определенному файлу этой папки, вызовите `Application.app_dir(:an_app_name, "priv")` для получения абсолютного пути папки.

Объединяя все необходимые приложения OTP, вы делаете релиз независимым. Поскольку система включает все требуемые бинарные файлы (в том числе и стандартные библиотеки Elixir и Erlang), для ее работы на компьютере-хосте все готово.

Это можно проверить, взглянув на пути загрузки:

```
$ _build/prod/rel/todo/bin/todo console

iex(todo@127.0.0.1)1> :code.get_path()   <-- Получение списка путей загрузки

['ch13/todo_release/_build/prod/rel/todo/lib/todo-0.1.0/consolidated',
 'ch13/todo_release/_build/prod/rel/todo/lib/kernel-6.0.1/ebin',
 'ch13/todo_release/_build/prod/rel/todo/lib/stdlib-3.5.1/ebin',
 'ch13/todo_release/_build/prod/rel/todo/lib/compiler-7.2.4/ebin',
```

```
 'ch13/todo_release/_build/prod/rel/todo/lib/elixir-1.7.3/ebin',
 'ch13/todo_release/_build/prod/rel/todo/lib/artificery-0.2.6/ebin',
 'ch13/todo_release/_build/prod/rel/todo/lib/runtime_tools-1.13/ebin',
 'ch13/todo_release/_build/prod/rel/todo/lib/distillery-2.0.10/ebin',
 'ch13/todo_release/_build/prod/rel/todo/lib/mime-1.2.0/ebin',
 'ch13/todo_release/_build/prod/rel/todo/lib/crypto-4.3.2/ebin',
 'ch13/todo_release/_build/prod/rel/todo/lib/logger-1.7.3/ebin',
 'ch13/todo_release/_build/prod/rel/todo/lib/plug-1.4.3/ebin',
 'ch13/todo_release/_build/prod/rel/todo/lib/cowlib-1.0.2/ebin',
 'ch13/todo_release/_build/prod/rel/todo/lib/asn1-5.0.6/ebin',
 'ch13/todo_release/_build/prod/rel/todo/lib/public_key-1.6.1/ebin',
 'ch13/todo_release/_build/prod/rel/todo/lib/ssl-9.0.1/ebin',
 'ch13/todo_release/_build/prod/rel/todo/lib/ranch-1.3.2/ebin',
 'ch13/todo_release/_build/prod/rel/todo/lib/cowboy-1.1.2/ebin',
 'ch13/todo_release/_build/prod/rel/todo/lib/poolboy-1.5.1/ebin',
 'ch13/todo_release/_build/prod/rel/todo/lib/todo-0.1.0/ebin',
 'ch13/todo_release/_build/prod/rel/todo/lib/sasl-3.2/ebin',
 'ch13/todo_release/_build/prod/rel/todo/lib/iex-1.7.3/ebin',
 'ch13/todo_release/_build/prod/rel/todo/lib/mix-1.7.3/ebin']
```

Обратите внимание, что все пути ведут к папке релиза. Если же запустить систему простым способом, выполнив команду `iex -S mix` и вызвав функцию `:code.get_path/0`, вы увидите более длинный список путей загрузки, и какие-то из них будут указывать на папку сборки, а другие – на папки установки Elixir и Erlang. Таким образом, вам должно быть очевидно, что данный релиз является самодостаточным. Среде выполнения понадобятся лишь модули, находящиеся в папке релиза.

В релиз также входит минимальный набор бинарных файлов Erlang. Они находятся по адресу _build/prod/rel/todo/erts-X.Y, где X.Y соответствует номеру текущей версии среды выполнения (не связанной с номером версии Erlang). Релиз содержит среду выполнения Erlang, а значит, является полностью независимым. Это также позволяет запускать на одном компьютере несколько систем, при разработке которых использовались разные версии Elixir и Erlang.

Настройки

Файлы конфигураций находятся в папке _build/prod/rel/todo/releases/0.1.0, где цифры 0.1.0 соответствуют версии приложения `todo` (указанной в файле mix.exs). Наибольший интерес в этой папке представляют файлы vm.args и sys.config.

В файле vm.args задаются флаги для среды выполнения Erlang, например упомянутый выше флаг `+P`, устанавливающий максимальное число запущенных процессов. Некоторые основные настройки, такие как имя узла и пароль cookie, генерируются инструментом `distillery`. Вы также можете создавать свои файлы vm.args и передавать их инструменту `distillery`. Узнать об этом подробнее можно в официальной документации (**https://hexdocs.pm/distillery/configuration.html**).

