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.

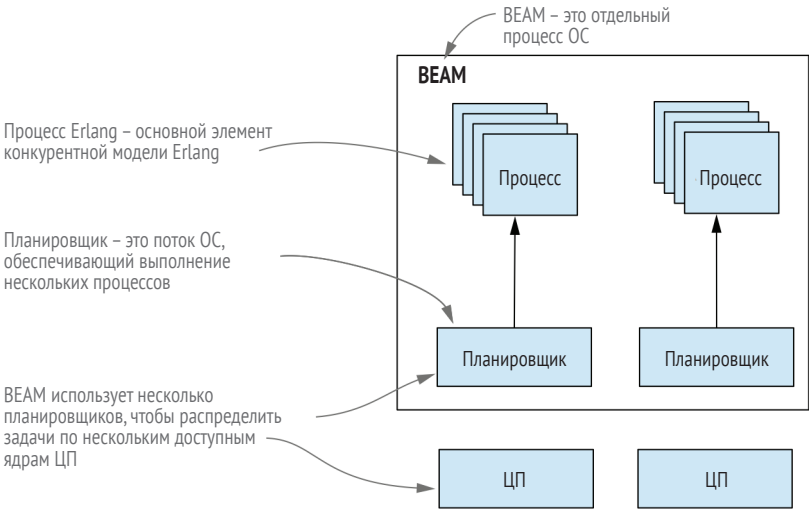

Рис. 1.1 ❖ Конкурентность в виртуальной машине Erlang

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.

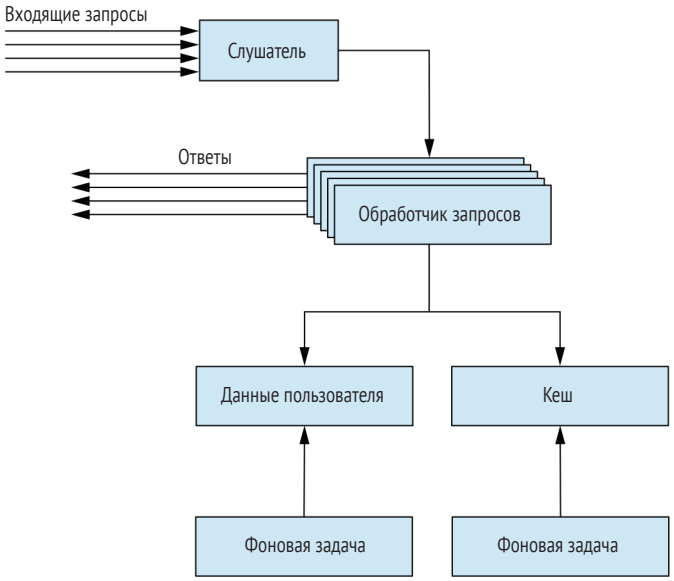

Рис. 1.2 ❖ Система на стороне сервера

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.