Sys.config – это файл Erlang, содержащий переменные окружения OTP, указанные в файлах mix.exs и config.exs. Например, в главе 11 вы реализовали возможность изменения HTTP-порта с помощью переменной окружения и установили ей значение 5454 в файле config.exs. Эта настройка передается и в файл sys.config.

При старте системы до запуска приложений OTP соответствующим переменным окружения присваиваются значения, указанные в файле sys.config. Следовательно, это позволяет менять настройки без осуществления повторной сборки релиза. Например, чтобы установить другой HTTP-порт, исправьте значение в файле sys.config на нужном компьютере и перезапустите систему.

Сжатые пакеты релиза

Внимательно посмотрев на содержимое папки _build/prod/rel/todo/releases/0.1.0, вы увидите сжатый архив todo.tar.gz, который по сути является сжатой версией всего релиза. Для развертывания системы на другом компьютере просто скопируйте этот файл, распакуйте его и запустите систему командой `bin/todo start`.

Данный файл необходим не только для удобства развертывания, но и для обновления системы в режиме онлайн. После внесения изменений в код увеличьте номер версии в файле mix.exs и заново соберите релиз. Новая его версия будет помещена в другую папку (например, _build/prod/rel/todo/releases/0.2.0). Затем просто загрузите этот файл в соответствующую подпапку работающей системы и вызовите `bin/todo upgrade "0.0.2"`. Ваша система обновится на лету без перезапуска. Более подробно об этом читайте в документации `distillery` (**https://hexdocs.pm/distillery/walkthrough.html#building-an-upgrade-release**).

С онлайн-обновлениями все не так просто

В небольшом проекте онлайн-обновление выглядит на удивление простым, особенно с помощью инструмента `distillery`. Однако обновление более сложных проектов требует гораздо больше манипуляций. Не забывайте, что система на основе BEAM имеет множество серверных процессов, состояние которых может храниться в таблицах ETS. Для обновления такой системы необходимо реализовать обратный вызов `code_changegen_server`, в котором должно производиться изменение старого состояния процесса на новое. Более того, в каких-то случаях вам понадобится перестроить конфигурацию дерева супервизоров или даже остановить/перезапустить некоторые OTP-приложения. Чтобы обновление прошло должным образом, вы должны очень хорошо продумать каждое изменение и как следует его протестировать. Поэтому перед этим стоит прикинуть, насколько система готова к небольшим простоям, вызванным перезапуском процессов BEAM.

Если ваша система является распределенной, вы можете проводить обновление узлов постепенно, один за одним, что позволит системе продолжать работать в это время. Онлайн-обновление следует рассматривать только в том случае, когда простого перезапуска недостаточно для достижения конкретной цели. Это описано более подробным образом в документациях distillery и Erlang (`http://erlang.org/doc/design_principles/release_handling.html`).

Мелкоструктурная сборка релиза

Инструменты вроде `distillery` осуществляют сборку всего релиза на основном компьютере в одно действие. Для большинства задач этого вполне достаточно. Но, например, в случае если вам требуется развернуть систему на большом количестве разных компьютеров, внедренная среда выполнения Erlang сработает

не везде, так как она привязана к определенной ОС и архитектуре. Скорее всего, вам понадобится осуществить сборку только файлов с расширением .beam и файлов конфигурации на основном компьютере, а завершить сборку на каждом из оставшихся компьютеров кластера отдельно. Вы даже можете загружать код для узлов с удаленного компьютера, а значит, релиз необязательно должен содержать бинарные файлы приложения.

Как видите, вариантов предостаточно. При наличии определенных требований придется все же повозиться со сборкой некоторых частей вручную. Для этого как следует изучите дополнительную информацию о релизах из документаций Erlang (**http://erlang.org/doc/design_principles/release_structure.html**), `:systools` (**http://erlang.org/doc/man/systools.html**) и `:reltool` (**http://erlang.org/doc/man/reltool.html**).

На этом мы и закончим тему релизов. Теперь поговорим о том, каким способом можно провести анализ поведения системы после ее запуска.

13.3. Анализ поведения системы

Даже после того, как вы осуществили сборку системы и запустили ее в промышленную эксплуатацию, ваша работа не окончена. Вы обязательно столкнетесь с ошибками, недостаточной оптимизацией кода, потреблением слишком большого количества ресурсов. Конечно, грамотно реализованная система способна сама восстанавливаться после ошибок и справляться с высокой нагрузкой. Но даже в этом случае необходимо докопаться до сути возникающих проблем и по возможности что-то исправить.

В конкурентных и распределенных системах не всегда легко обнаружить и понять природу возникающих ошибок. Эта тема достойна отдельной книги, и, кстати, такая книга уже есть – «Stuff Goes Bad: Erlang in Anger» (**http://www.erlang-in-anger.com**), автор Фред Геберт (Fred Hébert). В этом разделе приводится лишь краткий обзор некоторых стандартных способов анализа сложных систем на основе BEAM, но если у вас в планах запускать код на Elixir/Erlang в промышленную эксплуатацию, в какой-то момент вам понадобится копнуть глубже, и данная книга будет отличным выбором.

13.3.1. Отладка

Отладка не обязательно относится к работающей системе, но точно заслуживает внимания. Вы удивитесь, но стандартная пошаговая отладка редко используется в Erlang (в поставку которого входит отладчик с графическим интерфейсом, см. **http://erlang.org/doc/apps/debugger/debugger_chapter.html**). Все потому, что в конкурентной системе много действий происходят одновременно, и ее невозможно отладить привычным способом. Представьте, что вы установили точку останова в каком-либо процессе. Что произойдет с другими процессами, когда они обнаружат ее? Должны ли они продолжить работу или так же встать на паузу? Должны ли другие процессы сделать один шаг при переходе с точки останова на следующую строку кода? Как должны быть обработаны тайм-ауты? Как видите, при использовании классического способа отладки в распределенных системах с высокой степенью конкурентности возникает множество вопросов.

Следует рассмотреть более подходящие для анализа конкурентных систем стратегии, а именно журналирование и трассировку. При обнаружении проблемы необходимо получить как можно больше информации, чтобы найти причину ее возникновения.

Хорошая новость заключается в том, что журналирование доступно «из коробки» в виде Elixir-приложения *logger* (**https://hexdocs.pm/logger/Logger.html**). В частности, каждый раз, когда какой-либо OTP-процесс (по типу `GenServer`) аварийно завершается, выводится ошибка вместе с ее стек-трейсом. Стек-трейс также содержит информацию о файле и строке, в которых эта ошибка возникла, что даст первый толчок в анализе сбоя.

В некоторых случаях стек-трейс не дает очевидной информации о причине ошибки, соответственно, требуется получить какие-то дополнительные данные. На этапе разработки для этого служит вспомогательная функция `IO.inspect`. Напоминаю, что она принимает выражение, возвращает его результат и выводит его на экран. Вы можете вставить эту функцию в любую часть своего кода (или добавить ее в конвейер вызовов с помощью оператора `|>`), что абсолютно не повлияет на поведение программы. Такой элементарный способ поможет вам быстро определить причину ошибки. Я часто использую его при анализе только что написанного кода, для того чтобы понять, каким образом передаются значения в проблемные места. Как только проблема исчезает, вызовы `IO.inspect` можно удалить.

Еще одна полезная функция – это функция *pry*, позволяющая в оболочке `IEx` на время остановить выполнение программы и проверить состояние системы, а именно переменные в области видимости. Более подробно о ее особенностях можно узнать из документации *`IEx.pry/1`* (**https://hexdocs.pm/iex/IEx.html#pry/1**).

На официальном сайте Elixir (**https://elixir-lang.org/getting-started/debugging.html**) также представлен обзор основных методов отладки.

Разумеется, не стоит забывать про автоматизированное тестирование. Изолированное тестирование отдельных частей системы позволит достаточно быстро обнаружить и исправить возникшие ошибки.

Стоит также отметить парочку полезных средств для сопоставительных испытаний и профилирования. Самое простое из них – это функция `:timer.tc/1` (**http://erlang.org/doc/man/timer.html#tc-1**), принимающая анонимную функцию и возвращающая ее результат и время ее выполнения в микросекундах.

Существует также немного более структурированная библиотека для сопоставительного анализа – Benchfella (**https://github.com/alco/benchfella**), отлично интегрируемая с `mix` и предоставляющая фреймворк для генерации автоматизированных эталонных тестов. Еще одна часто используемая для этого библиотека – Benchee (**https://github.com/PragTob/benchee**).

В дополнение к этому вместе с Erlang поставляется несколько инструментов профилирования: `cprof`, `eprof` и `fprof`. Elixir предоставляет следующие задачи `mix` для запуска каждого из этих инструментов:

- `mix profile.cprof` (**https://hexdocs.pm/mix/Mix.Tasks.Profile.Cprof.html**);
- `mix profile.eprof` (**https://hexdocs.pm/mix/Mix.Tasks.Profile.Eprof.html**);
- `mix profile.fprof` (**https://hexdocs.pm/mix/Mix.Tasks.Profile.Fprof.html**).

В данной книге они не будут подробно рассмотрены. Если дело дошло до профилирования, советую обратиться к официальной документации этих задач `mix`, а также документации Erlang (**http://erlang.org/doc/efficiency_guide/profiling.html**).

13.3.2. Журналирование

На этапе промышленной эксплуатации больше не следует использовать функцию `IO.inspect`. Более правильным решением будет запись различных данных об ошибках в журнал. В этом вам поможет приложение `logger`, которое по умолчанию включается в список зависимостей при создании проекта `mix`. Как уже отмечалось выше, `logger` автоматически перехватывает различные отчеты BEAM о тех же ошибках при отказе процессов.

По умолчанию отчеты об ошибках выводятся в консоль. Если вы запускаете систему в виде релиза, стандартный вывод будет перенаправлен в папку log в корневом каталоге релиза, и вы сможете впоследствии найти и проанализировать ошибки.

Вы также можете разработать свой бэкенд приложения `logger` для записи данных в файл syslog или отправки отчетов на другой компьютер. Подробнее об этом можете почитать в документации `logger` (**https://hexdocs.pm/logger/Logger.html**).

13.3.3. Взаимодействие с системой

Большим преимуществом среды выполнения Erlang является возможность подключиться к запущенному узлу и взаимодействовать с ним различными способами. Вы можете отправлять процессам сообщения и перезапускать различные процессы (в том числе и супервизоры) или приложения OTP, а также заставить виртуальную машину перезагрузить код для какого-либо модуля.

Помимо этого, получить данные о системе и отдельных ее процессах можно с помощью многочисленных встроенных функций. Например, вы можете запустить удаленную оболочку и воспользоваться функциями `:erlang.system_info/1` и `:erlang.memory/0` для получения информации о выполнении кода.

С помощью функции `Process.list/0` можно получить список всех процессов и затем обратиться к каждому из них функцией `Process.info/1`, возвращающей информацию об использованной памяти и общем числе операций, выполненных процессом (в Erlang они называются *редукциями*). Конечно, эти сервисы уступают дорогу инструментам, способным подключиться к запущенной системе и вывести информацию о ней в виде графического интерфейса.

Один из них – приложение `observer`, с которым вы познакомились в главе 11. Оно работает только на тех хостах, в операционной системе которых представлен оконный интерфейс. Как правило, основной сервер его не имеет, но вы можете запустить `observer` локально и получать данные с удаленного узла.

Посмотрим на это в действии. Запустите систему в фоновом режиме и дополнительный узел для работы приложения `observer`. Оно подключится к удаленному узлу, соберет по нему данные и выведет их в графический интерфейс.

В промышленной системе необязательно запускать приложение `observer`, но она должна содержать модули, собирающие данные для приложения `observer`, работающего на удаленном узле. Эти модули входят в состав приложения `runtime_tools`, которое вам следует включить в свой релиз. Это делается с помощью указания опции `:extra_applications` в файле mix.exs, как показано в листинге ниже.

Листинг 13.4 ❖ Включение `runtime_tools` в релиз (todo_release/mix.exs)

```
defmodule Todo.MixProject do
  ...

  def application do
    [
      extra_applications: [:logger, :runtime_tools],
          ...
    ]
  end

  ...
end
```

Включение runtime_tools в OTP-релиз

После опции `:extra_applications` в зависимостях указываются предустановленные OTP-приложения Elixir и Erlang. По умолчанию Elixir-приложение `logger` вносится в этот список при создании нового проекта с помощью инструмента `mix`.

> **ПРИМЕЧАНИЕ** Обратите внимание, что, в отличие от функции deps из файла mix.exs, опция `:extra_applications` служит другой цели. С помощью deps указываются сторонние зависимости, которые необходимо загрузить и скомпилировать, а с помощью `:extra_applications` – приложения Elixir и Erlang, которые уже скомпилированы на диске в составе стандартных пакетов Elixir и Erlang. Код этих зависимостей не требуется загружать и компилировать, но эти зависимости все равно необходимо указывать, чтобы они наверняка входили в OTP-релиз.

Добавив `runtime_tools` в релиз, вы можете запускать `observer` удаленно. Пришло время перейти к примеру. Запустите свою систему в фоновом режиме:

```
$ _build/prod/rel/todo/bin/todo start
```

Теперь запустите интерактивную оболочку в виде именованного узла, а затем и приложение `observer`:

```
$ iex --hidden --name observer@127.0.0.1 --cookie todo

iex(observer@127.0.0.1)1> :observer.start()
```

Можно видеть, что cookie нового узла задается в явном виде, чтобы он совпадал с уже используемым в системе паролем. Также узел запускается в скрытом режиме, как это делалось в примере из раздела 13.1.1. После запуска `observer` необходимо выбрать todo@127.0.0.1 в меню Nodes. После этого `observer` будет показывать данные о рабочем узле.

Стоит отметить, что приложения `observer` и `runtime_tools` реализованы на Erlang и полагаются на функции низшего уровня для получения и вывода данных различными способами. Вы также можете использовать любой другой фронтенд или даже написать свой собственный. Например, вы можете выбрать Wobserver (**https://github.com/shinyscorpion/wobserver**) – `observer` на основе веба, предоставляющий похожие функции, но поверх HTTP-интерфейса.

13.3.4. Трассировка

Можно также подключить трассировку по процессам и вызовам функций с помощью сервисов модулей `:sys` (**http://erlang.org/doc/man/sys.html**) и `:dbg` (**http://**

erlang.org/doc/man/dbg.html). Модуль `:sys` позволяет трассировать совместимые с OTP процессы, такие как `GenServer`.

Трассировка производится на стандартном выводе ошибки, поэтому необходимо именно подключиться к системе, а не использовать удаленную оболочку. Затем можно включить трассировку определенного процесса с помощью функции `:sys.trace/2`:

```
$ _build/prod/rel/todo/bin/todo attach

iex(todo@127.0.0.1)1> :sys.trace(Todo.Cache.server_process("bob"), true)
```

Информация о событиях, связанных с данным процессом, например полученные запросы, будет включена в стандартный вывод.

Теперь отправим HTTP-запрос по списку Bob:

```
$ curl "http://localhost:5454/entries?list=bob&date=2018-12-19"
```

В подключенной оболочке вы увидите следующие записи:

```
*DBG* {todo_server,<<"bob">>} got call {entries,
  #{'__struct__' => 'Elixir.Date', calendar => 'Elixir.Calendar.ISO',
    day => 19,month => 12, year => 2018}} from <0.983.0>}

*DBG* {todo_server,<<"bob">>} sent [] to <0.322.0>,
  new state {<<"bob">>, #{'__struct__' => 'Elixir.Todo.List',
    auto_id => 1, entries => #{}}}
```

Данный вывод может показаться нечитабельным, но если присмотреться, то вы увидите две записи трассировки: одна – для полученного синхронного запроса, а вторая – для отправленного ответа. В нем также присутствует полное состояние серверного процесса. Для вывода термов используется синтаксис Erlang.

Трассировка – очень мощный инструмент для анализа поведения работающей системы. Но увлекаться им не стоит, так как излишняя трассировка может снизить производительность системы. Если серверный процесс испытывает большую нагрузку или имеет слишком объемное состояние, виртуальная машина потратит очень большое количество времени на трассировку ввода-вывода, что может замедлить работу всей системы.

Получив определенную информацию о процессе, можно отключить трассировку:

```
iex(todo@127.0.0.1)1> :sys.trace(Todo.Cache.server_process("bob"), false)
```

Есть и другие не менее полезные сервисы модуля `:sys`, позволяющие получить состояние OTP-процесса (`:sys.get_state/1`) и даже изменить его (`:sys.replace_state/2`). Эти две функции следует использовать исключительно для отладки или ручных правок работающего кода. В обычном коде их вызывать не стоит.

Еще одно средство трассировки – это функция `:erlang.trace/3` (**http://erlang.org/doc/man/erlang.html#trace-3**), позволяющая подписаться на такие события системы, как передача сообщений или вызовы функций.

Вдобавок к этой функции доступен целый модуль `:dbg` (**http://erlang.org/doc/man/dbg.html**), упрощающий трассировку. Запустить его можно напрямую из прикрепленной оболочки или запустить другой узел и включить с него трассировку основной системы. Разберем это на примере.

Предположим, что у вас уже запущен узел todo. Запустите еще один узел:

```
$ iex --name tracer@127.0.0.1 --cookie todo --hidden
```

Теперь установите на узле tracer трассировку главного узла на все вызовы функций модуля Todo.Server:

```
iex(tracer@127.0.0.1)1> :dbg.tracer()
iex(tracer@127.0.0.1)2> :dbg.n(:'todo@127.0.0.1')
iex(tracer@127.0.0.1)3> :dbg.p(:all, [:call])
iex(tracer@127.0.0.1)4> :dbg.tp(Todo.Server, [])
```

Запуск процесса трассировки

Отслеживание только событий узла todo

Отслеживание вызовов функций во всех процессах

Установка шаблона трассировки всех функций процесса Todo.Server

Включив трассировку, вы можете отправить HTTP-запрос на получение записей списка Bob. В оболочке узла tracer вы получите следующий вывод:

```
(<10188.996.0>) call 'Elixir.Todo.Server':whereis(<<"bob">>)

(<10188.996.0>) call 'Elixir.Todo.Server':entries(
  <10188.958.0>,
  #{'__struct__' => 'Elixir.Date',calendar => 'Elixir.Calendar.ISO',
    day => 19, month => 12, year => 2013}
)

(<10188.958.0>) call 'Elixir.Todo.Server':handle_call(
  {
    entries,
    #{'__struct__' => 'Elixir.Date',calendar => 'Elixir.Calendar.ISO',
      day => 19,month => 12,year => 2013}
  },
  {
    <10188.996.0>,
    #Ref<10188.2468446053.3623092231.112404>},
    {<<"bob">>, #{'__struct__' => 'Elixir.Todo.List',
      auto_id => 1, entries => #{}}
  }
)
```

Еще раз напоминаю, что в системе, запущенной в промышленной эксплуатации, не стоит увлекаться трассировкой в целях сохранения производительности. Получив достаточное количество информации, вызовите функцию :dbg.stop_clear/0, чтобы остановить трассировку.

Это был совсем краткий обзор модуля :dbg, который умеет гораздо больше. Чтобы узнать о всех его возможностях, обратитесь к его документации.

Стоит также обратить внимание на стороннюю библиотеку Recon (**https://github.com/ferd/recon**), предоставляющую множество полезных функций для анализа узлов BEAM.

На этом мы заканчиваем изучение Elixir, Erlang и OTP. В книге были рассмотрены основные аспекты языка Elixir, базовые идиомы функционального программирования, модель конкурентности Erlang и наиболее часто используемые поведения OTP (GenServer, Supervisor и Application). По своему опыту могу сказать, что это самые необходимые элементы для разработки систем на Erlang и Elixir.

Разумеется, многие темы не были полностью раскрыты, и вы не должны на этом останавливаться. Для дальнейшего изучения вам могут понадобиться другие источники информации – книги, статьи, подкасты. Попробуйте начать с раздела «Learning» на официальном сайте Elixir (**https://elixir-lang.org/learning.html**).

Выводы

- Чтобы запустить систему, необходимо скомпилировать код всех ее модулей, запустить экземпляр BEAM с правильными путями загрузки файлов и все OTP-приложения.
- Проще всего запустить систему с помощью инструментов `IEx` и `mix`.
- OTP-релиз – это независимая система, состоящая только из необходимых для работы системы файлов – скомпилированных OTP-приложений и среды выполнения Erlang (опционально).
- Сборка OTP-релизов осуществляется довольно просто с помощью библиотеки `distillery`.
- Когда релиз уже запущен, к нему можно подключиться с помощью удаленной оболочки или прикрепиться к его консоли. Тогда вы сможете различными способами осуществлять взаимодействие с системой и поиск подробной информации о виртуальной машине и отдельных процессах.

Предметный указатель

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.a-planeta.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Саша Юрич

Elixir в действии

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Нестерова Н. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать офсетная.
Усл. печ. л. 30,55. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**